

Venice v 1.13.6



Primitives	Literals	Number	String	Char	Boolean	Keyword	Symbol	Nil	Just		
Collections	List Array	Vector ByteBuf	Set	Map	LazySeq	Stack	Queue	DelayQueue	Deque	CircularBuffer	DAG
Custom Types	Types	Protocols									
Concepts	Recursion	Destructuring									
Core Functions	Functions	Macros	Special Forms		Transducers		Namespaces		Exceptions		
Concurrency	Atoms	Locks	Locking	Futures	Promises	Delay	Agents	Volatiles	Parallel		
Scheduler	JDK Scheduler		Cron Scheduler								
Threads	ThreadLocal		Threads								
System	System	System Vars		REPL	Sandbox	Load Paths					
Shell	Shell System		Shell	Shebang							
Java	Java Interop		Java	MBeans							
Util	Math	Time	StopWatch	Regex	INET	CIDR					
I/O	I/O	File	File Watcher	Zip/GZip	Inter Process Communication						
Documents	JSON	JSON Lines		PDF	PDF Tools	CSV	XML	Excel	Images		
Modules	Kira Templates Keystores Aviron Limiter	Parsifal AsciiTable Aviron Queue	Grep Matrix Aviron Queue	Grep Shell Aviron Cyclor	Configuration Geo IP App	Component Swiss Cities QR Ref	ZipVault Mimetypes QR Bill	Fonts Ansi QR Code	Cryptography Logger Semver	RSA Aviron	
Build Tools	Gradle Wrapper		Gradle	Maven	Installer						
Test & Debug	Test	Tracing	Tap	Hexdump	Timing	Benchmark					
Database	JDBC Core		JDBC PostgreSQL		Chinook Dataset						
Web	Http Client J8		Tomcat WebApp Server			Ring	Multipart	SSE			
LLM	OpenAI	JTokkit									
Docker	Docker	Cargo	Cargo/ArangoDB		Cargo/Qdrant		Cargo/PostgreSQL				
License	License										
Others	Embedding in Java		Venice Doc		Markdown						

Primitives

Literals

Nil	nil
Boolean	true, false
Integer	150I, 1_000_000I, 0x1FFI
Long	1500, 1_000_000, 0x00A055FF
Float	3.569F, 2.0E+5F
Double	3.569, 2.0E+10
BigDecimal	6.897M, 2.345E+10M
BigInteger	1000N, 1_000_000N
Char	#\A, #\π, #\u03C0
	#\space, #\newline, #\return, #\tab, #\formfeed, #\backspace, #\lparen, #\rparen, #\quote
String	"abcd", "ab\"cd", "PI: \u03C0"
	""{ "age": 42 }""
String interpolation	"~{x}", ""~{x}""
	"~(inc x)", ""~(inc x)""

Numbers

Arithmetic	+ - * /
Convert	int long float double decimal bigint
Compare	== = not= < > <= >= compare
Test	zero? pos? neg? even? odd? number? int? long? float? double? decimal?
NaN/Infinite	nan? infinite?
BigDecimal	dec/add dec/sub dec/mul dec/div dec/scale

Strings

Create	str
Use	count compare empty-to-nil first last nth nfirst nlast seq rest butlast reverse shuffle str/subs str/nfirst str/nlast str/rest str/nrest str/butlast str/butnlast str/chars str/pos str/repeat str/reverse str/lorem-ipsum
Index	

Collections

Collections

Generic	count compare empty-to-nil empty into into! cons conj conj! remove repeat repeatedly cycle replace range group-by sort sort-by frequencies get-in seq reverse shuffle
Tests	empty? not-empty? distinct? coll? sequential? list? vector? set? sorted-set? mutable-set? map? sequential? hash-map? ordered-map? sorted-map? mutable-map? bytebuf?
Process	map map-indexed filter reduce tee group-by sort sort-by keep flatten docoll mapv run!

Lists

Create	() list list* mutable-list
Access	first second third fourth nth last peek rest butlast nfirst nlast index-of last-index-of sublist some
Modify	cons conj conj! rest pop into into! concat distinct dedupe partition partition-all partition-by partition-at interpose interleave cartesian-product combinations mapcat flatten sort sort-by take take-while take-last drop drop-while drop-last split-at split-with
Test	list? mutable-list? coll? sequential? every? not-every? any? not-any?

Vectors

Create	[] vector vector* mutable-vector mapv
Access	first second third nth last peek butlast rest nfirst nlast index-of last-index-of subvec some
Modify	cons conj conj! rest pop into into! concatv distinct dedupe partition partition-by partition-at interpose interleave cartesian-product combinations mapcat flatten sort sort-by take take-while take-last drop drop-while drop-last update update! assoc assoc! split-with

	<code>str/index-of</code> <code>str/index-of-char</code> <code>str/index-of-not-char</code> <code>str/last-index-of</code>
Split/Join	<code>str/split</code> <code>str/split-at</code> <code>str/split-lines</code> <code>str/split-columns</code> <code>str/join</code>
Replace	<code>str/replace-first</code> <code>str/replace-last</code> <code>str/replace-all</code>
Strip	<code>str/strip-start</code> <code>str/strip-end</code> <code>str/strip-indent</code> <code>str/strip-margin</code>
Linefeed	<code>str/platform-lf</code> <code>str/cr-lf</code> <code>str/crlf-to-lf</code>
Conversion	<code>str/lower-case</code> <code>str/upper-case</code>
Regex	<code>match?</code> <code>not-match?</code>
Trim	<code>str/trim</code> <code>str/trim-to-empty</code> <code>str/trim-to-nil</code> <code>str/trim-left</code> <code>str/trim-right</code>
Format	<code>str/format</code> <code>str/quote</code> <code>str/double-quote</code> <code>str/double-unquote</code> <code>str/align</code> <code>str/wrap</code> <code>str/expand</code> <code>str/truncate</code>
Hex	<code>str/hex-to-bytebuf</code> <code>str/bytebuf-to-hex</code> <code>str/format-bytebuf</code>
Bytebuf	<code>bytebuf-from-string</code> <code>bytebuf-to-string</code>
Encode/Decode	<code>str/encode-base64</code> <code>str/decode-base64</code> <code>str/encode-url</code> <code>str/decode-url</code> <code>str/escape-html</code> <code>str/escape-xml</code>
Test	<code>string?</code> <code>empty?</code> <code>not-empty?</code> <code>str/blank?</code> <code>str/not-blank?</code> <code>str/starts-with?</code> <code>str/ends-with?</code> <code>str/contains?</code> <code>str/equals-ignore-case?</code> <code>str/quoted?</code> <code>str/double-quoted?</code>
Test char	<code>str/char?</code> <code>str/digit?</code> <code>str/hexdigit?</code> <code>str/letter?</code> <code>str/whitespace?</code> <code>str/linefeed?</code> <code>str/lower-case?</code> <code>str/upper-case?</code>
Markdown	<code>str/markdown->text</code> <code>str/markdown->ansi</code>
UTF	<code>str/normalize-utf</code>
Validation	<code>str/valid-email-addr?</code>
Other	<code>str/levenshtein</code>
Chars	
Use	<code>char</code> <code>char?</code> <code>char-escaped</code> <code>char-literals</code>

Nested	<code>get-in</code> <code>assoc-in</code> <code>update-in</code> <code>dissoc-in</code>
Test	<code>vector?</code> <code>mutable-vector?</code> <code>coll?</code> <code>sequential?</code> <code>contains?</code> <code>not-contains?</code> <code>every?</code> <code>not-every?</code> <code>any?</code> <code>not-any?</code>
Sets	
Create	<code>#{} set sorted-set mutable-set</code>
Modify	<code>into</code> <code>into!</code> <code>cons</code> <code>cons!</code> <code>conj</code> <code>conj!</code> <code>disj</code>
Algebra	<code>difference</code> <code>union</code> <code>intersection</code> <code>subset?</code> <code>superset?</code>
Test	<code>set?</code> <code>sorted-set?</code> <code>mutable-set?</code> <code>coll?</code> <code>contains?</code> <code>not-contains?</code> <code>every?</code> <code>not-every?</code> <code>any?</code> <code>not-any?</code>
Maps	
Create	<code>{}</code> <code>hash-map</code> <code>ordered-map</code> <code>sorted-map</code> <code>mutable-map</code> <code>zipmap</code>
Access	<code>find</code> <code>get</code> <code>keys</code> <code>vals</code>
Modify	<code>cons</code> <code>conj</code> <code>conj!</code> <code>assoc</code> <code>assoc!</code> <code>update</code> <code>update!</code> <code>dissoc</code> <code>dissoc!</code> <code>into</code> <code>into!</code> <code>concat</code> <code>flatten</code> <code>filter-k</code> <code>filter-kv</code> <code>reduce-kv</code> <code>merge</code> <code>merge-with</code> <code>merge-deep</code> <code>map-invert</code> <code>map-keys</code> <code>map-vals</code> <code>select-keys</code>
Entries	<code>map-entry</code> <code>key</code> <code>val</code> <code>entries</code> <code>map-entry?</code>
Nested	<code>get-in</code> <code>assoc-in</code> <code>update-in</code> <code>dissoc-in</code>
Test	<code>map?</code> <code>hash-map?</code> <code>ordered-map?</code> <code>sorted-map?</code> <code>mutable-map?</code> <code>coll?</code> <code>contains?</code> <code>not-contains?</code>
Stack	
Create	<code>stack</code>
Access	<code>peek</code> <code>pop!</code> <code>push!</code> <code>into!</code> <code>conj!</code> <code>count</code>
Test	<code>empty?</code> <code>stack?</code>
Queue	
Create	<code>queue</code>
Access	<code>peek</code> <code>into!</code> <code>conj!</code> <code>count</code>
Sync	<code>put!</code> <code>take!</code>
Async	<code>offer!</code> <code>poll!</code>
Process	<code>docoll</code> <code>transduce</code> <code>reduce</code>

Conversion	<code>str</code> <code>str/lower-case</code> <code>str/upper-case</code>
Test char	<code>str/char?</code> <code>str/digit?</code> <code>str/letter?</code> <code>str/whitespace?</code> <code>str/linefeed?</code> <code>str/lower-case?</code> <code>str/upper-case?</code>
Booleans	
Boolean	<code>true</code> <code>false</code> <code>boolean</code> <code>not</code> <code>boolean?</code> <code>true?</code> <code>false?</code>
Keywords	
Keyword	<code>:a</code> <code>:blue</code> <code>keyword?</code> <code>keyword</code>
Symbols	
Symbol	<code>'a</code> <code>'blue</code> <code>symbol?</code> <code>qualified-symbol?</code> <code>symbol</code>
Nil	
Nil	<code>nil</code> <code>nil?</code> <code>some?</code>
Just	
Just	<code>just</code> <code>just?</code>

Byte Buffer

Create	<code>bytebuf</code> <code>bytebuf-allocate</code> <code>bytebuf-allocate-random</code> <code>bytebuf-byte-order!</code> <code>bytebuf-byte-order</code> <code>bytebuf-merge</code>
Capacity	<code>bytebuf-capacity</code> <code>bytebuf-remaining</code> <code>bytebuf-limit</code> <code>bytebuf-ensure-free-capacity!</code> <code>bytebuf-limit!</code> <code>count</code>
Search	<code>bytebuf-index-of</code>
String	<code>bytebuf-from-string</code> <code>bytebuf-to-string</code>
Test	<code>empty?</code> <code>not-empty?</code> <code>bytebuf?</code>
Use	<code>bytebuf-to-list</code> <code>bytebuf-sub</code> <code>bytebuf-pos</code> <code>bytebuf-pos!</code>
Read	<code>bytebuf-get-byte</code> <code>bytebuf-get-int</code> <code>bytebuf-get-long</code> <code>bytebuf-get-float</code> <code>bytebuf-get-double</code>
Write	<code>bytebuf-put-byte!</code> <code>bytebuf-put-int!</code> <code>bytebuf-put-long!</code> <code>bytebuf-put-float!</code> <code>bytebuf-put-double!</code> <code>bytebuf-put-buf!</code>
Base64	<code>str/encode-base64</code> <code>str/decode-base64</code>

Test	<code>empty?</code> <code>queue?</code>

DelayQueue

Create	<code>delay-queue</code>
Access	<code>peek</code> <code>count</code>
Sync	<code>put!</code> <code>take!</code>
Async	<code>poll!</code>
Test	<code>empty?</code> <code>delay-queue?</code>

Deque

Create	<code>deque</code>
Access	<code>peek</code> <code>into!</code> <code>conj!</code> <code>count</code>
Sync	<code>put!</code> <code>take!</code> <code>put-head!</code> <code>take-tail!</code>
Async	<code>offer!</code> <code>poll!</code> <code>offer-head!</code> <code>poll-tail!</code>
Process	<code>docoll</code> <code>transduce</code> <code>reduce</code>
Test	<code>empty?</code> <code>deque?</code>

CircularBuffer

Create	<code>circular-buffer</code>
Access	<code>peek</code> <code>poll!</code> <code>offer!</code> <code>take!</code> <code>put!</code> <code>count</code> <code>empty</code>
Test	<code>empty?</code> <code>circular-buffer?</code>

DAG (directed acyclic graph)

Create	<code>dag/dag</code> <code>dag/add-edges</code> <code>dag/add-nodes</code>
Access	<code>dag/nodes</code> <code>dag/edges</code> <code>dag/roots</code> <code>count</code>
Children	<code>dag/children</code> <code>dag/direct-children</code>
Parents	<code>dag/parents</code> <code>dag/direct-parents</code>
Sort	<code>dag/topological-sort</code> <code>dag/compare-fn</code>
Test	<code>dag/dag?</code> <code>dag/node?</code> <code>dag/edge?</code> <code>dag/parent-of?</code> <code>dag/child-of?</code> <code>empty?</code>

Lazy Sequences

Create	<code>lazy-seq</code>
Realize	<code>doall</code>
Test	<code>lazy-seq?</code>

Hex	str/hex-to-bytebuf	str/bytebuf-to-hex
	str/format-bytebuf	

Regex

General	regex/pattern	regex/matcher
	regex/reset	regex/pattern?
	regex/matches?	regex/matches-not?
	regex/matches	regex/group
	regex/groups	regex/count
	regex/find?	regex/find
	regex/find-all	regex/find+
	regex/find-all+	

Math

Arithmetic	inc	dec	min	max	clamp
	mod	mod-floor		abs	sgn
	negate	floor		ceil	sqr
	square	pow	exp	log	log2
	log10				
Util	digits				
Random	rand-long	rand-double			
	rand-bigint	rand-gaussian			
Trigonometry	math/to-radians	math/to-degrees			
	math/sin	math/cos	math/tan		
	math/asin	math/acos	math/atan		
Statistics	math/mean	math/median			
	math/quartiles	math/quantile			
	math/standard-deviation				
Algorithms	math/softmax				
Constants					
E	math/E				
PI	math/PI				

Transducers

Use	transduce				
Functions	map	map-indexed	filter	drop	
	drop-while	drop-last	take		
	take-while	take-last	keep		
	remove	dedupe	distinct	sorted	
	reverse	flatten	halt-when		
Reductions	rf-first	rf-last	rf-every?		
	rf-any?				
Early	reduced	reduced?	deref	deref?	

Arrays

Create	make-array	object-array	string-array		
	int-array	long-array	float-array		
	double-array				
Use	aget	aset	alength	asub	acopy
	amap				

Concurrency

Atoms	atom	atom?	deref	deref?
	reset!	swap!	swap-vals!	
	compare-and-set!		add-watch	
	remove-watch			
Locks	lock	lock?	acquire	
	try-acquire		release	locked?
Locking	locking			
Futures	future	future-task	future?	
	futures-fork	futures-wait		
	futures-thread-pool-info		done?	
	cancel	cancelled?	deref	
	deref?	realized?		
Promises	promise	promise?	deliver	
	deliver-ex	realized?		
	then-accept	then-accept-both		
	then-apply	then-combine		
	then-compose	when-complete		
	accept-either	apply-to-either		
	all-of	any-of	or-timeout	
	complete-on-timeout	timeout-after		
	done?	cancel	cancelled?	
Delay	delay	delay?	deref	deref?
	force	realized?		
Agents	agent	send	send-off	
	restart-agent	set-error-handler!		
	agent-error	await	await-for	
	shutdown-agents	shutdown-agents?		
	await-termination-agents			
	await-termination-agents?			
	agent-send-thread-pool-info			
	agent-send-off-thread-pool-info			
Volatiles	volatile	volatile?	deref	
	deref?	reset!	swap!	
ThreadLocal	thread-local	thread-local?		
	thread-local-clear			
	thread-local-map	assoc	dissoc	
	get	binding	def-dynamic	
Threads	thread	thread-id	thread-name	
	thread-daemon?	thread-interrupted?		
	thread-interrupted			
Parallel	pcalls	pmap	preduce	

Functions

Create	<code>fn</code> <code>defn</code> <code>defn-</code> <code>identity</code> <code>comp</code> <code>partial</code> <code>memoize</code> <code>juxt</code> <code>fn!l</code> <code>trampoline</code> <code>complement</code> <code>constantly</code> <code>every-pred</code> <code>any-pred</code>
Call	<code>apply</code> <code>-></code> <code>->></code>
Test	<code>fn?</code>
Misc	<code>nil?</code> <code>some?</code> <code>name</code> <code>qualified-name</code> <code>namespace</code> <code>fn-name</code> <code>callstack</code> <code>coalesce</code>
Load Source	<code>load-module</code> <code>load-file</code> <code>load-classpath-file</code> <code>read-string</code> <code>eval</code>
Environment	<code>set!</code> <code>resolve</code> <code>bound?</code> <code>var-get</code> <code>var-sym</code> <code>var-name</code> <code>var-ns</code> <code>var-sym-meta</code> <code>var-val-meta</code> <code>var-thread-local?</code> <code>var-local?</code> <code>var-global?</code> <code>name</code> <code>namespace</code>
Tree Walker	<code>prewalk</code> <code>postwalk</code> <code>prewalk-replace</code> <code>postwalk-replace</code>
Meta	<code>meta</code> <code>with-meta</code> <code>vary-meta</code>
Documentation	<code>doc</code> <code>finder</code> <code>modules</code>
Definiton	<code>fn-name</code> <code>fn-about</code> <code>fn-args</code> <code>fn-body</code> <code>fn-pre-conditions</code>
Syntax	<code>highlight</code>

Macros

Create	<code>def-</code> <code>defn</code> <code>defn-</code> <code>defmacro</code> <code>macroexpand</code> <code>macroexpand-all</code> <code>macro?</code>
Test	<code>macro?</code> <code>macroexpand-on-load?</code>
Quoting	<code>quote</code> <code>quasiquote</code>
Branch	<code>and</code> <code>or</code> <code>when</code> <code>when-not</code> <code>if-not</code> <code>if-let</code> <code>when-let</code> <code>letfn</code>
Conditions	<code>cond</code> <code>condp</code> <code>case</code>
Loop	<code>while</code> <code>dotimes</code> <code>doseq</code> <code>for</code> <code>list-comp</code>
Call	<code>doto</code> <code>-></code> <code>->></code> <code>-<></code> <code>as-></code> <code>cond-></code> <code>cond->></code> <code>some-></code> <code>some->></code>
Load Code	<code>load-module</code> <code>load-file</code> <code>load-classpath-file</code> <code>load-string</code> <code>loaded-modules</code>
Assert	<code>assert</code> <code>assert-false</code> <code>assert-eq</code> <code>assert-ne</code> <code>assert-throws</code>

Scheduler

JDK	<code>schedule-delay</code> <code>schedule-at-fixed-rate</code>
Cron	<code>cron/schedule-at</code> <code>cron/schedule-at-fixed-rate</code> <code>cron/schedule-at-round-times-in-day</code>

System

Venice	<code>version</code> <code>latest</code> <code>parse-version</code> <code>newer-version?</code>
Logo	<code>logo</code>
System	<code>system-prop</code> <code>system-env</code> <code>system-exit-code</code> <code>shutdown-hook</code> <code>charset-default-encoding</code>
Java	<code>java-home</code> <code>java-version</code> <code>java-version-info</code> <code>java-major-version</code> <code>java-vm-args</code> <code>java-source-location</code>
Auto-Run	<code>auto-run-jar</code>
Java VM	<code>pid</code> <code>gc</code> <code>total-memory</code> <code>used-memory</code>
OS	<code>os-type</code> <code>os-type?</code> <code>os-arch</code> <code>os-name</code> <code>os-version</code>
Jansi	<code>jansi-version</code>
Time	<code>current-time-millis</code> <code>nano-time</code> <code>format-nano-time</code> <code>format-micro-time</code> <code>format-milli-time</code>
Host	<code>host-name</code> <code>host-address</code> <code>ip-private?</code> <code>cpus</code> <code>byte-order</code>
User	<code>user-name</code> <code>io/user-home-dir</code>
Util	<code>uuid</code> <code>sleep</code>
Services	<code>service</code> <code>service?</code>

Shell System

Shell	<code>sh</code> <code>with-sh-dir</code> <code>with-sh-env</code> <code>with-sh-throw</code>
Shell Tools	<code>sh/open</code> <code>sh/pwd</code>
Process	<code>sh/alive?</code> <code>sh/kill</code> <code>sh/killall</code> <code>sh/pgrep</code> <code>sh/pargs</code> <code>sh/pkill</code> <code>sh/load-pid</code> <code>sh/which</code>

System Vars

System Vars	<code>*version*</code> <code>*newline*</code> <code>*loaded-modules*</code> <code>*loaded-files*</code> <code>*ns*</code> <code>*run-mode*</code> <code>*ansi-term*</code> <code>*ARGV*</code> <code>*out*</code> <code>*err*</code> <code>*in*</code>

	<code>assert-does-not-throw</code> <code>assert-throws-with-msg</code>
Util	<code>comment</code> <code>gensym</code> <code>time</code> <code>with-tmp-dir</code> <code>with-out-str</code> <code>with-err-str</code>
Profiling	<code>time</code> <code>perf</code>

Special Forms

Forms	<code>def</code> <code>defonce</code> <code>def-dynamic</code> <code>if</code> <code>do</code> <code>let</code> <code>binding</code> <code>fn</code> <code>set!</code>
Multi Methods	<code>defmulti</code> <code>defmethod</code>
Protocols	<code>defprotocol</code> <code>extend</code> <code>extends?</code>
Recursion	<code>loop</code> <code>recur</code> <code>tail-pos</code>
Exception	<code>throw</code> <code>try</code> <code>try-with</code>
Profiling	<code>dobench</code> <code>dorun</code> <code>prof</code>

Exceptions

Throw/Catch	<code>try</code> <code>try-with</code> <code>throw</code>
Create	<code>ex</code>
Test	<code>ex?</code> <code>ex-venice?</code>
Util	<code>ex-message</code> <code>ex-cause</code> <code>ex-value</code>
Stacktrace	<code>ex-venice-stacktrace</code> <code>ex-java-stacktrace</code>

Types

Util	<code>type</code> <code>supertype</code> <code>supertypes</code>
Test	<code>instance-of?</code> <code>deftype?</code>
Define	<code>deftype</code> <code>deftype-of</code> <code>deftype-or</code>
Create	<code>..:</code>
Describe	<code>deftype-describe</code>

Protocols

Core	<code>Object</code>

Namespace

Open	<code>ns</code>
Current	<code>*ns*</code>
Remove	<code>ns-unmap</code> <code>ns-remove</code>

Tap

Use	<code>tap></code>
Add	<code>add-tap</code>
Remove	<code>remove-tap</code> <code>clear-taps</code>

Time

Date	<code>time/date</code> <code>time/date?</code>
Local Date	<code>time/local-date</code> <code>time/local-date?</code> <code>time/local-date-parse</code>
Local Date Time	<code>time/local-date-time</code> <code>time/local-date-time?</code> <code>time/local-date-time-parse</code>
Zoned Date Time	<code>time/zoned-date-time</code> <code>time/zoned-date-time?</code> <code>time/zoned-date-time-parse</code>
Unix Timestamp	<code>time/unix-timestamp</code> <code>time/unix-timestamp-to-local-date-time</code>
Fields	<code>time/year</code> <code>time/month</code> <code>time/day-of-week</code> <code>time/day-of-month</code> <code>time/day-of-year</code> <code>time/hour</code> <code>time/minute</code> <code>time/second</code> <code>time/milli</code>
Fields etc	<code>time/length-of-year</code> <code>time/length-of-month</code> <code>time/first-day-of-month</code> <code>time/last-day-of-month</code>
Zone	<code>time/zone</code> <code>time/zone-offset</code>
Format	<code>time/formatter</code> <code>time/format</code>
Test	<code>time/after?</code> <code>time/not-after?</code> <code>time/before?</code> <code>time/not-before?</code> <code>time/within?</code> <code>time/leap-year?</code> <code>time/first-day-of-month?</code> <code>time/last-day-of-month?</code>
Miscellaneous	<code>time/between</code> <code>time/with-time</code> <code>time/plus</code> <code>time/minus</code> <code>time/period</code> <code>time/earliest</code> <code>time/latest</code>
Util	<code>time/zone-ids</code> <code>time/to-millis</code>

Stop Watch (Elapsed time measuring)

(load-module :stopwatch)

Stop Watch

Test	ns?		
Util	ns-list	namespace	
Alias	ns-alias	ns-aliases	ns-unalias
Meta	ns-meta	alter-ns-meta! reset-ns-meta!	

Java Interoperability

Java	. import java-unwrap-optional cast class		
Java	java->venice	venice->java	
Java	java-int-list	java-long-list	
	java-float-list	java-double-list	
	java-string-list		
	java-iterator-to-list		
	java-enumeration-to-list		
Proxify	proxify	java/as-runnable	
	java/as-callable	java/as-predicate	
	java/as-function	java/as-consumer	
	java/as-supplier	java/as-bipredicate	
	java/as-bifunction		
	java/as-biconsumer		
	java/as-unaryoperator		
	java/as-binaryoperator		
Test	java-obj?	java-optional?	enum?
	instance-of?	exists-class?	
Classes	class	class-of	class-name
	class-version		
Types	formal-type	remove-formal-type	
	class	supers	bases
Support	imports	stacktrace	classloader
	classloader-of		
JARs	jar-maven-manifest-version		
	java-package-version		
Modules	module-name		

REPL

Info	repl?	repl/info	
	repl/ansi-terminal?	repl/ad-hoc?	
	repl/complete-install?		
Terminal	repl/term-rows	repl/term-cols	
Dirs	repl/home-dir	repl/libs-dir	
	repl/tools-dir	repl/scripts-dir	
Config	repl/prompt!	repl/handler!	
	repl/color-theme	repl/color-theme!	
Env Vars	repl/cat-env	repl/get-env	
	repl/add-env	repl/remove-env	

```
stopwatch/create
stopwatch/create-time-limit
stopwatch/copy    stopwatch/start
stopwatch/stop    stopwatch/resume
stopwatch/split   stopwatch/expired?
stopwatch/elapsed
stopwatch/elapsed-formatted
```

I/O

to	print newline	println pr prn	printf	flush	
to-str	pr-str	with-out-str			
from	read-line	read-char			
classpath	io/load-classpath-resource io/classpath-resource?				
slurp	io/slurp io/slurp-stream io/read-line	io/slurp-lines io/slurp-reader io/read-char			
spit	io/spit io/spit-writer io/print-line	io/spit-stream io/print			
stream	io/copy-stream io/file-in-stream io/file-out-stream io/string-in-stream io/bytebuf-in-stream io/bytebuf-out-stream io/capturing-print-stream io/flush io/close				
stream wrap	io/wrap-os-with-buffered-writer io/wrap-os-with-print-writer io/wrap-is-with-buffered-reader io/wrap-is-with-gzip-input-stream io/wrap-os-with-gzip-output-stream io/wrap-is-with-inflater-input-stream io/wrap-os-with-deflater-output-stream				
reader/writer	io/buffered-reader io/buffered-writer io/string-reader io/string-writer io/flush io/close				
test	io/in-stream? io/reader?				io/out-stream? io/writer?
http	io/download		io/internet-avail?		
other	with-out-str io/mime-type		with-err-str io/default-charset		
vars	*out*	*err*	*in*		
logger	io/log		io/log-filehandler		

File I/O

Sandbox

Sandbox	<code>sandboxed?</code>	<code>sandbox/type</code> <code>sandbox/functions</code>

Loadpaths

Load Paths	<code>loadpath/paths</code> <code>loadpath/unrestricted?</code> <code>loadpath/normalize</code>

PDF

PDF	<code>pdf/render</code>	<code>pdf/text-to-pdf</code> <code>pdf/available?</code> <code>pdf/check-required-libs</code>
PDF Tools	<code>pdf/merge</code>	<code>pdf/copy</code> <code>pdf/pages</code> <code>pdf/watermark</code> <code>pdf/to-text</code> <code>pdf/extract-urls</code> <code>pdf/page-to-image</code> <code>pdf/page-count</code>

Install the required PDF libraries:

```
(do
  (load-module :pdf-install)
  (pdf-install/install :dir (repl/libs-dir)
    :silent false))
```

Zip/GZip

zip	<code>io/zip</code>	<code>io/zip-file</code>	<code>io/zip-list</code> <code>io/zip-list-entry-names</code> <code>io/zip-append</code> <code>io/zip-remove</code> <code>io/zip?</code> <code>io/unzip</code> <code>io/unzip-first</code> <code>io/unzip-nth</code> <code>io/unzip-all</code> <code>io/unzip-to-dir</code>
gzip	<code>io/gzip</code>	<code>io/gzip-to-stream</code>	<code>io/gzip?</code> <code>io/ungzip</code> <code>io/ungzip-to-stream</code>
zlib	<code>io/deflate</code>	<code>io/inflate</code>	

License

License	<code>license</code>	<code>license-all</code>

MBeans (Static and dynamic Java MBeans)

Server	<code>mbean/platform-mbean-server</code> <code>mbean/query-mbean-object-names</code> <code>mbean/object-name</code>
Server	<code>mbean/info</code> <code>mbean/attribute</code> <code>mbean/attribute!</code> <code>mbean/invoke</code>

file	<code>io/file</code> <code>io/file-parent</code> <code>io/file-name</code> <code>io/file-basename</code> <code>io/file-path</code> <code>io/file-path-slashify</code> <code>io/file-absolute</code> <code>io/file-canonical</code> <code>io/file-ext</code> <code>io/file-ext?</code> <code>io/file-size</code> <code>io/file-last-modified</code> <code>io/file-normalize-utf</code> <code>io/->path</code>

dir	<code>io/mkdir</code> <code>io/mkdirs</code>

slurp/spit	<code>io/slurp</code> <code>io/slurp-lines</code> <code>io/spit</code>

list	<code>io/list-files</code> <code>io/list-files-glob</code> <code>io/list-file-tree</code> <code>io/list-file-tree-lazy</code>

delete	<code>io/delete-file</code> <code>io/delete-files-glob</code> <code>io/delete-file-tree</code> <code>io/delete-file-on-exit</code>

copy	<code>io/copy-file</code> <code>io/copy-files-glob</code> <code>io/copy-file-tree</code>

move	<code>io/move-file</code> <code>io/move-files-glob</code>

touch	<code>io/touch-file</code>

truncate	<code>io/truncate-from-start-keep-lines</code>

permissions	<code>io/file-can-read?</code> <code>io/file-can-write?</code> <code>io/file-can-execute?</code> <code>io/file-set-readable</code> <code>io/file-set-writable</code> <code>io/file-set-executable</code>

links	<code>io/symbolic-link?</code> <code>io/create-symbolic-link</code> <code>io/create-hard-link</code>

test	<code>io/file?</code> <code>io/file-absolute?</code> <code>io/exists?</code> <code>io/exists-file?</code> <code>io/exists-dir?</code> <code>io/file-can-read?</code> <code>io/file-can-write?</code> <code>io/file-can-execute?</code> <code>io/file-hidden?</code> <code>io/symbolic-link?</code> <code>io/file-within-dir?</code> <code>io/path?</code>

glob	<code>io/glob-path-matcher</code> <code>io/file-matches-glob?</code> <code>io/list-files-glob</code> <code>io/copy-files-glob</code> <code>io/move-files-glob</code> <code>io/delete-files-glob</code>

disk space	<code>io/filesystem-total-space</code> <code>io/filesystem-usable-space</code>

URL/URI	<code>io/->url</code> <code>io/->uri</code>

temporary	<code>io/temp-file</code> <code>io/temp-dir</code> <code>io/tmp-dir</code>

user dir	<code>io/user-dir</code> <code>io/user-home-dir</code>

File Watcher

Register	mbean/register mbean/register-dynamic mbean/unregister
MX Beans	mbean/operating-system-mxbean mbean/runtime-mxbean mbean/memory-mxbean
Remote access	with-jmx-connection
JMX Connector	mbean/jmx-connector-server-start mbean/jmx-connector-server-stop mbean/jmx-connector-server-alive?

INET

Create	inet/inet-addr
Util	inet/inet-addr-to-bytes inet/inet-addr-from-bytes
Test	inet/ip4? inet/ip6? inet/linklocal-addr? inet/sitelocal-addr? inet/multicast-addr? inet/reachable?

CIDR (classless inter-domain routing)

CIDR	cidr/parse cidr/in-range? cidr/start-inet-addr cidr/end-inet-addr
CIDR Trie	cidr/trie cidr/size cidr/insert cidr/lookup cidr/lookup-reverse

JSON

read	json/read-str json/slurp
write	json/write-str json/spit
prettyfy	json/pretty-print

CSV

read	csv/read
write	csv/write csv/write-str

file watch	io/watch-dir io/registered-watch-dirs io/close-watcher io/await-for

IPC (Inter-Process-Communication)

Connection	ipc/server ipc/client ipc/copy ipc/close ipc/running?
Send	ipc/send ipc/send-async ipc/send-oneway
Pub / Sub	ipc/publish ipc/publish-async ipc/subscribe ipc/unsubscribe
Offer / Poll	ipc/offer ipc/offer-async ipc/poll ipc/poll-async
Authenticator	ipc/authenticator ipc/load-authenticator ipc/store-authenticator ipc/add-credentials ipc/remove-credentials ipc/clear-credentials
ACL	ipc/add-acl ipc/remove-acl ipc/default-acl
ACL CIDR	ipc/add-cidr ipc/remove-cidr ipc/clear-cidrs
Messages	ipc/text-message ipc/plain-text-message ipc/binary-message ipc/venice-message ipc/message-expired? ipc/message-field ipc/message->map ipc/message->json ipc/oneway? ipc/response-ok? ipc/response-err? ipc/message-size
Queues	ipc/create-queue ipc/create-temporary-queue ipc/remove-queue ipc/exists-queue? ipc/queue-status
Topics	ipc/create-topic ipc/remove-topic ipc/exists-topic?
Functions	ipc/create-function ipc/remove-function ipc/exists-function?
Benchmark	ipc/benchmark
Util	ipc/server-status ipc/server-thread-pool-statistics

Modules

Kira

Templating system

```
(load-module :kira)
```

Kira	kira/eval	kira/fn
Escape	kira/escape-xml	kira/escape-html

Cryptography

```
(load-module :crypt)
```

Ciphers	crypt/ciphers	crypt/max-key-size crypt/provider? crypt/add-bouncy-castle-provider
Hashes	crypt/md5-hash	crypt/sha1-hash crypt/sha512-hash crypt/pbkdf2-hash
Encrypt	crypt/encryptor-aes-256-gcm crypt/encryptor-chacha20 crypt/encryptor-chacha20-bouncycastle	
File hash	crypt/hash-file crypt/verify-file-hash	

RSA

```
(load-module :rsa)
```

Keys	rsa/generate-key-pair	rsa/key-pair rsa/keys rsa/private-key rsa/public-key
Key I/O	rsa/save-key-pair	rsa/load-key-pair rsa/load-key
Encryption	rsa/encrypt	rsa/decrypt
Signatures	rsa/sign	rsa/verify

Java Keystore

```
(load-module :keystores)
```

Load	keystores/load
Certificates	keystores/aliases keystores/certificate keystores/subject-dn keystores/issuer-dn keystores/parse-dn keystores/expiry-date keystores/expired?

Hexdump

```
(load-module :hexdump)
```

Hexdump	hexdump/dump

Semver

Semantic versioning

```
(load-module :semver)
```

Semver	semver/parse	semver/version
Validation	semver/valid?	semver/valid-format?
Test	semver/newer? semver/equal?	semver/older? semver/cmp

Geo IP

Geolocation mapping for IP addresses

```
(load-module :geoip)
```

Lookup	geoip/ip-to-country-resolver geoip/ip-to-country-loc-resolver geoip/ip-to-city-loc-resolver geoip/ip-to-city-loc-resolver-mem-optimized
Databases	geoip/download-google-country-db-to-csvfile geoip/download-maxmind-db-to-zipfile geoip/download-maxmind-db
DB Parser	geoip/parse-maxmind-country-ip-db geoip/parse-maxmind-city-ip-db geoip/parse-maxmind-country-db geoip/parse-maxmind-city-db
Util	geoip/build-maxmind-country-db-url geoip/build-maxmind-city-db-url geoip/map-location-to-numeric geoip/country-to-location-resolver geoip/addr-ranges->trie

Swiss Cities

Swiss Cities Geo / ZIP Codes Data

```
(load-module :swiss-cities)
```

Open	swiss-cities/download-city-data swiss-cities/load-city-data-from-classpath

JSON Lines

```
(load-module :jsonl)
```

read	jsonl/read-str	jsonl/slurp
	jsonl/lazy-seq-slurper	

write	jsonl/write-str	jsonl/spit
	jsonl/spitln	

Zip Vault

AES 256 encrypted and password protected zip file

```
(load-module :zipvault)
```

Create	zipvault/zip	zipvault/zip-folder

Add	zipvault/add-file	zipvault/add-files
	zipvault/add-folder	
	zipvault/add-empty-folder	
	zipvault/add-stream	

Remove	zipvault/remove-files

Extract	zipvault/extract-file
	zipvault/extract-all
	zipvault/extract-file-data

Util	zipvault/encrypted?
	zipvault/valid-zip-file?
	zipvault/entropy

XML

```
(load-module :xml)
```

XML	xml/parse-str	xml/parse	xml/path->
	xml/children	xml/text	

Java

```
(load-module :java)
```

Java	java/javadoc

Parsifal

A parser combinator

Parsifal is a port of Nate Young's Parsatron Clojure [parser combinators](#) project.

```
(load-module :parsifal)
```

Run	parsifal/run

Define	parsifal/defparser

Parsers	parsifal/any	parsifal/many
	parsifal/many1	parsifal/times
	parsifal/either	parsifal/choice
	parsifal/between	parsifal/>>

swiss-cities/slurp-city-data
swiss-cities/close
swiss-cities/open?

Query	swiss-cities/find-location swiss-cities/find-by-ortschaft swiss-cities/find-by-gemeinde swiss-cities/find-by-ortschaft-or-gemeinde swiss-cities/find-by-ortschaft-and-gemeinde swiss-cities/ortschaften swiss-cities/gemeinden swiss-cities/locations

PLZ	swiss-cities/find-by-plz swiss-cities/plz?

Distance	swiss-cities/distance

Excel

Read/Write Excel files

Venice is compatible with Apache POI 5.x.

To use charts with Excel documents Apache POI 5.2.0 or newer is required.

```
(load-module :excel)
```

Create/Open	excel/create	excel/open

Save	excel/write->file excel/write->stream excel/write->bytebuf

Sheet	excel/sheet	excel/sheet-count
	excel/sheet-name	
	excel/sheet-index	
	excel/sheet-row-range	
	excel/sheet-col-range	
	excel/add-sheet	excel/add-column
	excel/add-merge-region	
	excel/add-conditional-bg-color	
	excel/add-conditional-font-color	
	excel/add-conditional-border	
	excel/add-text-data-validation	
	excel/freeze-pane	
	excel/auto-size-columns	
	excel/auto-size-column	
	excel/hide-columns	
	excel/protect-sheet	

Sheet Layout	excel/print-layout excel/page-margins excel/header-margin excel/footer-margin	excel/header excel/footer
	excel/display-grid-lines	

Cells	excel/cell-empty?	excel/cell-lock
	excel/cell-locked?	
	excel/cell-hidden?	
	excel/cell-type	

Special Parsers	<code>parsifal/eof</code> <code>parsifal/never</code> <code>parsifal/always</code> <code>parsifal/lookahead</code> <code>parsifal/attempt</code>
Binding	<code>parsifal/let->></code>
Char Parsers	<code>parsifal/char</code> <code>parsifal/not-char</code> <code>parsifal/any-char</code> <code>parsifal/digit</code> <code>parsifal/hexdigit</code> <code>parsifal/letter</code> <code>parsifal/letter-or-digit</code> <code>parsifal/any-char-of</code> <code>parsifal/none-char-of</code> <code>parsifal/string</code>
Token Parsers	<code>parsifal/token</code>
Protocols	<code>parsifal/SourcePosition</code>
Line Info	<code>parsifal/lineno</code> <code>parsifal/pos</code>

Gradle Wrapper

Uses the 'gradlew.sh' or 'gradlew.bat' shell scripts from a Gradle project to run Gradle commands on the project. For projects not based on the Gradle Wrapper use the :gradle module instead.

```
(load-module :gradlew)
```

Gradle	<code>gradlew/version</code> <code>gradlew/run</code> <code>gradlew/run*</code>

Gradle

Uses the 'gradle.sh' or 'gradle.bat' shell scripts from a locally installed Gradle version to run Gradle commands on a project. For projects based on the Gradle Wrapper use the :gradlew module instead.

```
(load-module :gradle)
```

Gradle	<code>gradle/with-home</code> <code>gradle/version</code> <code>gradle/task</code>

Maven

```
(load-module :maven)
```

Artifact	<code>maven/parse-artifact</code> <code>maven/artifact-filename</code> <code>maven/artifact-uri</code>
Download	<code>maven/download</code> <code>maven/get</code>
Commands	<code>maven/home-dir</code> <code>maven/mvn</code> <code>maven/version</code> <code>maven/version-info</code> <code>maven/dependencies</code>
Install	<code>maven/install</code> <code>maven/uninstall</code>

Docker

```
(load-module :docker)
```

	<code>excel/cell-data-format-string</code> <code>excel/copy-cell-style</code> <code>excel/addr->string</code>
Rows	<code>excel/row-height</code> <code>excel/clear-row</code> <code>excel/delete-row</code> <code>excel/copy-row</code> <code>excel/copy-row-to-end</code> <code>excel/insert-empty-row</code>
Cols	<code>excel/col->string</code> <code>excel/col-hidden?</code> <code>excel/col-width</code>
Write Cells	<code>excel/write-data</code> <code>excel/write-items</code> <code>excel/write-item</code> <code>excel/write-value</code> <code>excel/write-values</code> <code>excel/write-values-keep-style</code>
Read Cells	<code>excel/read-val</code> <code>excel/read-string-val</code> <code>excel/read-boolean-val</code> <code>excel/read-long-val</code> <code>excel/read-double-val</code> <code>excel/read-date-val</code> <code>excel/read-datetime-val</code> <code>excel/read-error-code</code>
Formulas	<code>excel/evaluate-formulas</code> <code>excel/evaluate-formula</code> <code>excel/cell-formula-result-type</code> <code>excel/cell-formula</code> <code>excel/sum-formula</code> <code>excel/evaluate-formulas</code> <code>excel/remove-formula</code>
Styles	<code>excel/add-font</code> <code>excel/add-style</code> <code>excel/add-merge-region</code> <code>excel/add-conditional-bg-color</code> <code>excel/add-conditional-font-color</code> <code>excel/add-conditional-border</code> <code>excel/row-height</code> <code>excel/col-width</code> <code>excel/cell-style</code> <code>excel/cell-style-info</code> <code>excel/bg-color</code>
Images	<code>excel/add-image</code>
Comments	<code>excel/remove-comment</code>
Hyperlinks	<code>excel/add-url-hyperlink</code> <code>excel/add-email-hyperlink</code> <code>excel/remove-hyperlink</code>
Charts	<code>excel/add-line-chart</code> <code>excel/add-bar-chart</code> <code>excel/add-area-chart</code> <code>excel/add-pie-chart</code>
Charts Util	<code>excel/line-data-series</code> <code>excel/bar-data-series</code> <code>excel/area-data-series</code> <code>excel/pie-data-series</code> <code>excel/cell-address-range</code>

Install the required Apache POI 5.x libraries:

Docker	docker/version docker/cmd docker/debug
Images	docker/images docker/image-pull docker/rmi docker/image-rm docker/image-prune
Containers	docker/run docker/ps docker/start docker/stop docker/exec docker/exec& docker/rm docker/prune docker/cp docker/diff docker/pause docker/unpause docker/wait docker/logs
Volumes	docker/volume-list docker/volume-create docker/volume-inspect docker/volume-rm docker/volume-prune docker/volume-exists?
Utils	docker/images-query-by-repo docker/image-ready? docker/container-find-by-name docker/container-exists-with-name? docker/container-running-with-name? docker/container-start-by-name docker/container-stop-by-name docker/container-remove-by-name docker/container-status-by-name docker/container-exec-by-name docker/container-exec-by-name& docker/container-logs docker/container-purge-by-name docker/container-image-info-by-name

Cargo

Docker Testcontainers

Greatly simplifies starting/stopping docker containers. Depending on the state of the container pulls a new image, starts or runs the container and checks the logs if it has successfully started up, all this in one single command.

It's the base module for cargo modules like Cargo/ArangoDB, Cargo /PostgreSQL, and Cargo/Qdrant.

```
(load-module :cargo)
```

Cargo	cargo/start cargo/stop cargo/running? cargo/purge

Cargo ArangoDB

ArangoDB Testcontainers

```
(load-module :cargo-arangodb)
```

Lifecycle	cargo-arangodb/start cargo-arangodb/stop cargo-arangodb/running? cargo-arangodb/logs

Backup

```
(do
  (load-module :excel-install)
  (excel-install/install :dir (repl/libs-dir)
    :silent false))
```

Fonts

True Type Fonts

```
(load-module :fonts)
```

Download	fonts/download-font-family fonts/download-demo-fonts

Test

```
(load-module :test)
```

Define	test/deftest
Fixture	test/use-fixtures
Run	test/run-tests test/run-test-var test/successful?
Assert	assert assert-false assert-eq assert-ne assert-throws assert-does-not-throw assert-throws-with-msg

Configuration

Manages configurations with system property & env var support

```
(load-module :config)
```

Build	config/build
File	config/file config/resource
Env	config/env-var config/env
Properties	config/property-var config/properties

Component

Managing lifecycle and dependencies of components

```
(load-module :component)
```

Build	component/system-map component/system-using
Protocol	component/Component
Util	component/deps component/dep component/id

App

Venice application archive

```
cargo-arangodb/db-dump
cargo-arangodb/db-restore
cargo-arangodb/exists-db-dump?
cargo-arangodb/remove-db-dump
cargo-arangodb/list-db-dumps
cargo-arangodb/upload-db-dump
cargo-arangodb/download-db-dump
```

Cargo Qdrant Vector DB

Qdrant Testcontainers

```
(load-module :cargo-qdrant)
```

Lifecycle	<code>cargo-qdrant/start</code>	<code>cargo-qdrant/stop</code>
	<code>cargo-qdrant/running?</code>	
	<code>cargo-qdrant/logs</code>	

PostgreSQL DB

PostgreSQL Testcontainers

```
(load-module :cargo-postgresql)
```

Lifecycle	<code>cargo-postgresql/start</code>
	<code>cargo-postgresql/stop</code>
	<code>cargo-postgresql/running?</code>
	<code>cargo-postgresql/logs</code>

Tomcat

Embedded Tomcat WebApp Server

```
(load-module :tomcat)
```

Tomcat	<code>tomcat/start</code>	<code>tomcat/stop</code>
	<code>tomcat/destroy</code>	<code>tomcat/shutdown</code>
	<code>tomcat/state</code>	

Servlet	<code>tomcat/create-servlet</code>
	<code>tomcat/hello-world-servlet</code>

Install Java 3rd party libraries:

```
(do
  (load-module :tomcat-install)
  (tomcat-install/install :dir (repl/libs-dir)
                          :silent false))
```

Ring

```
(load-module :ring)
```

Servlet	<code>ring/create-servlet</code>

Routing	<code>ring/match-routes</code>

Utils	<code>ring-util/redirect</code>
	<code>ring-util/not-found-response</code>
	<code>ring-util/get-request-header</code>
	<code>ring-util/get-request-header-accept-mimetypes</code>
	<code>ring-util/get-request-parameters</code>

```
(load-module :app)
```

Build	<code>app/build</code>

Manifest	<code>app/manifest</code>

Benchmark

```
(load-module :benchmark)
```

Utils	<code>benchmark/benchmark</code>

Timing

Timing

```
(load-module :timing)
```

Timing	<code>timing/run</code>	<code>timing/elapsed</code>

Grep

Grep like search tool

```
(load-module :grep)
```

Grep	<code>grep/grep</code>	<code>grep/grep-zip</code>

QR-Reference

Create, parse, and format QR references according to the Swiss payment standards.

```
(load-module :qrref)
```

QR Ref	<code>qrref/create</code>	<code>qrref/format</code>
	<code>qrref/valid?</code>	<code>qrref/checksum</code>

QR-Bill

Create Swiss QR bills according to the Swiss payment standards.

```
(load-module :qrbill)
```

QR Bill	<code>qrbill/address</code>	<code>qrbill/bill</code>
	<code>qrbill/write</code>	

Install Java 3rd party libraries:

```
(do
  (load-module :qrbill-install)
  (qrbill-install/install :dir (repl/libs-dir)
                          :silent false))
```

QR-Codes

Encode and decode QR code images.

```
(load-module :qrcode)
```

QR Code	<code>qrcode/decode</code>	<code>qrcode/encode</code>

```
ring-util/get-request-parameter
ring-util/get-request-long-parameter
ring-util/html-request?
ring-util/json-request?
ring-util/parse-charset
ring-util/debug?
```

Middleware

```
ring-mw/mw-identity
ring-mw/mw-debug
ring-mw/mw-print-uri
ring-mw/mw-request-counter
ring-mw/mw-dump-request
ring-mw/mw-dump-response
```

Session

```
ring-session/session-invalidate
ring-session/session-clear
ring-session/session-id
ring-session/session-get-value
ring-session/session-remove-value
ring-session/session-creation-time
```

Multipart

```
ring-multipart/multipart-request?
ring-multipart/parts
ring-multipart/parts-delete-all
```

Tracing

Tracing functions

```
(load-module :trace)
```

Tracing

```
trace/trace    trace/trace-var
trace/untrace-var  trace/untrace-all
```

Test

```
trace/traced?    trace/traceable?
```

Util

```
trace/trace-str-limit
```

Tee

```
trace/tee->    trace/tee->>    trace/tee
```

Shell

Functions to deal with the operating system

```
(load-module :shell)
```

Open

```
shell/open    shell/open-macos-app
```

Process

```
shell/kill    shell/kill-forcibly
shell/wait-for-process-exit
shell/alive?  shell/pid    shell/nice
shell/process-handle
shell/process-handle?
shell/process-info    shell/processes
shell/processes-info
shell/descendant-processes
shell/parent-process
shell/get-process-pids
shell/exists-process-with-pid?
```

Util

```
shell/diff
```

JDBC Core

Install Java 3rd party libraries:

```
(do
  (load-module :qrcode-install)
  (qrcode-install/install :dir (repl/libs-dir)
    :silent false))
```

Ascii Table

Create and customize simple ASCII tables.

```
(load-module :ascii-table)
```

Render

```
ascii-table/render    ascii-table/print
```

Matrix

Simple matrix functions. To process large matrices use the "Efficient Java Matrix Library" (EJML) <http://ejml.org/wiki/> instead.

```
(load-module :matrix)
```

Matrix

```
matrix/validate    matrix/vector2d
matrix/empty?      matrix/rows
matrix/columns      matrix/row
matrix/column
```

Format

```
matrix/format
```

Elements

```
matrix/element    matrix/assoc-element
```

Add

```
matrix/add-column-at-start
matrix/add-column-at-end
matrix/add-row-at-start
matrix/add-row-at-end
```

Remove

```
matrix/remove-column
matrix/remove-row
```

LinAlg

```
matrix/transpose
```

Ansi

ANSI codes, styles, and colorization helper functions

```
(load-module :ansi)
```

Colors

```
ansi/fg-color    ansi/bg-color
```

Styles

```
ansi/style    ansi/ansi
ansi/with-ansi    ansi/without-ansi
```

Cursor

```
ansi/without-cursor
```

Progress

```
ansi/progress    ansi/progress-bar
```

Mimetypes

```
(load-module :mimetypes)
```

Mimetypes

```
mimetypes/probe-content-type
```


(load-module :jdbc-core)	
Create/Drop	jdbc-core/create-database jdbc-core/drop-database
Provider	jdbc-core/postgresql?
Meta Data	jdbc-core/meta-data jdbc-core/features jdbc-core/schemas jdbc-core/tables jdbc-core/columns
Connection	jdbc-core/closed?
Templates	jdbc-core/with-conn jdbc-core/with-tx
TX	jdbc-core/auto-commit? jdbc-core/auto-commit! jdbc-core/commit! jdbc-core/rollback! jdbc-core/tx-isolation jdbc-core/tx-isolation!
Statements	jdbc-core/create-statement jdbc-core/prepare-statement
Execute	jdbc-core/execute jdbc-core/execute-query jdbc-core/execute-query* jdbc-core/execute-update jdbc-core/generated-keys jdbc-core/count-rows
Prepared Stmt	jdbc-core/ps-clear-parameters jdbc-core/ps-string jdbc-core/ps-boolean jdbc-core/ps-int jdbc-core/ps-long jdbc-core/ps-float jdbc-core/ps-double jdbc-core/ps-decimal jdbc-core/ps-date jdbc-core/ps-timestamp jdbc-core/ps-clob jdbc-core/ps-blob
Result Set	jdbc-core/rs-first! jdbc-core/rs-next! jdbc-core/rs-last! jdbc-core/collect-result-set jdbc-core/render-query-result jdbc-core/print-query-result
Result Set Data	jdbc-core/rs-string jdbc-core/rs-boolean jdbc-core/rs-int jdbc-core/rs-long jdbc-core/rs-float jdbc-core/rs-double jdbc-core/rs-decimal jdbc-core/rs-date jdbc-core/rs-timestamp jdbc-core/rs-clob jdbc-core/rs-blob
Clob	

Multipart

(load-module :multipart)		
Multipart	multipart/render	multipart/parse multipart/http-content-type-header

SSE

Server Side Events	
(load-module :server-side-events)	
Render/Parse	server-side-events/render server-side-events/parse
Read	server-side-events/read-event server-side-events/read-events

HTTP Client J8

HTTP Client based on HttpURLConnection (Java 8+)	
(load-module :http-client-j8)	
HTTP Client	http-client-j8/send http-client-j8/upload-file http-client-j8/upload-multipart
Util	http-client-j8/build-url http-client-j8/create-authorization-bearer
Response	http-client-j8/slurp-response
SSE	http-client-j8/process-server-side-events
Tests	http-client-j8/status-ok-range? http-client-j8/status-redirect-range? http-client-j8/status-client-range? http-client-j8/status-server-error-range?

OpenAI Client

(load-module :openai-java)		
Client	openai-java/client	openai-java/close openai-java/version
Chat	openai-java/chat-completion openai-java/max-completion-tokens openai-java/add-user-message openai-java/add-user-message-with-files openai-java/add-assistant-message openai-java/json-response-format openai-java/add-function openai-java/execute	
Images	openai-java/generate-image	
Audio		

	jdbc-core/clob? jdbc-core/clob-length jdbc-core/clob-reader jdbc-core/clob-free
Blob	jdbc-core/blob? jdbc-core/blob-length jdbc-core/blob-in-stream jdbc-core/blob-bytebuf jdbc-core/blob-free

Installer

A simple artifact installer for Venice. This not a package manager!

```
(load-module :installer)
```

Install	installer/install installer/install-module installer/install-libs
Demo	installer/install-demo installer/install-demo-fonts
Clean	installer/clean

JDBC PostgreSQL

JDBC PostgreSQL support

```
(load-module :jdbc-postgresql)
```

Connection	jdbc-postgresql/create-connection
Meta Data	jdbc-postgresql/describe-table jdbc-postgresql/foreign-key-constraints

Install Java 3rd party libraries:

```
(do
  (load-module :postgresql-jdbc-install)
  (postgresql-jdbc-install/install :dir (repl
    /libs-dir)
    :silent false))
```

Chinook Dataset

Chinook dataset for PostgreSQL

```
(load-module :chinook-postgresql)
```

Data Model	chinook-postgresql/show-data-model
Data	chinook-postgresql/show-data
Load Data	chinook-postgresql/load-data
Download Data	chinook-postgresql/download-data

Aviron

ClamAV client

	openai-java/create-speech openai-java/transcribe openai-java/transcription-text
Response	openai-java/messages openai-java/image-binaries openai-java/image-urls openai-java/usage
Files	openai-java/list-files openai-java/create-file openai-java/create-binary-file openai-java/retrieve-file openai-java/delete-file openai-java/create-file-object openai-java/create-file-object-binary
Utils	openai-java/models openai-java/create-function-registry openai-java/register-function openai-java/format-usage

JTokkit

A tokenizer designed for use with OpenAI models

```
(load-module :jtokkit)
```

Encoding	jtokkit/encoding jtokkit/encoding-types jtokkit/model-types jtokkit/encode jtokkit/count-tokens

Install JTokkit 3rd party libraries:

```
(do
  (load-module :jtokkit-install)
  (jtokkit-install/install :dir (repl/libs-dir)
    :silent false))
```

Images

```
(load-module :images)
```

Load/Save	images/load images/save
Create/Copy	images/create images/copy
Properties	images/dimension images/alpha-channel?
File Formats	images/format-names
Transform	images/rotate images/flip images/crop images/pad images/resize-fit images/resize images/shear images/translate images/apply-ops images/convert-to-rgba images/convert-to-rgb
G2D	images/g2d images/anti-alias images/stroke images/fg-color

(load-module :aviron)	
Client	aviron/create-client aviron/print-config aviron/version
Scan	aviron/scan-stream aviron/scan-path aviron/scan-parallel
Scan Result	aviron/ok? aviron/virus? aviron/viruses
Admin	aviron/ping aviron/reachable? aviron/wait-for-operational-clamd aviron/shutdown-server aviron/clamav-version aviron/stats aviron/reload-virus-databases
Debug	aviron/last-command-run-details
Quarantine	aviron/quarantine-active? aviron/list-quarantine-files aviron/remove-quarantine-file aviron/remove-all-quarantine-files
Clamd Admin	aviron/clamd-pid aviron/clamd-cpu-limit aviron/clamd-cpu-limit-off aviron/cpus

Aviron Limiter

ClamAV client CPU Limiter

(load-module :aviron-limiter)	
Clamd Limiter	aviron-limiter/create-clamd-cpu-limiter aviron-limiter/clamd-activate-cpu-limit aviron-limiter/clamd-deactivate-cpu-limit aviron-limiter/clamd-last-seen-limit aviron-limiter/clamd-limit-for-timestamp aviron-limiter/format-profiles-as-table-by-hour
Dynamic Limiter	aviron-limiter/create-dynamic-cpu-limiter aviron-limiter/compute-dynamic-cpu-limit aviron-limiter/format-profiles-as-table-by-hour
CPU Profiles	aviron-limiter/create-cpu-profile aviron-limiter/get-cpu-profile-entries aviron-limiter/get-cpu-profile-limit

Logger

	images/bg-color images/get-clip images/set-clip images/get-clip-bounds
Transform	images/set-transform images/get-transform images/transform images/tx-identity images/tx-translate images/tx-scale images/tx-shear images/tx-rotate
Drawing	images/copy-area images/clear-rect images/draw-circle images/draw-oval images/draw-rect images/draw-round-rect images/draw-polygon images/draw-string images/draw-line images/draw-image
Filling	images/fill-circle images/fill-oval images/fill-rect images/fill-round-rect images/fill-polygon
Shapes	images/point images/rectangle images/polygon
Polygons	images/hexagon-poly images/rectangle-poly images/square-poly images/scale-points images/translate-points images/rotate-points

Aviron File Queue

ClamAV client file watcher queue

(load-module :aviron-queue)	
File Queue	aviron-queue/create aviron-queue/capacity aviron-queue/empty? aviron-queue/size aviron-queue/clear aviron-queue/remove aviron-queue/push aviron-queue/pop aviron-queue/overflow-count aviron-queue/overflow-reset

Aviron Dir Cyclcr

ClamAV client directory cyclcr

(load-module :aviron-cycler)	
Dir Cyclcr	aviron-cycler/create aviron-cycler/root-dir aviron-cycler/empty? aviron-cycler/size aviron-cycler/dirs aviron-cycler/dir-names aviron-cycler/next-dir aviron-cycler/peek-next-dir

```
(load-module :logger)
```

Logger	logger/logger logger/console-logger logger/file-logger

Level	logger/level logger/level-all

Rotation	logger/rotation-scheduler-running? logger/enable-auto-start-rotation- scheduler logger/start-rotation-scheduler logger/rotate logger/rotate-all logger/requires-rotation?

Util	logger/format-level logger/format-level

```
aviron-cycler/first?  
aviron-cycler/last?  
aviron-cycler/refresh  
aviron-cycler/last-dir-name  
aviron-cycler/last-dir-timestamp
```

Embedding in Java

Eval

```
import com.github.jlangch.venice.Venice;

public class Example {
    public static void main(String[] args) {
        final Venice venice = new Venice();

        final Long result = (Long)venice.eval("(+ 1 2)");
    }
}
```

Passing parameters

```
import com.github.jlangch.venice.Parameters;
import com.github.jlangch.venice.Venice;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        final Long result = (Long)venice.eval(
            "(+ x y 3)",
            Parameters.of("x", 6, "y", 3L));
    }
}
```

Dealing with Java objects

```
import java.awt.Point;
import com.github.jlangch.venice.Parameters;
import com.github.jlangch.venice.Venice;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        // returns a string: "Point=(x: 100.0, y: 200.0)"
        String ret = (String)venice.eval(
            "(let [x (:x point)                \n" +
            "      y (:y point)]                \n" +
            "  (str \"Point=(x: \" x \"\", y: \" y \"\")\"))",
            Parameters.of("point", new Point(100, 200)));

        // returns a java.awt.Point: [x=110,y=220]
        Point point = (Point)venice.eval(
            "(. :java.awt.Point :new (+ x 10) (+ y 20))",
            Parameters.of("x", 100, "y", 200));
    }
}
```

Precompiling

```
import com.github.jlangch.venice.IPreCompiled;
import com.github.jlangch.venice.Parameters;
import com.github.jlangch.venice.Venice;

public class Example {
```

```

public static void main(String[] args) {
    Venice venice = new Venice();

    IPreCompiled precompiled = venice.precompile("example", "(+ 1 x)");

    for(int ii=0; ii<100; ii++) {
        venice.eval(precompiled, Parameters.of("x", ii));
    }
}
}

```

Java Interop

```

import java.time.ZonedDateTime;
import com.github.jlangch.venice.Venice;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        Long val = (Long)venice.eval("( . :java.lang.Math :min 20 30)");

        ZonedDateTime ts = (ZonedDateTime)venice.eval(
            "(. (. :java.time.ZonedDateTime :now) :plusDays 5)");
    }
}

```

Sandbox

```

import com.github.jlangch.venice.SecurityException;
import com.github.jlangch.venice.Venice;
import com.github.jlangch.venice.javainterop.SandboxInterceptor;
import com.github.jlangch.venice.javainterop.SandboxRules;

public class SandboxExample {
    public static void main(final String[] args) {
        final SandboxInterceptor sandbox =
            new SandboxRules()
                // Venice functions: blacklist all unsafe functions
                .rejectAllUnsafeFunctions()

                // Venice functions: whitelist rules for print functions to offset
                // blacklist rules by individual functions
                .whitelistVeniceFunctions("*print*")

                .sandbox();

        final Venice venice = new Venice(sandbox);

        // => OK, 'println' is part of the unsafe functions, but enabled by the 2nd rule
        venice.eval("(println 100)");

        // => FAIL, 'read-line' is part of the unsafe functions
        try {
            venice.eval("(read-line)");
        }
        catch(SecurityException ex) {
            System.out.println("REJECTED: (read-line)");
        }
    }
}

```

Recursion

Recursion

Functional languages support **Tail Call Optimization (TCO)** to provide memory efficient recursion. Venice supports *automatic Tail Call Optimization* and *Self Recursion* through the `loop..recur` syntax. Self recursion is a way to mimic TCO.

In addition Venice provides the *trampoline* function for mutual recursion for more involved forms of recursion.

Self-Recursive Calls (loop - recur)

Venice self-recursive calls do not consume a new a stack frame for every new recursion iteration and have a constant memory usage. It's the only non-stack-consuming looping construct in Venice. To make it work the `recur` expression must be in *tail position*. This way Venice can turn the recursive `loop..recur` construct behind the scene into a plain loop.

Definition: The tail position is a position which an expression would return a value from. There are no more forms evaluated after the form in the tail position is evaluated.

Remember: Venice offers various alternative solutions to recursion to solve loops like `(+ 1 2 3 4 5 6)` to sum up a list of numbers or the powerful `reduce` function: `(reduce + [1 2 3 4 5])`. Many Venice functions accept an arbitrary number of arguments to prevent you from writing loops.

Example 1: Recursively sum up the numbers 0..n:

```
;; Definition:
;; sum 0 -> 0
;; sum n -> n + sum (n - 1)
(do
  (defn sum [n]
    ;; the transformed recursion uses an accumulator for intermediate results
    (loop [cnt n, acc 0]
      (if (zero? cnt)
        acc
        (recur (dec cnt) (+ acc cnt))))))

(sum 100000) ; => 5000050000
```

Example 2: Recursively compute the factorial of a number:

```
;; Definition:
;; factorial 1 -> 1
;; factorial n -> n * factorial (n - 1)
(do
  (defn factorial [x]
    ;; the transformed recursion uses an accumulator for intermediate results
    (loop [n x, acc 1N]
      (if (== n 1)
        acc
        (recur (dec n) (* acc n))))))

(factorial 5) ; => 120N
(factorial 10000) ; => 284625968091...00000N (35661 digits)
```

Example 3: Recursively compute the Fibonacci numbers (0 1 1 2 3 5 8 ...):

```
;; Definition:
;; fib 0 -> 0
;; fib 1 -> 1
;; fib n -> fib (n - 2) + fib (n - 1)
(do
  (defn fib [x]
    (loop [n x, a 0N, b 1N]
      (case n
        0 a
        1 b
        (recur (dec n) b (+ a b)))))

  (fib 6) ; => 8N
  (fib 100000) ; => 259740693472217...28746875N (20901 digits)
```

Recursion with lazy sequences

Example 1: Lazy Fibonacci number sequence computed by a recursive function:

```
(do
  (defn fib
    ([ ] (fib 0N 1N))
    ([a b] (cons a #(fib b (+ a b)))))

  (doall (take 7 (fib)))) ; => (0 1 1 2 3 5 8)
```

Example 2: Factorial numbers:

```
(do
  (defn factorial
    ([ ] (factorial 1 1N))
    ([x] (first (drop (dec x) (factorial))))
    ([n acc] (cons acc #(factorial (inc n) (* acc (inc n))))))

  (factorial 5) ; => 120N
  (factorial 10000) ; => 284625968091...00000N (35661 digits)
```

Mutually recursive calls (trampoline)

`trampoline` can be used to convert algorithms requiring mutual recursion without stack consumption. Calls `f`, if `f` returns a function, calls that function with no arguments, and continues to repeat, until the return value is not a function, then returns that non-function value.

The function `trampoline` is defined simplified as

```
(defn trampoline [f]
  (loop [f f]
    (let [ret (f)]
      (if (fn? ret) (recur ret) ret)))))
```

Examples:


```
(do
  (defn is-odd? [n]
    (if (zero? n) false #(is-even? (dec n))))

  (defn is-even? [n]
    (if (zero? n) true #(is-odd? (dec n))))

  (trampoline (is-odd? 10000)))
```

```
(do
  (defn factorial
    ([n] #(factorial n 1N))
    ([n acc] (if (< n 2)
                 acc
                 #(factorial (dec n) (* acc n)))))

  (trampoline (factorial 10000)))
```

Tail Call Optimization (TCO)

Venice has support for automatic *tail call optimization*. The recursive call must be in tail position.

```
(do
  (defn factorial
    ([n]      (factorial n 1N))
    ([n acc] (if (== n 1)
                 acc
                 (factorial (dec n) (* acc n)))))

  (factorial 5)      ; => 120N
  (factorial 10000) ; => 284625968091...00000N (35661 digits)
```

Recursion vs Folding

Tail call recursive functions, can always be written in terms of a reducing (folding) function. E.g.:

```
(do
  (defn factorial [n]
    ;; reducing factorial
    (reduce * 1N (range 1 (inc n))))

  (factorial 5)      ; => 120N
  (factorial 10000) ; => 284625968091...00000N (35661 digits)
```

But not all recursive functions can be transformed into a tail recursive function and translated into a loop. The [Ackermann's function](#) is such an example of a non [primitive recursive function](#) that can not be de-recursed into loops.

Recursion and Memoization

For some recursive algorithms *memoization* can speed up computation dramatically:

```
(do
  (def fibonacci
```

```

(memoize (fn [n]
           (if (< n 2)
               (max n 0)
               (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))

(fibonacci 30))

```

Please note that this naive memoization approach with recursive functions does **not** work as expected:

```

(do
  (defn fib-simple [n]
    (if (< n 2)
        (max n 0)
        (+ (fib-simple (- n 1)) (fib-simple (- n 2)))))

  (def fib-memoize (memoize fib-simple))

  (fib-memoize 30))

```

memoization is doing a good job in computing fibonacci numbers using simple recursion. It eliminates the recurring computation of the predecessors values.

Nevertheless there are recursive algorithms like the *Ackermann* function where memoization has to raise its arms.

Compare recursion efficiency

To see how efficient tail call optimization for recursion is we compare simple recursion with self recursion applied to computing Fibonacci numbers.

Note: all examples run with upfront macro expansion enabled.

```

(do
  (load-module :benchmark ['benchmark :as 'b])

  (defn fib-simple [n]
    (if (< n 2)
        n
        (+ (fib-simple (- n 1)) (fib-simple (- n 2)))))

  (defn fib-tco
    ([n]
     (fib-tco n 0N 1N))
    ([n a b]
     (case n
         0 a
         1 b
         (fib-tco (dec n) b (+ a b)))))

  (defn fib-loop-recur [x]
    (loop [n x, a 0N, b 1N]
      (case n
          0 a
          1 b
          (recur (dec n) b (+ a b)))))

  (def fib-memoize
    (memoize
     (fn [n]
       (if (< n 2)
           n
           (+ (fib-memoize (- n 1)) (fib-memoize (- n 2)))))))

```

```
;; (b/benchmark (fib-simple 30) 5 5)
;; (b/benchmark (fib-tco 30) 5000 1000)
;; (b/benchmark (fib-loop-recur 30) 5000 1000)
;; (time (fib-memoize 30)) ;; memoized functions can not be benchmarked
```

```
;; run on MacBook Air M2, with 'macroexpand' enabled
```

```
;; +-----+-----+
;; | (fib-simple 30)      | 1.171s |
;; | (fib-tco 30)        | 31.286μs |
;; | (fib-loop-recur 30) | 27.946μs |
;; | (fib-memoize 30)    | 2.540ms |
;; +-----+-----+
```

Destructuring

Destructuring

Sequential Destructuring

Sequential destructuring breaks up a sequential data structure as a Venice list or vector within a let binding

```
(do
  (let [[x y z] [1 2 3]]
    (println x y z))
  ;=> 1 2 3

  ;; for strings, the elements are destructured by character.
  (let [[x y z] "abc"]
    (println x y z))) ;; => a b c
```

or within function parameters

```
(do
  (defn position [[x y]]
    (println "x:" x "y:" y))

  (position [1 2])) ;; => x: 1 y: 2
```

The destructured collection must not be of same size as the number of binding names

```
(do
  (let [[a b c d e f] '(1 2 3)]
    (println a b c d e f)) ;=> 1 2 3 nil nil nil

  (let [[a b c] '(1 2 3 4 5 6 7 8 9)]
    (println a b c))) ;; => 1 2 3
```

Working with tail elements '&' and ignoring bindings '_'

```
(do
  (let [[a b c & z] '(1 2 3 4 5 6 7 8 9)]
    (println a b c z)) ;; => 1 2 3 (4 5 6 7 8 9)

  (let [[a _ b _ c & z] '(1 2 3 4 5 6 7 8 9)]
    (println a b c z))) ;; => 1 3 5 (6 7 8 9)
```

Binding the entire collection with ':as'

```
(do
  (let [[a b c & z :as all] '(1 2 3 4 5 6 7 8 9)]
    (println a b c z all))
  ;; => 1 2 3 (4 5 6 7 8 9) (1 2 3 4 5 6 7 8 9)
)
```

Nested bindings

```
(do
  (def line [[5 10] [10 20]])
  (let [[[x1 y1][x2 y2]] line]
    (printf "Line from (%d,%d) to (%d,%d)%n" x1 y1 x2 y2))
  ;; => "Line from (5,10) to (10,20)"
)
```

`:as` or `&` can be used at any level

```
(do
  (def line [[5 10] [10 20]])
  (let [[[a b :as group1] [c d :as group2]] line]
    (println a b group1)      ;; => 5 10 [5 10]
    (println c d group2)))    ;; => 10 20 [10 20]
```

Associative Destructuring

Associative destructuring breaks up an associative (key/value) data structure as a Venice map within a let binding.

```
(do
  (let [{a :a, b :b, c :c} {:a "A" :b "B" :d "D"}]
    (println a b c)))    ;; => A B nil
```

```
(do
  (def map_keyword {:a "A" :b "B" :c 3 :d 4})
  (def map_strings {"a" "A" "b" "B" "c" 3 "d" 4})

  (let [{:keys [a b c]} map_keyword]
    (println a b c))    ;; => A B 3

  (let [{:strs [a b c]} map_strings]
    (println a b c)))    ;; => A B 3
```

Binding the entire collection with `:as`

```
(do
  (def map_keyword {:a "A" :b "B" :c 3 :d 4})

  (let [{:keys [a b c] :as all} map_keyword]
    (println a b c all))
  ;; => A B 3 {:a A :b B :c 3 :d 4}
```

Binding with defaults `:or`

```
(do
  (defn configure [options]
    (let [{:keys [port debug verbose] :or {port 8000, debug false, verbose false}} options]
      (println "port =" port " debug =" debug " verbose =" verbose)))
    ;; => port 8000, debug false, verbose false

  (configure {:debug true}))
```

Nested destructuring

Associative destructuring can be nested and combined with sequential destructuring

```
(do
  (def users
    {:peter {:role "clerk"
              :branch "Zurich"
              :age 40}

     :magda {:role "head of HR"
              :branch "Bern"
              :age 45}

     :kurt  {:role "assistant"
              :branch "Lucerne"
              :age 32}})

  (let [{[:keys [role branch]] :peter} users]
    (println "Peter is a" role "located at" branch)))
;; => Peter is a clerk located at Zurich
```

Shebang

Shebang Scripts

With a little bit of sorcery a Venice script can be run as a Unix *Shebang* script.

REPL based Venice Shebang script

This *shebang* demo uses the Venice interpreter from an installed Venice REPL, giving the script access to all the 3rd party libraries installed within the REPL.

Prerequisites

1. The Venice REPL must be installed
2. The Venice version must be v1.12.26 or higher
3. MacOS or Linux operating systems

Example: shebang-demo.venice

```
#!/bin/sh

#_ ""

# Venice Shebang demo script

# The "run-script.sh" is provided by the installed Venice REPL. It
# starts a Venice interpreter on the REPL environment and runs this
# script.

REPL_HOME=/Users/juerg/Desktop/venice/

exec ${REPL_HOME}/run-script.sh "$0" "$@"

""

(println "Venice Shebang Demo")
(println)

(println "Args:" *ARGV*)

(println "Time:" (time/local-date-time))
```

Execution:

```
> chmod +x ./shebang-demo.venice
> ./shebang-demo.venice 1 2 3
Venice Shebang Demo

Args: (1 2 3)
Time: 2024-07-26T14:49:47.963
nil
```

Standalone Venice Shebang script

This *shebang* demo implicitly downloads the Venice library from the Maven repository when the script starts, provided the Venice library is not yet available in the installation directory.

Example: shebang-demo.venice

```
#!/bin/sh

#_ ""

# Venice Shebang demo script

VERSION=1.12.84           # Venice version to use
DIR=/tmp/venice           # Install dir
REPO=https://repo1.maven.org/maven2 # Maven repository

JAR=venice-${VERSION}.jar

[ -d ${DIR} ] || mkdir ${DIR}

if [ ! -f ${DIR}/${JAR} ]; then
    echo "Downloading ${JAR} from ${REPO} to ${DIR} ..."
    curl -s "${REPO}/com/github/jlangch/venice/${VERSION}/${JAR}" --output ${DIR}/${JAR}
fi

exec java -server -jar "${DIR}/${JAR}" -file "$@" "$@"

"""

(println "Venice Shebang Demo")
(println)

(println "Args:" *ARGV*)

(println "Time:" (time/local-date-time))
```

Execution:

```
> chmod +x ./shebang-demo.venice
> ./shebang-demo.venice 1 2 3
Venice Shebang Demo

Args: (1 2 3)
Time: 2024-07-26T14:49:47.963
nil
```

Running a Venice Shebang script as Unix cron job

Open the cron editor:

```
> export EDITOR=/bin/vi
> crontab -e
```

Add the following line to schedule the job:

```
30 23 * * Mon-Fri /bin/sh /home/foo/shebang-demo.venice 1 2 3
```


VeniceDoc

VeniceDoc is a documentation generator for the *Venice* language for generating API documentation in HTML format from *Venice* source code. It is used internally for generating the PDF and HTML cheatsheets. The function `doc` makes use of it to display the documentation for functions.

Example

Define a function `add` with documentation:

```
(defn
  ^{ :arglists '(
      "(add)", "(add x)", "(add x y)", "(add x y & more)"
    :doc
      """
      Returns the sum of the numbers.
      `(add)` returns 0.
      """
    :examples '(
      "(add)",
      "(add 1)",
      "(add 1 2)",
      "(add 1 2 3 4)"
    :see-also '(
      "+", "-", "x", "/" ) }

  add

  ([] 0)
  ([x] x)
  ([x y] (+ x y))
  ([x y & xs] (+ x y xs)))
```

Show its documentation from the REPL:

```
venice> (doc add)
```

REPL Output:

```
(add), (add x), (add x y), (add x y & more)

Returns the sum of the numbers. (add) returns 0.

EXAMPLES:
  (add)

  (add 1)

  (add 1 2)

  (add 1 2 3 4)
```

SEE ALSO:

`+`, `-`, `*`, `/`

VeniceDoc Format

The documentation is defined as a Venice metadata `map` :

```
{ :arglists '("add)", "(add x)")
  :doc "Returns the sum of the numbers."
  :examples '("add 1)", "(add 1 2)")
  :see-also '("+", "-", "*", "/") }
```

key	description
:arglist	the optional arglist, a list of variadic arg specs
:doc	the documentation in Venice markdown format
:examples	optional examples, a list of Venice scripts. Use triple quotes for multi-line scripts
:see-also	an optional list of cross referenced functions

Markdown

Venice Markdown

Headings

To create a heading, add one to four `#` symbols before the heading text. The number of `#` will determine the size of the heading.

```
# The largest heading
## The second largest heading
### The third largest heading
#### The fourth largest heading
```

Paragraphs and Line Breaks

```
A paragraph is simply one or more consecutive lines of text, separated by
one or more blank lines (a line containing nothing but spaces or tabs).
```

```
Within a paragraph line breaks can be added by placing a `pilcrow`
```

```
Line 1¶Line 2¶
Line 3
```

A paragraph is simply one or more consecutive lines of text, separated by one or more blank lines (a line containing nothing but spaces or tabs).

Within a paragraph line breaks can be added by placing a `¶`

Line 1
Line 2
Line 3

Styling

Venice markdown supports *italic*, **bold**, and ***bold-italic*** styling

```
This is *italic*, **bold**, and ***bold-italic*** styled text.
```

This is *italic*, **bold**, and ***bold-italic*** styled text.

Lists

Unordered List

```
* item 1
* item 2
* item 3
```

- item 1
- item 2

- item 3

Ordered List

```
1. item 1
2. item 2
3. item 3
```

1. item 1
2. item 2
3. item 3

Multiline list items with explicit line breaks:

```
* item 1
* item 2
  next line
  next line
* item 3
```

- item 1
- item 2
next line
next line
- item 3

Multiline list items with auto line breaks:

```
* item 1
* Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
  tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
  veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
  ea commodo consequat. Duis aute irure dolor in reprehenderit in
  voluptate velit esse cillum dolore eu fugiat nulla pariatur.
* item 3
```

- item 1
- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
- item 3

Links

Links are created by wrapping link text in brackets `[ ]`, and then wrapping the URL in parentheses `( )`.

```
[Venice](https://github.com/jlangch/venice)
```

[Venice](https://github.com/jlangch/venice)

Tables

A simple table

JAN	1
FEB	20
MAR	300

JAN	1
FEB	20
MAR	300

Column alignment

:---	:---:	---:
1	1	1
200	200	200
30000	30000	30000

1	1	1
200	200	200
30000	30000	30000

Width header

Col 1	Col 2	Col 3
:---	:---:	---:
1	1	1
200	200	200
30000	30000	30000

Col 1	Col 2	Col 3
1	1	1
200	200	200
30000	30000	30000

PDF rendered tables have always a width of 100%. In some use cases an additional left aligned column can trick the rendered table:

Col 1	Col 2	Col 3	
:---	:---:	---:	:---
1	1	1	
200	200	200	
30000	30000	30000	

Col 1	Col 2	Col 3
1	1	1
200	200	200
30000	30000	30000

Line breaks in cells

JAN	1 2 3
FEB	20
MAR	300

JAN	1
	2

	3
FEB	20
MAR	300

Column format using CSS styles

The Venice markdown supports a few CSS styles

Text alignment:

- `text-align: left`
- `text-align: center`
- `text-align: right`

Column width:

- `width: 15%`
- `width: 15pm`
- `width: 15em`
- `width: auto`

```
| Col 1 | Col 2 |
| [![text-align: left; width: 6em]] | [![text-align: left; width: 6em]] |
| 1 | 1 |
| 200 | 200 |
| 30000 | 30000 |
```

Col 1	Col 2
1	1
200	200
30000	30000

Code

Code can be called out within a text by enclosing it with single backticks.

```
To open a namespace use `(ns name)`.
```

To open a namespace use `(ns name)`.

Code block are enclosed with three backticks:

```
```
(defn hello []
 (println "Hello stranger"))

(hello)
```
```

producing

```
(defn hello []
  (println "Hello stranger"))

(hello)
```


Function Details

section top

#{}

Creates a set.

```
#{10 20 30}  
=> #{10 20 30}
```

section top

()

Creates a list.

```
'(10 20 30)  
=> (10 20 30)
```

section top

*

```
(*)  
(* x)  
(* x y)  
(* x y & more)
```

Returns the product of numbers. (*) returns 1

```
(*)  
=> 1
```

```
(* 4)  
=> 4
```

```
(* 4 3)  
=> 12
```

```
(* 4 3 2)  
=> 24
```

```
(* 4I 3I)  
=> 12I
```

```
(* 6.0 2)  
=> 12.0
```



```
( * 6 1.5M )
```

```
=> 9.0M
```

SEE ALSO

[+](#)

Returns the sum of the numbers. (+) returns 0.

[-](#)

If one number is supplied, returns the negation, else subtracts the numbers from x and returns the result.

[/](#)

If no denominators are supplied, returns 1/numerator, else returns numerator divided by all of the denominators.

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

ARGV

A list of the supplied command line arguments, or `nil` if the instantiator of the Venice instance decided not to make the command line arguments available.

ARGV

```
=> nil
```

[section](#)

[top](#)

ansi-term

`true` if Venice runs in an ANSI terminal, otherwise `false`

ansi-term

```
=> false
```

[section](#)

[top](#)

err

A `:java.io.PrintStream` object representing standard error for print operations.

Defaults to `System.err`, wrapped in an `PrintStream`.

`*err*` is a dynamic var. Any `:java.io.PrintStream` can be dynamically bound to it:

```
(binding [*err* print-stream]
 (println "text"))
```

SEE ALSO

[with-err-str](#)

Evaluates `exprs` in a context in which `*err*` is bound to a capturing output stream. Returns the string created by any nested printing ...

[*out*](#)

A `:java.io.PrintStream` object representing standard output for print operations.

[*in*](#)

A `:java.io.Reader` object representing standard input for read operations.

[section](#)

[top](#)

`*in*`

A `:java.io.Reader` object representing standard input for read operations.

Defaults to `System.in`, wrapped in an `InputStreamReader`.

`*in*` is a dynamic var. Any `:java.io.Reader` can be dynamically bound to it:

```
(binding [*in* reader]
 (read-line))
```

SEE ALSO

[read-line](#)

Without `arg` reads the next line from the stream that is the current value of `*in*`. With `arg` reads the next line from the passed stream ...

[read-char](#)

Without `arg` reads the next char from the stream that is the current value of `*in*`. With `arg` reads the next char from the passed stream ...

[*out*](#)

A `:java.io.PrintStream` object representing standard output for print operations.

[*err*](#)

A `:java.io.PrintStream` object representing standard error for print operations.

[section](#)

[top](#)

`*loaded-files*`

The loaded files

```
*loaded-files*
=> #{} 
```

[section](#)

[top](#)

`*loaded-modules*`

The loaded modules

```
*loaded-modules*
=> #{:tomcat :ring :ascii-canvas :csv :logger :jsonl :xchart :ring-multipart :ascii-table :java :xml :semver :
ring-mw :pretty-print :cargo :ring-session :ipc :cron :app :gradlew :images :hexdump :test :inet :maven :io :
timing :aviron-cycler :benchmark :aviron-limiter :str :core :openai :regex :mbean :installer :parsifal :shell :
multipart :jdbc-core :zipvault :math :http-client-j8 :openai-java :kira :mime-types :qrref :rsa :cargo-qdrant :
crypt :keystores :qrcode :cargo-postgresql :ring-util :aviron :matrix :docker :trace :fonts :chinook-postgresql
:json :cidr :aviron-queue :jetty :qrbill :geoip :server-side-events :jtokkit :grep :sandbox :jdbc-postgresql :
ansi :gradle :excel :http-client :component :pdf :cargo-arangodb :swiss-cities :time :config :stopwatch :ascii-
charts}
```

[section](#) [top](#)

newline

The system newline

```
*newline*
=> "\n"
```

[section](#) [top](#)

ns

The current namespace

```
*ns*
=> user

(do
  (ns test)
  *ns*)
=> test
```

[section](#) [top](#)

out

A `:java.io.PrintStream` object representing standard output for print operations.

Defaults to `System.out`, wrapped in an `PrintStream`.

`*out*` is a dynamic var. Any `:java.io.PrintStream` can be dynamically bound to it:

```
(binding [*out* print-stream]
  (println "text"))
```

SEE ALSO

[with-out-str](#)

Evaluates `exprs` in a context in which `*out*` is bound to a capturing output stream. Returns the string created by any nested printing ...

[*err*](#)

A `:java.io.PrintStream` object representing standard error for print operations.

[*in*](#)

A `java.io.Reader` object representing standard input for read operations.

[section](#)

[top](#)

`*run-mode*`

The current run-mode one of `:repl`, `:script`, `:app`

```
*run-mode*  
=> :script
```

[section](#)

[top](#)

`*version*`

The Venice version

```
*version*  
=> "0.0.0"
```

[section](#)

[top](#)

`+`

```
(+)  
(+ x)  
(+ x y)  
(+ x y & more)
```

Returns the sum of the numbers. `(+)` returns 0.

```
(+)  
=> 0
```

```
(+ 1)  
=> 1
```

```
(+ 1 2)  
=> 3
```

```
(+ 1 2 3 4)  
=> 10
```

```
(+ 1I 2I)  
=> 3I
```

```
(+ 1 2.5)  
=> 3.5
```

```
(+ 1 2.5M)  
=> 3.5M
```

SEE ALSO

–

If one number is supplied, returns the negation, else subtracts the numbers from x and returns the result.

*

Returns the product of numbers. (*) returns 1

/

If no denominators are supplied, returns 1/numerator, else returns numerator divided by all of the denominators.

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

–

```
(- x)
(- x y)
(- x y & more)
```

If one number is supplied, returns the negation, else subtracts the numbers from x and returns the result.

```
(- 4)
=> -4

(- 8 3 -2 -1)
=> 8

(- 5I 2I)
=> 3I

(- 8 2.5)
=> 5.5

(- 8 1.5M)
=> 6.5M
```

SEE ALSO

+

Returns the sum of the numbers. (+) returns 0.

*

Returns the product of numbers. (*) returns 1

/

If no denominators are supplied, returns 1/numerator, else returns numerator divided by all of the denominators.

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

[-<>](#)

(-<> x & forms)

Threads the x through the forms. Inserts x at position of the <> symbol of the first form, making a list of it if it is not a list already.

If there are more forms, inserts the first form at position of the <> symbol in second form, etc.

```
(-<> 5
  (+ <> 3)
  (/ 2 <>)
  (- <> 1))
=> -1
```

SEE ALSO

[->](#)

Threads the x through the forms. Inserts x as the second item in the first form, making a list of it if it is not a list already.

[->>](#)

Threads the x through the forms. Inserts x as the last item in the first form, making a list of it if it is not a list already.

[as->](#)

Binds name to expr, evaluates the first form in the lexical context of that binding, then binds name to that result, repeating for ...

[section](#)

[top](#)

[->](#)

(-> x & forms)

Threads the x through the forms. Inserts x as the second item in the first form, making a list of it if it is not a list already.

If there are more forms, inserts the first form as the second item in second form, etc.

```
(-> 5 (+ 3) (/ 2) (- 1)) ;; (- (/ (+ 5 3) 2) 1)
=> 3

(do
  (def person
    {:name "Peter Meier"
     :address {:street "Lindenstrasse 45"
               :city "Bern"
               :zip 3000}})

  (-> person :address :street)) ;; "Lindenstrasse 45"
=> "Lindenstrasse 45"
```

SEE ALSO

[->>](#)

Threads the x through the forms. Inserts x as the last item in the first form, making a list of it if it is not a list already.

[-<>](#)

Threads the x through the forms. Inserts x at position of the <> symbol of the first form, making a list of it if it is not a list already.

[as->](#)

Binds name to expr, evaluates the first form in the lexical context of that binding, then binds name to that result, repeating for ...

[section](#)

[top](#)

[->>](#)

```
(->> x & forms)
```

Threads the x through the forms. Inserts x as the last item in the first form, making a list of it if it is not a list already.

If there are more forms, inserts the first form as the last item in second form, etc.

```
(->> 5 (+ 3) (/ 32) (- 1))
=> -3

(->> [ {:a 1 :b 2} {:a 3 :b 4} {:a 5 :b 6} {:a 7 :b 8} ]
  (map (fn [x] (get x :b)))
  (filter (fn [x] (> x 4)))
  (map inc))
=> (7 9)

(->> [ {:a 1 :b 2} {:a 3 :b 4} {:a 5 :b 6} {:a 7 :b 8} ]
  (map #(get % :b))
  (filter #(> % 4))
  (map inc))
=> (7 9)
```

SEE ALSO

[->](#)

Threads the x through the forms. Inserts x as the second item in the first form, making a list of it if it is not a list already.

[-<>](#)

Threads the x through the forms. Inserts x at position of the <> symbol of the first form, making a list of it if it is not a list already.

[as->](#)

Binds name to expr, evaluates the first form in the lexical context of that binding, then binds name to that result, repeating for ...

-

```
(. classname :new args)
(. classname method-name args)
(. classname field-name)
(. classname :class)
(. object method-name args)
(. object field-name)
(. object :class)
```

Java interop. Calls a constructor or an class/object method or accesses a class/instance field. The function is sandboxed.

```
;; invoke constructor
(. :java.lang.Long :new 10)
=> 10

;; invoke static method
(. :java.time.ZonedDateTime :now)
=> 2026-06-04T17:20:03.804+02:00[Europe/Zurich]

;; invoke static method
(. :java.lang.Math :min 10 20)
=> 10

;; access static field
(. :java.lang.Math :PI)
=> 3.141592653589793

;; invoke method
(. (. :java.lang.Long :new 10) :toString)
=> "10"

;; get class name
(. :java.lang.Math :class)
=> class java.lang.Math

;; get class name
(. (. :java.io.File :new "/temp") :class)
=> class java.io.File
```

SEE ALSO

[import](#)

Imports one or multiple Java classes. Imports are bound to the current namespace.

[proxify](#)

Proxifies a Java interface to be passed as a Callback object to Java functions. The interface's methods are implemented by Venice functions.

[java/as-runnable](#)

Wraps the function `f` in a `java.lang.Runnable` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-callable](#)

Wraps the function `f` in a `java.util.concurrent.Callable` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

..

(.: type-name args*)

Instantiates a custom type.

Note: Venice implicitly creates a builder function suffixed with a dot:

```
(deftype :complex [real :long, imaginary :long])
(complex. 200 300)
```

For readability prefer `(complex. 200 300)` over `(.: :complex 100 200)`.

```
(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (def x (.: :complex 100 200))
  [(:real x) (:imaginary x)])
=> [100 200]
```

SEE ALSO

[deftype](#)

Defines a new custom record type for the name with the fields.

[deftype?](#)

Returns true if type is a custom type else false.

[deftype-of](#)

Defines a new custom wrapper type based on a base type.

[deftype-or](#)

Defines a new custom choice type.

[deftype-describe](#)

Describes a custom type.

[section](#)

[top](#)

/

```
(/ x)
(/ x y)
(/ x y & more)
```

If no denominators are supplied, returns 1/numerator, else returns numerator divided by all of the denominators.

```
(/ 2.0)
=> 0.5
```

```
(/ 12 2 3)
=> 2
```

```
(/ 12 3)
=> 4
```

```
(/ 12I 3I)
=> 4I
```

```
(/ 6.0 2)
```

```
=> 3.0
```

```
(/ 6 1.5M)
```

```
=> 4.0000000000000000M
```

SEE ALSO

[+](#)

Returns the sum of the numbers. (+) returns 0.

[-](#)

If one number is supplied, returns the negation, else subtracts the numbers from x and returns the result.

[*](#)

Returns the product of numbers. (*) returns 1

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

[<](#)

```
(< x y)
```

```
(< x y & more)
```

Returns true if the numbers are in monotonically increasing order, otherwise false.

```
(< 2 3)
```

```
=> true
```

```
(< 2 3.0)
```

```
=> true
```

```
(< 2 3.0M)
```

```
=> true
```

```
(< 2 3 4 5 6 7)
```

```
=> true
```

```
(let [x 10]
```

```
  (< 0 x 100))
```

```
=> true
```

SEE ALSO

[<=](#)

Returns true if the numbers are in monotonically non-decreasing order, otherwise false.

[>](#)

Returns true if the numbers are in monotonically decreasing order, otherwise false.

[>=](#)

Returns true if the numbers are in monotonically non-increasing order, otherwise false.

[section](#)

[top](#)

[<=](#)

```
(<= x y)
(<= x y & more)
```

Returns true if the numbers are in monotonically non-decreasing order, otherwise false.

```
(<= 2 3)
=> true
```

```
(<= 3 3)
=> true
```

```
(<= 2 3.0)
=> true
```

```
(<= 2 3.0M)
=> true
```

```
(<= 2 3 4 5 6 7)
=> true
```

```
(let [x 10]
  (<= 0 x 100))
=> true
```

SEE ALSO

[<](#)

Returns true if the numbers are in monotonically increasing order, otherwise false.

[>](#)

Returns true if the numbers are in monotonically decreasing order, otherwise false.

[>=](#)

Returns true if the numbers are in monotonically non-increasing order, otherwise false.

[section](#)

[top](#)

[=](#)

```
(= x)
(= x y)
(= x y & more)
```

Returns true if both operands have equivalent type and value

```
(= "abc" "abc")  
=> true
```

```
(= 0 0)  
=> true
```

```
(= 0 1)  
=> false
```

```
(= 0 0.0)  
=> false
```

```
(= 0 0.0M)  
=> false
```

```
(= "0" 0)  
=> false
```

```
(= 4)  
=> true
```

```
(= 4 4 4)  
=> true
```

SEE ALSO

[==](#)

Returns true if both operands have equivalent value.

[not=](#)

Same as (not (= x y))

[section](#)

[top](#)

[==](#)

```
(== x)  
(== x y)  
(== x y & more)
```

Returns true if both operands have equivalent value.

Numbers of different types can be checked for value equality.

```
(== "abc" "abc")  
=> true
```

```
(== 0 0)  
=> true
```

```
(== 0 1)  
=> false
```

```
(== 0 0.0)  
=> true
```

```
(== 0 0.0M)
=> true

(== "0" 0)
=> false

(== 4)
=> true

(== 4I 4 4.0 4.0M 4N)
=> true
```

SEE ALSO

[=](#)
Returns true if both operands have equivalent type and value

[not=](#)
Same as (not (= x y))

[section](#) [top](#)

[>](#)

```
(> x y)
(> x y & more)
```

Returns true if the numbers are in monotonically decreasing order, otherwise false.

```
(> 3 2)
=> true

(> 3 3)
=> false

(> 3.0 2)
=> true

(> 3.0M 2)
=> true

(> 7 6 5 4 3 2)
=> true
```

SEE ALSO

[<](#)
Returns true if the numbers are in monotonically increasing order, otherwise false.

[<=](#)
Returns true if the numbers are in monotonically non-decreasing order, otherwise false.

[>=](#)
Returns true if the numbers are in monotonically non-increasing order, otherwise false.

[section](#) [top](#)

>=

```
(>= x y)
(>= x y & more)
```

Returns true if the numbers are in monotonically non-increasing order, otherwise false.

```
(>= 3 2)
=> true
```

```
(>= 3 3)
=> true
```

```
(>= 3.0 2)
=> true
```

```
(>= 3.0M 2)
=> true
```

```
(>= 7 6 5 4 3 2)
=> true
```

SEE ALSO

[<](#)

Returns true if the numbers are in monotonically increasing order, otherwise false.

[<=](#)

Returns true if the numbers are in monotonically non-decreasing order, otherwise false.

[>](#)

Returns true if the numbers are in monotonically decreasing order, otherwise false.

[section](#)

[top](#)

Object

Defines a protocol to customize the `toString` and/or the `compareTo` function of custom datatypes.

Definition:

```
(defprotocol Object
  (toString [this] (to-str false this))
  (compareTo [this other] (compare this other)))
```

`compareTo` returns a negative integer, zero, or a positive integer as *this* value is less than, equal to, or greater than the *other* value.

```
(do
  (deftype :point [x :long, y :long]
    Object
    (toString [this] (str/format "[%s %s]" (:x this) (:y this)))
    (compareTo [self other] (. (:x self) :compareTo (:x other))))

  ; custom `toString`
  (println "toString:" (point. 1 2))

  ; custom `compareTo`: sort by 'x' ascending
  (println "compareTo:"
    (sort [(point. 2 100) (point. 3 101) (point. 1 102)])))
```

```
toString: [1 2]
compareTo: [[1 102] [2 100] [3 101]]
=> nil
```

SEE ALSO

[defprotocol](#)

Defines a new protocol with the supplied function specs.

[deftype](#)

Defines a new custom record type for the name with the fields.

[section](#)

[top](#)

[]

Creates a vector.

```
[10 20 30]
=> [10 20 30]
```

[section](#)

[top](#)

abs

(abs x)

Returns the absolute value of the number

```
(abs 10)
=> 10
```

```
(abs -10)
=> 10
```

```
(abs -10I)
=> 10I
```

```
(abs -10.1)
=> 10.1
```

```
(abs -10.12M)
=> 10.12M
```

SEE ALSO

[sgn](#)

sgn function for a number.

[negate](#)

Negates x

[section](#)

[top](#)

accept-either

```
(accept-either p p-other f)
```

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result as argument to the supplied function f.

```
(-> (promise (fn [] (sleep 200) 200))
    (accept-either (promise (fn [] (sleep 100) 100))
                   (fn [v] (println (+ v 1)))))
(deref))
101
=> nil
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function f with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function f with the two ...

[then-apply](#)

Applies a function f on the result of the previous stage of the promise p.

[then-combine](#)

Applies a function f to the result of the previous stage of promise p and the result of another promise p-other

[then-compose](#)

Composes the result of two promises. f receives the result of the first promise p and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise p with the same result or exception at this stage, that executes the action f. Passes the current stage's result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)

[top](#)

acopy

```
(acopy src src-pos dest dest-pos dest-len)
```

Copies an array from the src array, beginning at the specified position, to the specified position of the dest array. Returns the modified destination array

```
(acopy (long-array '(1 2 3 4 5)) 2 (long-array 20) 10 3)
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 4, 5, 0, 0, 0, 0, 0, 0, 0]
```

[section](#)

[top](#)

acquire


```
(acquire lock)
```

Acquires a lock, blocking until the lock is available.

```
(let [l (lock)]
  (acquire l)
  ;; do something
  (release l))
=> nil
```

SEE ALSO

[lock](#)

Creates a new lock object.

[try-acquire](#)

Acquires a lock within the given timeout time. Without a timeout returns immediately if the lock is not available.

[release](#)

Releases a lock.

[locked?](#)

Returns true if the lock is in use else false.

[section](#)

[top](#)

add-tap

```
(add-tap f)
```

adds f, a fn of one argument, to the tap set. This function will be called with anything sent via `tap>`.

This function may (briefly) block, and will never impede calls to `tap>`, but blocking indefinitely may cause tap values to be dropped.

Remember f in order to `remove-tap`

```
(add-tap println)
=> nil
```

SEE ALSO

[remove-tap](#)

Remove f from the tap set.

[clear-taps](#)

Removes all tap sets.

[tap>](#)

Sends x to any taps. Will not block. Returns true if there was room in the queue, false if not (x is dropped).

[section](#)

[top](#)

add-watch

```
(add-watch ref key fn)
```

Adds a watch function to an agent/atom reference. The watch fn must be a fn of 4 args: a key, the reference, its old-state, its new-state.

```
(do
  (def x (agent 10))
  (defn watcher [key ref old new]
    (println "watcher: " key))
  (add-watch x :test watcher))
=> nil
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)[top](#)

agent

(agent state & options)

Creates and returns an agent with an initial value of state and zero or more options.

Options:

- :error-handler handler-fn
- :error-mode mode-keyword
- :validator validate-fn

The `handler-fn` is called if an action throws an exception. It's a function taking two args the agent and the exception. The mode-keyword may be either `:continue` (the default) or `:fail`. The `validate-fn` must be nil or a side-effect-free fn of one argument, which will be passed the intended new state on any state change. If the new state is unacceptable, the `validate-fn` should return false or throw an exception.

```
(do
  (def x (agent 100))
  (send x + 5)
  (sleep 100)
  (deref x))
=> 105
```

SEE ALSO

[send](#)

Dispatch an action to an agent. Returns the agent immediately.

[send-off](#)

Dispatch a potentially blocking action to an agent. Returns the agent immediately.

[await](#)

Blocks the current thread (indefinitely) until all actions dispatched thus far (from this thread or agent) to the agents have occurred.

[await-for](#)

Blocks the current thread until all actions dispatched thus far (from this thread or agent) to the agents have occurred, or the timeout ...

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[set-error-handler!](#)

Sets the error-handler of an agent to handler-fn. If an action being run by the agent throws an exception handler-fn will be called ...

[agent-error](#)

Returns the exception thrown during an asynchronous action of the agent if the agent is failed. Returns nil if the agent is not failed.

[section](#)[top](#)

agent-error

(agent-error agent)

Returns the exception thrown during an asynchronous action of the agent if the agent is failed. Returns `nil` if the agent is not failed.

```
(do
  (def x (agent 100 :error-mode :fail))
  (send x (fn [n] (/ n 0)))
  (sleep 500)
  (agent-error x))
=> com.github.jlangch.venice.VncException: / by zero
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[set-error-handler!](#)

Sets the error-handler of an agent to handler-fn. If an action being run by the agent throws an exception handler-fn will be called ...

[agent-error-mode](#)

Returns the agent's error mode

[section](#)

[top](#)

agent-send-off-thread-pool-info

(agent-send-off-thread-pool-info)

Returns the thread pool info of the ThreadPoolExecutor serving agent send-off.

<i>core-pool-size</i>	the number of threads to keep in the pool, even if they are idle
<i>maximum-pool-size</i>	the maximum allowed number of threads
<i>current-pool-size</i>	the current number of threads in the pool
<i>largest-pool-size</i>	the largest number of threads that have ever simultaneously been in the pool
<i>active-thread-count</i>	the approximate number of threads that are actively executing tasks
<i>scheduled-task-count</i>	the approximate total number of tasks that have ever been scheduled for execution
<i>completed-task-count</i>	the approximate total number of tasks that have completed execution

```
(agent-send-off-thread-pool-info)
```

```
=> {:core-pool-size 0 :maximum-pool-size 2147483647 :current-pool-size 2 :largest-pool-size 2 :active-thread-count 0 :scheduled-task-count 10 :completed-task-count 10}
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[send-off](#)

Dispatch a potentially blocking action to an agent. Returns the agent immediately.

[section](#)

[top](#)

agent-send-thread-pool-info

(agent-send-thread-pool-info)

Returns the thread pool info of the ThreadPoolExecutor serving agent send.

<i>core-pool-size</i>	the number of threads to keep in the pool, even if they are idle
<i>maximum-pool-size</i>	the maximum allowed number of threads
<i>current-pool-size</i>	the current number of threads in the pool
<i>largest-pool-size</i>	the largest number of threads that have ever simultaneously been in the pool
<i>active-thread-count</i>	the approximate number of threads that are actively executing tasks
<i>scheduled-task-count</i>	the approximate total number of tasks that have ever been scheduled for execution
<i>completed-task-count</i>	the approximate total number of tasks that have completed execution

(agent-send-thread-pool-info)

```
=> {:core-pool-size 10 :maximum-pool-size 10 :current-pool-size 9 :largest-pool-size 9 :active-thread-count 0 :
scheduled-task-count 9 :completed-task-count 9}
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[send](#)

Dispatch an action to an agent. Returns the agent immediately.

[section](#)

[top](#)

aget

(aget array idx)

Returns the value at the index of an array of Java Objects

```
(aget (long-array '(1 2 3 4 5)) 1)
=> 2
```

[section](#)

[top](#)

alength

(alength array)

Returns the length of an array

```
(alength (long-array '(1 2 3 4 5)))
=> 5
```

[section](#)

[top](#)

all-of

```
(all-of p & ps)
```

Returns a new promise that is completed when all of the given promises complete. If any of the given promises complete exceptionally, then the returned promise also does so. Otherwise, the results, if any, of the given promises are not reflected in the returned promise, but may be obtained by inspecting them individually.

```
(-> (all-of (promise (fn [] (sleep 100) 1))
           (promise (fn [] (sleep 100) 2))
           (promise (fn [] (sleep 500) 3))))
      (deref))
=> nil
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[any-of](#)

Returns a new promise that is completed when any of the given promises complete, with the same result. Otherwise, if it completed exceptionally, ...

[section](#)[top](#)

alter-ns-meta!

```
(alter-ns-meta! n f & args)
```

Alters the metadata for a namespace. `f` must be free of side-effects.

```
(do
  (ns foo)
  (alter-ns-meta! foo assoc :a 1))
=> {:a 1}

(do
  (ns foo)
  (def n 'foo)
  (alter-ns-meta! (var-get n) assoc :a 1)
  (pr-str (ns-meta (var-get n))))
=> "{:a 1}"
```

SEE ALSO

[ns-meta](#)

Returns the meta data of the namespace `n` or `nil` if `n` is not an existing namespace

[reset-ns-meta!](#)

Resets the metadata for a namespace

[ns](#)

Opens a namespace.

[section](#)[top](#)

amap

```
(amap f arr)
```

Applies `f` to each item in the array `arr`. Returns a new array with the mapped values.

```
(str (amap (fn [x] (+ 1 x)) (long-array 6 0)))  
=> "[1, 1, 1, 1, 1, 1]"
```

[section](#)

[top](#)

and

```
(and x)  
(and x & next)
```

Ands the predicate forms

```
(and true true)  
=> true
```

```
(and true false)  
=> false
```

```
(and)  
=> true
```

SEE ALSO

[or](#)

Ors the predicate forms

[not](#)

Returns true if `x` is logical false, false otherwise.

[section](#)

[top](#)

ansi/ansi

```
(ansi style)
```

Output an ANSI escape code using a style key.

If `*use-ansi*` is bound to false, outputs an empty string instead of an ANSI code.

```
(println (str (ansi/ansi :blue) "foo"))
```

```
(println (str (ansi/ansi :underline) "foo"))
```

```
(println (str (ansi/ansi (ansi/fg-color 33)) "foo"))
```

```
(do  
  (dotimes [n 10] (println ">>>" n))  
  (sleep 1 :seconds)  
  (println (ansi/ansi :clear-screen))  
  (println "Hello"))
```

[section](#)

[top](#)

ansi/bg-color

```
(bg-color code)
(fg-color r g b)
```

Defines an extended background color from the 256-color extended color set. The code ranges from 0 to 255.

```
(ansi/bg-color 197)
```

SEE ALSO

[ansi/fg-color](#)

Defines an extended foreground color from the 256-color extended color set. The code ranges from 0 to 255.

[section](#)

[top](#)

ansi/fg-color

```
(fg-color code)
(fg-color r g b)
```

Defines an extended foreground color from the 256-color extended color set. The code ranges from 0 to 255.

The color range of a 256 color terminal consists of 4 parts in which case you actually get 258 colors:

- Color numbers 0 to 7 are the default terminal colors, the actual RGB value of which is not standardized and can often be configured.
- Color numbers 8 to 15 are the *bright* colors. Most of the time these are a lighter shade of the color with index - 8. They are also not standardized and can often be configured. Depending on terminal and shell, they are often used instead of or in conjunction with bold font faces.
- Color numbers 16 to 231 are RGB colors. These 216 colors are defined by 6 values on each of the three RGB axes. That is, instead of values 0 - 255, each color only ranges from 0 - 5.

The color number is then calculated like this

```
number = 16 + 36 * r + 6 * g + b
```

with `r`, `g` and `b` in the range 0 - 5.

- The color numbers 232 to 255 are grayscale with 24 shades of gray from dark to light.

```
(ansi/fg-color 197)
```

SEE ALSO

[ansi/bg-color](#)

Defines an extended background color from the 256-color extended color set. The code ranges from 0 to 255.

[section](#)

[top](#)

ansi/progress

```
(progress & options)
```

Returns a progress handler that renders the progress as a percentage string.

The returned progress handler takes two args:

- progress, a value 0..100 in :percent mode otherwise any value
- status, one of {start :progress :end :failed}

E.g: Download: 54%

Progress options:

:caption txt	A caption text. Defaults to empty.
:start-msg msg	A start message. Defaults to "{caption} started".
:end-msg msg	An end message. Defaults to "{caption} ok".
:end-col col	An end message ansi color code.
:failed-msg msg	A failed message. Defaults to "{caption} failed".
:failed-col col	A failed message ansi color code.
:mode m	A mode {:percent, :custom}. Defaults to :percent.

```
(let [pb (ansi/progress :caption "Test:")]
  (pb 0 :start)
  (doseq [x (range 0 101 10)]
    (pb x :progress)
    (sleep 1 :seconds))
  (pb 100 :end))

(io/download "https://foo.org/image.png"
  :binary true
  :user-agent "Mozilla"
  :progress-fn (ansi/progress :caption "Download:"))
```

[section](#)

[top](#)

ansi/progress-bar

(progress-bar & options)

Returns a progress handler that renders a progress bar.

The returned progress handler takes two args:

- progress (0..100%)
- status {start :progress :end :failed}

E.g:

- Download: [#####]
- Download: [#####] 70%

Progress bar options:

:caption txt	A caption text. Defaults to empty.
:width val	The width of the bar in chars. Defaults to 25.
:start-msg msg	A start message. Defaults to "{caption} started".
:end-msg msg	An end message. Defaults to "{caption} ok".
:end-col col	An end message ansi color code.
:failed-msg msg	A failed message. Defaults to "{caption} failed".
:failed-col col	A failed message ansi color code.
:show-percent bool	If true shows the percentage. Defaults to 'false'.

```
(let [pb (ansi/progress-bar
  :caption "Test:"
  :width 20
  :show-percent true)]
```



```
(pb 0 :start)
(doseq [x (range 0 101 10)]
  (pb x :progress)
  (sleep 1 :seconds))
(pb 100 :end))

(io/download "https://foo.org/image.png"
  :binary true
  :user-agent "Mozilla"
  :progress-fn (ansi/progress-bar
    :caption "Download:"
    :width 25
    :show-percent true))
```

[section](#)[top](#)

ansi/style

```
(style text styles)
```

Applies ANSI color and style to a text string.

```
(println (ansi/style "foo" :green))

(println (ansi/style "foo" :green :underline))

(println (ansi/style "foo" :green :bg-yellow :underline))

(println (ansi/style "foo" (ansi/fg-color 21) (ansi/bg-color 221) :underline))

(println (ansi/style "foo" nil))
```

[section](#)[top](#)

ansi/with-ansi

```
(with-ansi & forms)
```

Runs the given forms with the *use-ansi* variable temporarily bound to true, to enable the production of any ANSI color codes specified in the forms.

```
(ansi/with-ansi (println (ansi/style "foo" :green)))
```

[section](#)[top](#)

ansi/without-ansi

```
(without-ansi & forms)
```

Runs the given forms with the *use-ansi* variable temporarily bound to false, to suppress the production of any ANSI color codes specified in the forms.

```
(ansi/without-ansi (println (ansi/style "foo" :green)))
```

[section](#)[top](#)

ansi/without-cursor

```
(without-cursor & forms)
```

Runs the given forms with the cursor turned off.

[section](#)[top](#)

any-of

```
(any-of p & ps)
```

Returns a new promise that is completed when any of the given promises complete, with the same result. Otherwise, if it completed exceptionally, the returned promise also does so.

```
(-> (any-of (promise (fn [] (sleep 300) 1))
            (promise (fn [] (sleep 100) 2))
            (promise (fn [] (sleep 500) 3)))
    (deref))
=> 2
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[all-of](#)

Returns a new promise that is completed when all of the given promises complete. If any of the given promises complete exceptionally, ...

[section](#)[top](#)

any-pred

```
(any-pred p1 & p)
```

Takes a set of predicates and returns a function `f` that returns the first logical true value returned by one of its composing predicates against any of its arguments, else it returns logical false. Note that `f` is short-circuiting in that it will stop execution on the first argument that triggers a logical true result against the original predicates.

```
((any-pred number?) 1)
=> true
```

```
((any-pred number?) 1 "a")
=> true
```

```
((any-pred number? string?) 2 "a")
=> true
```

any?

```
(any? pred coll)
```

Returns true if the predicate is true for at least one collection item, false otherwise.

```
(any? number? nil)
=> false
```

```
(any? number? [])
=> false
```

```
(any? number? [1 :a :b])
=> true
```

```
(any? number? [1 2 3])
=> true
```

```
(any? #(== % 10) [10 20 30])
=> true
```

```
(any? #(>= % 10) [1 5 10])
=> true
```

SEE ALSO

[every?](#)

Returns true if coll is a collection and the predicate is true for all collection items, false otherwise.

[not-any?](#)

Returns false if the predicate is true for at least one collection item, true otherwise

[not-every?](#)

Returns true if coll is a collection and the predicate is not true for all collection items, false otherwise.

app/build

```
(app/build name main-file file-map dest-dir)
```

Creates a Venice application archive that can be distributed and executed as a single file.

The archive is stored as: {dest-dir}/{name}.zip

Returns a map with information on the created archive:

```
{ "file"   "{dest-dir}/{name}.zip",
  "name"   "{name}" }
```

Build example:

```
/staging
├─ billing.venice
├─ utils
│  └─ util.venice
└─ render.venice
```

```
└─ data
   └─ bill.template
   └─ logo.jpg
```

With these staged files the archive is built as:

```
(app/build
  "billing"
  "billing.venice"
  { "billing.venice"      "/staging/billing.venice"
    "utils/util.venice"   "/staging/utils/util.venice"
    "utils/render.venice" "/staging/utils/render.venice"
    "data/bill.template"  "/staging/data/bill.template"
    "data/logo.jpg"      "/staging/data/logo.jpg" }
  ".")
```

Loading Venice files works relative to the application. You can only load files that are in the app archive. If for instances "billing.venice" in the above example requires "utils/render.venice" just add `(load-file "utils/render.venice")` to "billing.venice".

The app can be run from the command line as:

```
> java -jar venice-1.13.6.jar -app billing.zip
```

Venice reads the archive and loads the archive's main file.

Or with additional Java libraries (all JARs in 'libs' dir):

```
> java -cp "libs/*" com.github.jlangch.venice.Launcher -app billing.zip
```

section

top

app/manifest

```
(app/manifest app)
```

Returns the manifest of a Venice application archive as a map.

section

top

apply

```
(apply f args* coll)
```

Applies f to all arguments composed of args and coll

```
(apply + [1 2 3])
=> 6
```

```
(apply + 1 2 [3 4 5])
=> 15
```

```
(apply str [1 2 3 4 5])
=> "12345"
```

```
(apply inc [1])
=> 2
```

section

top

apply-to-either

```
(apply-to-either p p-other f)
```

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result as argument to the supplied function f.

```
(-> (promise (fn [] (sleep 200) 200))
    (apply-to-either (promise (fn [] (sleep 100) 100))
                     (fn [v] (+ v 1))))
(deref))
=> 101
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function f with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function f with the two ...

[then-apply](#)

Applies a function f on the result of the previous stage of the promise p.

[then-combine](#)

Applies a function f to the result of the previous stage of promise p and the result of another promise p-other

[then-compose](#)

Composes the result of two promises. f receives the result of the first promise p and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise p with the same result or exception at this stage, that executes the action f. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

as->

```
(as-> expr name & forms)
```

Binds name to expr, evaluates the first form in the lexical context of that binding, then binds name to that result, repeating for each successive form, returning the result of the last form. This allows a value to thread into any argument position.

```
; allows to use arbitrary positioning of the argument
(as-> [ :foo :bar ] v
    (map name v)
    (first v)
    (str/subs v 1)))
=> "oo"
```

```
; allows the use of if statements in the thread
(as-> {:a 1 :b 2} m
  (update m :a #(+ % 10))
  (if true
    (update m :b #(+ % 10))
    m))
=> {:a 11 :b 12}
```

SEE ALSO

[->](#)

Threads the x through the forms. Inserts x as the second item in the first form, making a list of it if it is not a list already.

[->>](#)

Threads the x through the forms. Inserts x as the last item in the first form, making a list of it if it is not a list already.

[-<>](#)

Threads the x through the forms. Inserts x at position of the <> symbol of the first form, making a list of it if it is not a list already.

[section](#)

[top](#)

ascii-table/print

```
(ascii-table/print header data footer border padding)
(ascii-table/print columns data border padding)
```

Renders and prints an ascii table.

Actually does:

```
(println (ascii-table/render ...))
```

```
(do
  (load-module :ascii-table)
  (ascii-table/print ["head 1" "head 2"]
    ["1 1" "1 2"] ["2 1" "2 2"]
    ["foot 1" "foot 2"]
    :standard
    1))
```

```
+-----+-----+
| head 1 | head 2 |
+-----+-----+
| 1 1    | 1 2    |
+-----+-----+
| 2 1    | 2 2    |
+-----+-----+
| foot 1 | foot 2 |
+-----+-----+
=> nil
```

SEE ALSO

[ascii-table/render](#)

Renders an ascii table.

[section](#)

[top](#)

ascii-table/render


```

      :overflow :newline}
    :body  {:align :left
      :overflow :newline}
    :width 8}
  {:header {:text "head 2"
    :align :right
    :overflow :newline}
  :footer  {:text "6"
    :align :right
    :overflow :newline}
  :body    {:align :right
    :overflow :newline}
  :width 8}]
[["1" "2"] ["3" "4"]]
:double
1)))

```

head 1	head 2
1	2
3	4
4	6

=> nil

SEE ALSO

[ascii-table/print](#)

Renders and prints an ascii table.

section

top

aset

```
(aset array idx val)
```

Sets the value at the index of an array

```
(aset (long-array '(1 2 3 4 5)) 1 20)
=> [1, 20, 3, 4, 5]
```

section

top

assert

```
(assert expr)
(assert expr message)
```

Evaluates expr and throws an `:AssertionException` exception if it does not evaluate to logical true.

```
(assert (= 3 (+ 1 2)))
=> true
```



```
(assert (= 4 (+ 1 2)))  
=> AssertionError: Assert failed.  
Expression:  
(= 4 (+ 1 2))
```

SEE ALSO

[assert-false](#)

Evaluates expr and throws an :AssertionException exception if it does not evaluate to logical false.

[assert-eq](#)

Assert that expected and actual are equal. Throws an :AssertionException exception if they are not equal.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an :AssertionException exception if they are equal.

[assert-throws](#)

Evaluates expr and throws an :AssertionException exception if it does not throw the expected exception of type ex-type.

[assert-does-not-throw](#)

Evaluates expr and throws an :AssertionException exception if it does throw any kind of exception.

[test/deftest](#)

Defines a test function with no arguments.

[section](#)

[top](#)

assert-does-not-throw

```
(assert-does-not-throw expr)  
(assert-does-not-throw expr message)
```

Evaluates expr and throws an :AssertionException exception if it does throw any kind of exception.

```
(assert-does-not-throw (/ 2 1))  
=> true  
  
(assert-does-not-throw (/ 2 0))  
=> AssertionError: Assert failed.  
Unexpected exception: :com.github.jlangch.venice.VncException  
Expression:  
(/ 2 0)
```

SEE ALSO

[assert](#)

Evaluates expr and throws an :AssertionException exception if it does not evaluate to logical true.

[assert-false](#)

Evaluates expr and throws an :AssertionException exception if it does not evaluate to logical false.

[assert-eq](#)

Assert that expected and actual are equal. Throws an :AssertionException exception if they are not equal.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an :AssertionException exception if they are equal.

[assert-throws](#)

Evaluates expr and throws an :AssertionException exception if it does not throw the expected exception of type ex-type.

[test/deftest](#)

Defines a test function with no arguments.

assert-eq

```
(assert-eq expected actual)
(assert-eq expected actual message)
```

Assert that expected and actual are equal. Throws an `:AssertionException` exception if they are not equal.

```
(assert-eq 3 (+ 1 2))
=> true

(assert-eq 4 (+ 1 2))
=> AssertionError: Assert failed.
Expected: 4
Actual:   3
Expression:
(+ 1 2)
```

SEE ALSO

[assert](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical true.

[assert-false](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical false.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an `:AssertionException` exception if they are equal.

[assert-throws](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not throw the expected exception of type `ex-type`.

[assert-does-not-throw](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does throw any kind of exception.

[test/deftest](#)

Defines a test function with no arguments.

assert-false

```
(assert-false expr)
(assert-false expr message)
```

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical false.

```
(assert-false (= 3 (+ 1 3)))
=> true

(assert-false (= 4 (+ 1 3)))
=> AssertionError: Assert failed.
Expression:
(= 4 (+ 1 3))
```

SEE ALSO

[assert](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical true.

[assert-eq](#)

Assert that expected and actual are equal. Throws an `:AssertionException` exception if they are not equal.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an `:AssertionException` exception if they are equal.

[assert-throws](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not throw the expected exception of type `ex-type`.

[assert-does-not-throw](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does throw any kind of exception.

[test/deftest](#)

Defines a test function with no arguments.

[section](#)[top](#)

assert-ne

```
(assert-ne unexpected actual)
(assert-ne unexpected actual message)
```

Assert that unexpected and actual are not equal. Throws an `:AssertionException` exception if they are equal.

```
(assert-ne :foo :bar)
=> true

(assert-ne :foo :foo)
=> AssertionError: Assert failed.
Unexpected: :foo
Actual:    :foo
Expression:
:foo
```

SEE ALSO

[assert](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical true.

[assert-false](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical false.

[assert-eq](#)

Assert that expected and actual are equal. Throws an `:AssertionException` exception if they are not equal.

[assert-throws](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not throw the expected exception of type `ex-type`.

[assert-does-not-throw](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does throw any kind of exception.

[test/deftest](#)

Defines a test function with no arguments.

[section](#)[top](#)

assert-throws

```
(assert-throws ex-type expr)
(assert-throws ex-type expr message)
```

Evaluates `expr` and throws an `:AssertionException` exception if it does not throw the expected exception of type `ex-type`.

```
(assert-throws :VncException (/ 2 0))
=> true
```

```
(assert-throws :VncException (/ 2 1))
=> AssertionError: Assert failed.
Expected: :VncException
But no exception has been thrown!
Expression:
(/ 2 1)
```

SEE ALSO

[assert](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical true.

[assert-false](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical false.

[assert-eq](#)

Assert that expected and actual are equal. Throws an `:AssertionException` exception if they are not equal.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an `:AssertionException` exception if they are equal.

[assert-does-not-throw](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does throw any kind of exception.

[test/deftest](#)

Defines a test function with no arguments.

[section](#)

[top](#)

assert-throws-with-msg

```
(assert-throws-with-msg ex-type ex-msg-regexp expr)
(assert-throws-with-msg ex-type ex-msg-regexp expr message)
```

Evaluates `expr` and throws an `:AssertionException` exception if it does not throw the expected exception of type `ex-type`.

```
(assert-throws-with-msg :VncException #"/ by zero" (/ 2 0))
=> true
```

SEE ALSO

[assert](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical true.

[assert-false](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does not evaluate to logical false.

[assert-eq](#)

Assert that expected and actual are equal. Throws an `:AssertionException` exception if they are not equal.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an `:AssertionException` exception if they are equal.

[assert-does-not-throw](#)

Evaluates `expr` and throws an `:AssertionException` exception if it does throw any kind of exception.

[test/deftest](#)

Defines a test function with no arguments.

[section](#)[top](#)

assoc

```
(assoc coll key val)
(assoc coll key val & kvs)
```

When applied to a map, returns a new map of the same type, that contains the mapping of key(s) to val(s). When applied to a vector, returns a new vector that contains val at index. Note - index must be $\leq$ (count vector). When applied to a custom type, returns a new custom type with passed fields changed.

```
(assoc {} :a 1 :b 2)
=> {:a 1 :b 2}
```

```
(assoc nil :a 1 :b 2)
=> {:a 1 :b 2}
```

```
(assoc [1 2 3] 0 10)
=> [10 2 3]
```

```
(assoc [1 2 3] 3 10)
=> [1 2 3 10]
```

```
(assoc [1 2 3] 6 10)
=> [1 2 3 10]
```

```
(do
  (deftype :complex [real :long, imaginary :long])
  (def x (complex. 100 200))
  (def y (assoc x :real 110))
  (pr-str y))
=> "{:custom-type* :user/complex :real 110 :imaginary 200}"
```

SEE ALSO

[dissoc](#)

Returns a new coll of the same type, that does not contain a mapping for key(s)

[update](#)

Updates a value in an associative structure, where `k` is a key and `f` is a function that will take the old value and any supplied fargs ...

[section](#)[top](#)

assoc!

```
(assoc! coll key val)
(assoc! coll key val & kvs)
```

Associates key/vals with a mutable map, returns the map

```
(assoc! nil :a 1 :b 2)
=> {:a 1 :b 2}

(assoc! (mutable-map) :a 1 :b 2)
=> {:a 1 :b 2}

(assoc! (mutable-vector 1 2 3) 0 10)
=> [10 2 3]

(assoc! (mutable-vector 1 2 3) 3 10)
=> [1 2 3 10]

(assoc! (mutable-vector 1 2 3) 6 10)
=> [1 2 3 10]
```

SEE ALSO

[dissoc!](#)

Dissociates keys from a mutable map, returns the map

[update!](#)

Updates a value in a mutable associative structure, where k is a key and f is a function that will take the old value and any supplied ...

[section](#)

[top](#)

assoc-in

```
(assoc-in m ks v)
```

Associates a value in a nested associative structure, where ks is a sequence of keys and v is the new value and returns a new nested structure. If any levels do not exist, hash-maps or vectors will be created.

```
(do
  (def users [ {:name "James" :age 26}
                {:name "John" :age 43}])
  (assoc-in users [1 :age] 44))
=> [{:name "James" :age 26} {:name "John" :age 44}]

(do
  (def users [ {:name "James" :age 26}
                {:name "John" :age 43}])
  (assoc-in users [2] {:name "Jack" :age 19}))
=> [{:name "James" :age 26} {:name "John" :age 43} {:name "Jack" :age 19}]

(assoc-in [[1 2] [3 4]] [0 0] 9)
=> [[9 2] [3 4]]
```

[section](#)

[top](#)

asub

```
(asub array start len)
```

Returns a sub array

```
(asub (long-array '(1 2 3 4 5)) 2 3)
=> [3, 4, 5]
```

[section](#)

[top](#)

atom

```
(atom x)
(atom x & options)
```

Creates an atom with the initial value x.

Options:

- :meta metadata-map
- :validator validate-fn

If metadata-map is supplied, it will become the metadata on the atom. validate-fn must be nil or a side-effect-free fn of one argument, which will be passed the intended new state on any state change. If the new state is unacceptable, the validate-fn should return false or throw an exception.

```
(do
  (def counter (atom 0))
  (swap! counter inc)
  (deref counter))
=> 1
```

```
(do
  (def counter (atom 0))
  (reset! counter 9)
  @counter)
=> 9
```

SEE ALSO

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[reset!](#)

Sets the value of an atom or a volatile to newval without regard for the current value. Returns newval.

[swap!](#)

Atomically swaps the value of an atom or a volatile to be: (apply f current-value-of-box args). Note that f may be called multiple ...

[compare-and-set!](#)

Atomically sets the value of atom to newval if and only if the current value of the atom is identical to oldval. Returns true if set ...

[add-watch](#)

Adds a watch function to an agent/atom reference. The watch fn must be a fn of 4 args: a key, the reference, its old-state, its new-state.

[remove-watch](#)

Removes a watch function from an agent/atom reference.

[section](#)

[top](#)

atom?

```
(atom? x)
```

Returns true if x is an atom, otherwise false

```
(do
  (def counter (atom 0))
  (atom? counter))
=> true
```

[section](#)

[top](#)

auto-run-jar

(auto-run-jar script-name script-version script save-to)

Turns any standalone Venice script into an executable Venice JAR.

The `auto-run-jar` command takes a Venice JAR and copies it to a new JAR with a modified JAR manifest that executes the script when running the JAR with: `java -jar xxxx.jar`

The created JAR name will be: {script-name}.jar

The command line parameters will be available as `*ARGV*` list in the script.

```
;; run the created JAR: java -jar {path-to-jar}/example.jar
(let [script      ""(println "sum:" (+ 1 2))""
      script-name  "example"
      script-version "1.0"
      dir          "." ]
  (auto-run-jar script-name script-version script dir))

;; run the created JAR: java -jar {path-to-jar}/example.jar 1 2
(let [script      ""
      (println "sum:" (+ (long (first *ARGV*))
                          (long (second *ARGV*))))
      "" ]
      script-name  "example"
      script-version "1.0"
      dir          "." ]
  (auto-run-jar script-name script-version script dir))

;; create the executable JAR from a Venice script file:
;; (io/spit "./example.venice"
;;          ""(println "sum:" (+ 1 2))""
;;          :encoding :utf-8)
;; run the created JAR: java -jar {path-to-jar}/example.jar
(let [script      (io/slurp "./example.venice" :encoding :utf-8)
      script-name  "example"
      script-version "1.0"
      dir          "." ]
  (auto-run-jar script-name script-version script dir))
```

[section](#)

[top](#)

aviron-cycler/create

```
(create root-dir)
(create root-dir state-file)
```


Create a DirCycler on a directory with an optional state file.

If the state file is given (it must not necessarily exist at construction time) the cycler saves its state automatically to this file.

If the state file exists at construction time the cycler loads the state from the file and proceeds where it has left off.

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/root-dir demo-fs)]
    (aviron-cycler/create filestore-dir)))

(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        root-dir     (aviron-demo-filestore/root-dir demo-fs)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        state-file    (io/file root-dir "cyclr.state")]
    (aviron-cycler/create filestore-dir state-file)))
```

SEE ALSO

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/dir-names

```
(dir-names cycler)
```

Returns a list of the managed subdir names

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
```

```
filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
cyclr         (aviron-cycler/create filestore-dir)]
(aviron-cycler/dir-names cyclr)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cyclr or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cyclr. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)[top](#)

aviron-cycler/dirs

```
(dirs cyclr)
```

Returns a list of the managed subdirs

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cyclr         (aviron-cycler/create filestore-dir)]
    (aviron-cycler/dirs cyclr)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cyclr or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/empty?

```
(empty? cycler)
```

Returns `true` if the number of sub dirs is zero else `false`

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/size empty?)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/first?

```
(first? cycler)
```

Returns `true` if the current sub dir is the first one else `false`

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/first? cycler)
    (aviron-cycler/next-dir cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/last-dir-name

```
(last-dir-name cycler)
```

Returns the last sub dir name in the cycle or `nil` if there was not yet a call to `next`

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/next-dir cycler)
    (aviron-cycler/last-dir-name cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/last-dir-timestamp

`(last-dir-timestamp cycler)`

Returns the timestamp the last sub dir was run or `nil` if there was no run yet.

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler        (aviron-cycler/create filestore-dir)]
    (aviron-cycler/next-dir cycler)
    (aviron-cycler/last-dir-timestamp cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[section](#)

[top](#)

aviron-cycler/last?

(last? cycler)

Returns `true` if the current sub dir is the last one (before starting over again with the first one) else `false`

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/next-dir cycler)
    (aviron-cycler/last? cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/next-dir

(next-dir cycler)

Returns the next dir in the cycle or `nil` if the root dir does not have any sub dirs

```
(do
  (load-module :aviron-cycler)
```

```
(load-module :aviron-demo-filestore)
(let [demo-fs      (aviron-demo-filestore/create)
      _           (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
      filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
      cyclcr       (aviron-cycler/create filestore-dir)]
  (aviron-cycler/next-dir cyclcr)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/peek-next-dir

```
(peek-next-dir cycler)
```

Returns the next dir in the cycle without advancing the cycler or `nil` if the root dir does not have any sub dirs

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _           (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cyclcr       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/peek-next-dir cyclcr)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/refresh

```
(refresh cycler)
```

Refreshes the cycler. Rescans all sub dirs.

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/refresh cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/root-dir

```
(root-dir cycler)
```


Returns the root dir

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/root-dir cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-cycler/size

```
(size cycler)
```

Returns the number of sub dirs

```
(do
  (load-module :aviron-cycler)
  (load-module :aviron-demo-filestore)
  (let [demo-fs      (aviron-demo-filestore/create)
        _            (aviron-demo-filestore/populate-with-demo-files demo-fs 3 5)
        filestore-dir (aviron-demo-filestore/filestore-dir demo-fs)
        cycler       (aviron-cycler/create filestore-dir)]
    (aviron-cycler/size cycler)))
```

SEE ALSO

[aviron-cycler/create](#)

Create a DirCycler on a directory with an optional state file.

[aviron-cycler/root-dir](#)

Returns the root dir

[aviron-cycler/dirs](#)

Returns a list of the managed subdirs

[aviron-cycler/dir-names](#)

Returns a list of the managed subdir names

[aviron-cycler/next-dir](#)

Returns the next dir in the cycle or nil if the root dir does not have any sub dirs

[aviron-cycler/peek-next-dir](#)

Returns the next dir in the cycle without advancing the cycler or nil if the root dir does not have any sub dirs

[aviron-cycler/refresh](#)

Refreshes the cycler. Rescans all sub dirs.

[aviron-cycler/last-dir-name](#)

Returns the last sub dir name in the cycle or nil if there was not yet a call to next

[aviron-cycler/last-dir-timestamp](#)

Returns the timestamp the last sub dir was run or nil if there was no run yet.

[section](#)

[top](#)

aviron-limiter/clamd-activate-cpu-limit

```
(clamd-activate-cpu-limit limiter)
(clamd-activate-cpu-limit limiter limit)
```

Activates a new CPU limit on the clamd daemon .

Without passing a 'limit' computes the CPU limit based on the limiter's profile and the current date and time.

With a 'limit' activates this CPU limit on the clamd process.

```
;; [1] Activating explicit CPU limit
(do
  (load-module :aviron-limiter)
  (let [pid-file   (io/file "clamd-pid-file")
        limiter    (aviron-limiter/create-clamd-cpu-limiter pid-file)
        last-limit (aviron-limiter/clamd-last-seen-limit limiter)
        new-limit   60
        activated?  (aviron-limiter/clamd-activate-cpu-limit limiter new-limit)]
    (when activated?
      (println "Adjusted clamd CPU limit:" last-limit "->" new-limit))))

;; [2] Activating current limit from CPU profile
(do
  (load-module :aviron-limiter)
  (let [profile      (aviron-limiter/create-cpu-profile
                      "anyday"
                      [ "00:00-05:59 @ 100%"
                        "06:00-08:59 @ 50%"
                        "09:00-17:59 @ 0%"
                        "18:00-21:59 @ 50%"
                        "22:00-23:59 @ 100%" ])
        pid-file     (io/file "clamd-pid-file")
        limiter       (aviron-limiter/create-clamd-cpu-limiter pid-file (repeat 7 profile))
        last-limit    (aviron-limiter/clamd-last-seen-limit limiter)
        activated?    (aviron-limiter/clamd-activate-cpu-limit limiter)
        new-limit     (aviron-limiter/clamd-last-seen-limit limiter)]
    (when activated?
      (println "Adjusted clamd CPU limit:" last-limit "->" new-limit))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-clamd-cpu-limiter](#)

Creates a new CPU limiter to operate on clamd daemons.

[aviron-limiter/clamd-deactivate-cpu-limit](#)

Deactivates any CPU limitation on the clamd deamon.

[aviron-limiter/clamd-last-seen-limit](#)

Returns the last seen (activated) CPU limit on the clamd daemon.

[aviron-limiter/clamd-limit-for-timestamp](#)

Returns the limit for a given timestamp. If the timestamp is nil returns the limit for now.

[aviron-limiter/format-profiles-as-table-by-hour](#)

Returns the limiter's profile data as a string formatted table.

[section](#)

[top](#)

aviron-limiter/clamd-deactivate-cpu-limit

```
(clamd-deactivate-cpu-limit limiter)
```

Deactivates any CPU limitation on the clamd deamon.

```
(do
  (load-module :aviron-limiter)
  (let [pid-file (io/file "clamd-pid-file")
        limiter (aviron-limiter/create-clamd-cpu-limiter pid-file)]
    (aviron-limiter/clamd-deactivate-cpu-limit limiter)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-clamd-cpu-limiter](#)

Creates a new CPU limiter to operate on clamd daemons.

[aviron-limiter/clamd-activate-cpu-limit](#)

Activates a new CPU limit on the clamd daemon .

[aviron-limiter/clamd-last-seen-limit](#)

Returns the last seen (activated) CPU limit on the clamd daemon.

[aviron-limiter/clamd-limit-for-timestamp](#)

Returns the limit for a given timestamp. If the timestamp is nil returns the limit for now.

[aviron-limiter/format-profiles-as-table-by-hour](#)

Returns the limiter's profile data as a string formatted table.

[section](#)

[top](#)

aviron-limiter/clamd-last-seen-limit

```
(clamd-last-seen-limit limiter)
```

Returns the last seen (activated) CPU limit on the clamd daemon.

```
(do
  (load-module :aviron-limiter)
  (let [limiter (aviron-limiter/create-clamd-cpu-limiter)]
    (aviron-limiter/clamd-last-seen-limit limiter)))

(do
  (load-module :aviron-limiter)
  (let [pid-file (io/file "clamd-pid-file")
        limiter (aviron-limiter/create-clamd-cpu-limiter pid-file)
        last-limit (aviron-limiter/clamd-last-seen-limit limiter)
        new-limit 60
        activated? (aviron-limiter/clamd-activate-cpu-limit limiter new-limit)]
    (when activated?
      (println "Adjusted clamd CPU limit:" last-limit "->" new-limit))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-clamd-cpu-limiter](#)

Creates a new CPU limiter to operate on clamd daemons.

[aviron-limiter/clamd-activate-cpu-limit](#)

Activates a new CPU limit on the clamd daemon .

[aviron-limiter/clamd-deactivate-cpu-limit](#)

Deactivates any CPU limitation on the clamd deamon.

[aviron-limiter/clamd-limit-for-timestamp](#)

Returns the limit for a given timestamp. If the timestamp is nil returns the limit for now.

[aviron-limiter/format-profiles-as-table-by-hour](#)

Returns the limiter's profile data as a string formatted table.

[section](#)
[top](#)

aviron-limiter/clamd-limit-for-timestamp

```
(clamd-limit-for-timestamp limiter)
```

Returns the limit for a given timestamp. If the timestamp is `nil` returns the limit for *now*.

```
(do
  (load-module :aviron-limiter)
  (let [profile (aviron-limiter/create-cpu-profile
                 "anyday"
                 [ "00:00-05:59 @ 100%"
                   "06:00-08:59 @ 50%"
                   "09:00-17:59 @ 0%"
                   "18:00-21:59 @ 50%"
                   "22:00-23:59 @ 100%" ])
        pid-file (io/file "clamd-pid-file")
        limiter (aviron-limiter/create-clamd-cpu-limiter
                  pid-file
                  (repeat 7 profile))]
    (aviron-limiter/clamd-limit-for-timestamp
      limiter
      (time/local-date-time 2025 7 1 18 30 0))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-clamd-cpu-limiter](#)

Creates a new CPU limiter to operate on clamd daemons.

[aviron-limiter/clamd-activate-cpu-limit](#)

Activates a new CPU limit on the clamd daemon .

[aviron-limiter/clamd-deactivate-cpu-limit](#)

Deactivates any CPU limitation on the clamd deamon.

[aviron-limiter/clamd-last-seen-limit](#)

Returns the last seen (activated) CPU limit on the clamd daemon.

[aviron-limiter/format-profiles-as-table-by-hour](#)

Returns the limiter's profile data as a string formatted table.

[section](#)

[top](#)

aviron-limiter/compute-dynamic-cpu-limit

```
(compute-dynamic-cpu-limit limiter)
(compute-dynamic-cpu-limit limiter timestamp)
```

Compute the CPU limit based on dynamic CPU limiter.

Without an argument computes the CPU limit for the current date/time.

Otherwise computes the CPU limit for the given timestamp (type `time/local-date-time`).

```
;; [1] Compute limit for current time based on a limit compute
;;      function
(do
  (load-module :aviron-limiter)
  (defn limit-fn [ts] 100I) ;; Important: must return an integer!
  (let [limiter (aviron-limiter/create-dynamic-cpu-limiter limit-fn)]
    (aviron-limiter/compute-dynamic-cpu-limit limiter)))

;; [2] Compute limit for a specific timestamp based on a limit compute
;;      function
(do
  (load-module :aviron-limiter)
  (defn limit-fn [ts] 100I) ;; Important: must return an integer!
  (let [limiter (aviron-limiter/create-dynamic-cpu-limiter limit-fn)
        ts      (time/local-date-time 2025 7 26 10 30 0)]
    (aviron-limiter/compute-dynamic-cpu-limit limiter ts)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-dynamic-cpu-limiter](#)

Creates a new dynamic CPU limiter.

[section](#)

[top](#)

aviron-limiter/create-clamd-cpu-limiter

```
(create-clamd-cpu-limiter)
(create-clamd-cpu-limiter profiles)
(create-clamd-cpu-limiter limit-fn)
```

Creates a new CPU limiter to operate on clamd daemons.

Without an argument creates a limiter with a default profile (100% CPU usage).

Otherwise creates a limiter with a list of explicit profiles for each day in the week (Mon to Sun) or a limiter based on a limiter function that computes the CPU limit for a timestamp.

Note: Uses internally the dynamic CPU limiter to compute the new CPU limits for the clamd daemon.

```
;; [1] Limiter with default profile
(do
  (load-module :aviron-limiter)
  (aviron-limiter/create-clamd-cpu-limiter))

;; [2] Limiter with profiles for each day in the week (Mon to Sun)
(do
  (load-module :aviron-limiter)
  (let [weekday (aviron-limiter/create-cpu-profile
                  "weekday"
                  [ "00:00-05:59 @ 100%"
                    "06:00-21:59 @ 0%"
                    "22:00-23:59 @ 100%" ])
        weekend (aviron-limiter/create-cpu-profile
                 "weekend"
                 [ "00:00-05:59 @ 100%"
                   "06:00-21:59 @ 60%"
                   "22:00-23:59 @ 100%" ])]
    (aviron-limiter/create-clamd-cpu-limiter
      (io/file "clamd-pid-file")
      (concat (repeat 5 weekday)
              (repeat 2 weekend)))))

;; [3] Limiter based on a limiter function that computes the CPU limit for
;; a timestamp
(do
  (load-module :aviron-limiter)
  (defn limit-fn [ts] 100I)
  (aviron-limiter/create-clamd-cpu-limiter (io/file "clamd-pid-file") limit-fn))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/clamd-activate-cpu-limit](#)

Activates a new CPU limit on the clamd daemon .

[aviron-limiter/clamd-deactivate-cpu-limit](#)

Deactivates any CPU limitation on the clamd deamon.

[aviron-limiter/clamd-last-seen-limit](#)

Returns the last seen (activated) CPU limit on the clamd daemon.

[aviron-limiter/clamd-limit-for-timestamp](#)

Returns the limit for a given timestamp. If the timestamp is nil returns the limit for now.

[aviron-limiter/format-profiles-as-table-by-hour](#)

Returns the limiter's profile data as a string formatted table.

aviron-limiter/create-cpu-profile

```
(create-cpu-profile name data)
```

Create a named CPU profile.

```
(do
  (load-module :aviron-limiter)
  (aviron-limiter/create-cpu-profile
    "weekday"
    [ "00:00-05:59 @ 100%"
      "06:00-08:59 @ 50%"
      "09:00-17:59 @ 0%"
      "18:00-21:59 @ 50%"
      "22:00-23:59 @ 100%" ]))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/get-cpu-profile-entries](#)

Returns the entries java CPU profile.

[aviron-limiter/get-cpu-profile-limit](#)

Returns the CPU limit defined by the profile for a given hour and minute.

[section](#)

[top](#)

aviron-limiter/create-dynamic-cpu-limiter

```
(create-dynamic-cpu-limiter)
(create-dynamic-cpu-limiter profiles)
(create-dynamic-cpu-limiter limit-fn)
```

Creates a new dynamic CPU limiter.

Without an argument creates a limiter with a default profile (100% CPU usage).

Otherwise creates a limiter with a list of explicit profiles for each day in the week (Mon to Sun) or a limiter based on a limiter function that computes the CPU limit for a timestamp.

```
;; [1] Limiter with default profile
(do
  (load-module :aviron-limiter)
  (let [limiter (aviron-limiter/create-dynamic-cpu-limiter)
        ts      (time/local-date-time 2025 7 21 12 0 0)]
    (aviron-limiter/compute-dynamic-cpu-limit limiter ts)))

;; [2] Limiter with profiles for each day in the week (Mon to Sun)
(do
  (load-module :aviron-limiter)
  (let [weekday (aviron-limiter/create-cpu-profile
    "weekday"
    [ "00:00-05:59 @ 100%"
      "06:00-21:59 @ 0%"
      "22:00-23:59 @ 100%" ])
        weekend (aviron-limiter/create-cpu-profile
    "weekend"
```

```

        [ "00:00-05:59 @ 100%"
          "06:00-21:59 @ 60%"
          "22:00-23:59 @ 100%" ])

limiter (aviron-limiter/create-dynamic-cpu-limiter
         (concat (repeat 5 weekday)
                  (repeat 2 weekend)))

ts-mon   (time/local-date-time 2025 7 21 12 0 0)
ts-sat   (time/local-date-time 2025 7 26 12 0 0)]
(println "Monday: " (aviron-limiter/compute-dynamic-cpu-limit limiter ts-mon))
(println "Saturday:" (aviron-limiter/compute-dynamic-cpu-limit limiter ts-sat)))

;; [3] Limiter based on a limiter function that computes the CPU limit
;;      for a timestamp
(do
  (load-module :aviron-limiter)
  (defn limit-fn [ts] 100I) ;; Important: must return an integer!
  (let [limiter (aviron-limiter/create-dynamic-cpu-limiter limit-fn)
        ts      (time/local-date-time 2025 7 21 12 0 0)]
    (aviron-limiter/compute-dynamic-cpu-limit limiter ts)))

```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/compute-dynamic-cpu-limit](#)

Compute the CPU limit based on dynamic CPU limiter.

[section](#)

[top](#)

aviron-limiter/format-profiles-as-table-by-hour

```
(format-profiles-as-table-by-hour limiter)
```

Returns the limiter's profile data as a string formatted table.

The limiter may be a limiter obtained from:

- `aviron-limiter/create-dynamic-cpu-limiter`
- `aviron-limiter/create-clamd-cpu-limiter`

```

(do
  (load-module :aviron-limiter)
  (let [profile (aviron-limiter/create-cpu-profile
                 "anyday"
                 [ "00:00-05:59 @ 100%"
                   "06:00-08:59 @ 50%"
                   "09:00-17:59 @ 0%"
                   "18:00-21:59 @ 50%"
                   "22:00-23:59 @ 100%" ]])
        pid-file (io/file "clamd-pid-file")
        limiter (aviron-limiter/create-clamd-cpu-limiter
                  pid-file
                  (repeat 7 profile))]
    (println (aviron-limiter/format-profiles-as-table-by-hour limiter))))

```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-clamd-cpu-limiter](#)

Creates a new CPU limiter to operate on clamd daemons.

[aviron-limiter/clamd-activate-cpu-limit](#)

Activates a new CPU limit on the clamd daemon .

[aviron-limiter/clamd-deactivate-cpu-limit](#)

Deactivates any CPU limitation on the clamd deamon.

[aviron-limiter/clamd-last-seen-limit](#)

Returns the last seen (activated) CPU limit on the clamd daemon.

[aviron-limiter/clamd-limit-for-timestamp](#)

Returns the limit for a given timestamp. If the timestamp is nil returns the limit for now.

[section](#)

[top](#)

aviron-limiter/get-cpu-profile-entries

```
(get-cpu-profile-entries profile)
```

Returns the entries java CPU profile.

```
(do
  (load-module :aviron-limiter)
  (let [profile (aviron-limiter/create-cpu-profile
                  "weekday"
                  [ "00:00-05:59 @ 100%"
                    "06:00-08:59 @ 50%"
                    "09:00-17:59 @ 0%"
                    "18:00-21:59 @ 50%"
                    "22:00-23:59 @ 100%" ]])]
    (aviron-limiter/get-cpu-profile-entries profile)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-cpu-profile](#)

Create a named CPU profile.

[aviron-limiter/get-cpu-profile-limit](#)

Returns the CPU limit defined by the profile for a given hour and minute.

[section](#)

[top](#)

aviron-limiter/get-cpu-profile-limit

```
(get-cpu-profile-limit profile hour minute)
```

Returns the CPU limit defined by the profile for a given hour and minute.

```
(do
  (load-module :aviron-limiter)
  (let [profile (aviron-limiter/create-cpu-profile
                  "weekday"
                  [ "00:00-05:59 @ 100%"
                    "06:00-08:59 @ 50%"
```

```
"09:00-17:59 @ 0%"
"18:00-21:59 @ 50%"
"22:00-23:59 @ 100%" ]]]
(aviron-limiter/get-cpu-profile-limit profile 18 45)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron-limiter/create-cpu-profile](#)

Create a named CPU profile.

[aviron-limiter/get-cpu-profile-entries](#)

Returns the entries java CPU profile.

[section](#)

[top](#)

aviron-queue/capacity

```
(capacity queue)
```

Returns the capacity of the file queue

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/capacity q)))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/push](#)

Push a new file to the tail of the queue

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/clear

```
(clear queue)
```

Clears the file queue

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/clear q)))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/push](#)

Push a new file to the tail of the queue

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/create

```
(create)
(create capacity)
```

Create a new file queue with either the default capacity or an explicit capacity.

The file queue is buffering file watching events. It asynchronously decouples the event producing file watcher from the event consuming AV scanner client.

The queue never blocks and never grows beyond limits to protect the system! Therefore the queue is non blocking and has a fix capacity. As a consequence it must discard old events if overrun.

File watchers (like the Java `WatchService` or the `fswatch` tool) have the same behavior. If they get overrun with file change events they discard events and signal it by sending an 'OVERFLOW' event to their clients.

```
(do
  (load-module :aviron-queue)
  (aviron-queue/create))

(do
  (load-module :aviron-queue)
  (aviron-queue/create 5000))
```

SEE ALSO

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/push](#)

Push a new file to the tail of the queue

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/empty?

(empty? queue)

Returns `true` if the file queue is empty else `false`

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/empty? q)))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/push](#)

Push a new file to the tail of the queue

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/overflow-count

(overflow-count queue)

Returns the overflow event count.

Whenever pushing a file to a full queue the queue's head element is discarded in favor of pushing the new file. The overflow count is incremented in this case.

The queue never blocks and never grows beyond limits to protect the system! Therefore the queue is non blocking and has a fix capacity. As a consequence it must discard old events if overrun.

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create 5)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (aviron-queue/push q (io/file "/data/d.txt"))
    (aviron-queue/push q (io/file "/data/e.txt"))
    (aviron-queue/push q (io/file "/data/f.txt"))
    ;; overflow event (the head element "/data/a.txt" is discarded)
    (aviron-queue/push q (io/file "/data/g.txt"))
    (println (aviron-queue/overflow-count q))))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/overflow-reset

```
(overflow-reset queue)
```

Resets the overflow count

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create 5)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (aviron-queue/push q (io/file "/data/d.txt"))
    (aviron-queue/push q (io/file "/data/e.txt"))
    (aviron-queue/push q (io/file "/data/f.txt"))
    ;; overflow event (the head element "/data/a.txt" is discarded)
    (aviron-queue/push q (io/file "/data/g.txt"))
    (println (aviron-queue/overflow-count q))
    (println (aviron-queue/overflow-reset q))
    (println (aviron-queue/overflow-count q))))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)[top](#)

aviron-queue/pop

```
(pop queue)
(pop queue n)
(pop queue n existing-files-only)
```

Pops the next or n next files from the head of the queue.

If 'existing-files-only' is `true` discards any queue head files that are not existing.

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (println (aviron-queue/pop q))
    (println (aviron-queue/pop q))
    (println (aviron-queue/pop q))))

(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (aviron-queue/push q (io/file "/data/c.txt"))
    (println (aviron-queue/pop q 5))))

(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (aviron-queue/push q (io/file "/data/c.txt"))
    (println (aviron-queue/pop q 5 true))))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/push](#)

Push a new file to the tail of the queue

[section](#)

[top](#)

aviron-queue/push

```
(push queue file)
```

Push a new file to the tail of the queue

Whenever pushing a file to a full queue the queue's head element is discarded in favor of pushing the new file. The overflow count is incremented in this case.

The queue never blocks and never grows beyond limits to protect the system! Therefore the queue is non blocking and has a fix capacity. As a consequence it must discard old events if overrun.

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (println (aviron-queue/pop q))
    (println (aviron-queue/pop q))))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/remove

```
(remove queue file)
```

Removes a file form the queue.

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/push q (io/file "/data/a.txt"))
    (aviron-queue/push q (io/file "/data/b.txt"))
    (aviron-queue/push q (io/file "/data/c.txt"))
    (aviron-queue/remove q (io/file "/data/b.txt")))))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/size](#)

Returns the size of the file queue

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/push](#)

Push a new file to the tail of the queue

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron-queue/size

```
(size queue)
```

Returns the size of the file queue

```
(do
  (load-module :aviron-queue)
  (let [q (aviron-queue/create)]
    (aviron-queue/size q)))
```

SEE ALSO

[aviron-queue/create](#)

Create a new file queue with either the default capacity or an explicit capacity.

[aviron-queue/capacity](#)

Returns the capacity of the file queue

[aviron-queue/empty?](#)

Returns true if the file queue is empty else false

[aviron-queue/clear](#)

Clears the file queue

[aviron-queue/remove](#)

Removes a file form the queue.

[aviron-queue/push](#)

Push a new file to the tail of the queue

[aviron-queue/pop](#)

Pops the next or n next files from the head of the queue.

[section](#)

[top](#)

aviron/clamav-version

```
(clamav-version client)
```

Return the ClamAV version

```
(do
  (load-module :aviron)
  (aviron/clamav-version (aviron/create-client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ping](#)

Sends a "PING" command to the ClamAV server.

[aviron/stats](#)

Returns the formatted statistics about the scan queue, contents of scan queue, and memory usage.

[aviron/reload-virus-databases](#)

Reload the virus databases.

[aviron/shutdown-server](#)

Shutdown the ClamAV server and perform a clean exit.

[section](#)

[top](#)

aviron/clamd-cpu-limit

```
(clamd-cpu-limit clamd-pid limit)
```

Activates a CPU limit [1..LIMIT] percent on a clamd daemon process.

The max value of LIMIT depends on the number of logical proprocessors:

- on a 8 core **MacBook Air** LIMIT is 800%
- on a **Intel** single core with 2 hyperthreads LIMIT is 200%

```
(do
  (load-module :aviron)
  (aviron/clamd-cpu-limit (aviron/clamd-pid) 50))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/clamd-cpu-limit-off](#)

Deactivates a CPU limit on a clamd daemon process.

[aviron/clamd-pid](#)

Returns the clamd PID or nil if the clamd daemon is not running.

[aviron/cpus](#)

Returns the number of available processors or the number of hyperthreads if the CPU supports hyperthreads.

[section](#)

[top](#)

aviron/clamd-cpu-limit-off

```
(clamd-cpu-limit-off clamd-pid)
```

Deactivates a CPU limit on a clamd daemon process.

```
(do
  (load-module :aviron)
  (aviron/clamd-cpu-limit-off (aviron/clamd-pid)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/clamd-cpu-limit](#)

Activates a CPU limit [1..LIMIT] percent on a clamd daemon process.

[aviron/clamd-pid](#)

Returns the clamd PID or nil if the clamd daemon is not running.

[aviron/cpus](#)

Returns the number of available processors or the number of hyperthreads if the CPU supports hyperthreads.

[section](#)

[top](#)

aviron/clamd-pid

```
(clamd-pid)
(clamd-pid pid-file)
```

Returns the clamd PID or `nil` if the clamd daemon is not running.

Without 'pid-file' argument runs a `pgrep clamd` shell command to get the pid. With a 'pid-file' loads the pid from the pid file.

```
(do
  (load-module :aviron)
  (aviron/clamd-pid))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/clamd-cpu-limit](#)

Activates a CPU limit [1..LIMIT] percent on a clamd daemon process.

[aviron/clamd-cpu-limit-off](#)

Deactivates a CPU limit on a clamd daemon process.

[aviron/cpus](#)

Returns the number of available processors or the number of hyperthreads if the CPU supports hyperthreads.

[section](#)

[top](#)

aviron/cpus

(cpus)

Returns the number of available processors or the number of hyperthreads if the CPU supports hyperthreads.

```
(do
  (load-module :aviron)
  (aviron/cpus))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/clamd-pid](#)

Returns the clamd PID or nil if the clamd daemon is not running.

[aviron/clamd-cpu-limit](#)

Activates a CPU limit [1..LIMIT] percent on a clamd daemon process.

[aviron/clamd-cpu-limit-off](#)

Deactivates a CPU limit on a clamd daemon process.

[section](#)

[top](#)

aviron/create-client

(create-client & options)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, or getting the scanning stats.

Options:

:server-hostname s	The ClamAV server hostname. Defaults to "localhost"
:server-port n	The ClamAV server port. Defaults to 3310
:server-file-separator k	The ClamAV server file separator, one of { :UNIX , :WINDOWS , :JVM_PLATFORM }. Defaults to :JVM_PLATFORM
:connection-timeout n	The connection timeout in milliseconds, 0 means indefinite. Defaults to 3'000ms
:read-timeout n	The read timeout in milliseconds, 0 means indefinite. Defaults to 20'000ms
:quarantine-dir f	A quarantine directory. If the quarantine action is :MOVE or :COPY infected files will be move/copied to the quarantine directory.
:quarantine-action e	A quarantine file action for infected, one of { :NONE , :MOVE , :COPY }. files. Defaults to :NONE

```
(do
  (load-module :aviron)
```

```
(let [client (aviron/create-client :server-hostname "localhost"
                                   :server-file-separator :UNIX)]

  (println (aviron/reachable? client))

  (println (aviron/clamav-version client))

  (let [result (aviron/scan-path client "/data/summary.docx")]
    (when (aviron/virus? result)
      (println (aviron/viruses result))))))
```

SEE ALSO

[aviron/scan-stream](#)

Scans a file's data passed in the 'is' stream. Uses a default chunk size of 2048 bytes.

[aviron/scan-path](#)

Scans a single file or directory (recursively). Stops by default after the first file with a detected virus.

[aviron/scan-parallel](#)

Scans a single file or directory (recursively) using multiple threads.

[aviron/shutdown-server](#)

Shutdown the ClamAV server and perform a clean exit.

[aviron/ping](#)

Sends a "PING" command to the ClamAV server.

[aviron/clamav-version](#)

Return the ClamAV version

[aviron/stats](#)

Returns the formatted statistics about the scan queue, contents of scan queue, and memory usage.

[aviron/reachable?](#)

Tests if the ClamAV server is reachable. Uses a default timeout of 3'000ms.

[aviron/wait-for-operational-clamd](#)

Wait for Clamd to get operational.

[aviron/reload-virus-databases](#)

Reload the virus databases.

[aviron/shutdown-server](#)

Shutdown the ClamAV server and perform a clean exit.

[aviron/last-command-run-details](#)

Returns the raw command string and the server's result for the last command sent to the ClamAV server.

[section](#)

[top](#)

aviron/last-command-run-details

```
(last-command-run-details client)
```

Returns the raw command string and the server's result for the last command sent to the ClamAV server.

This function is provided for debugging purposes

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/clamav-version client)
    (println (aviron/last-command-run-details client))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[section](#)[top](#)

aviron/list-quarantine-files

```
(list-quarantine-files client)
```

Returns a list of the quarantine files.

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/list-quarantine-files client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/quarantine-active?](#)

Returns true if the quarantine is active else false

[aviron/remove-quarantine-file](#)

Removes a quarantine file.

[aviron/remove-all-quarantine-files](#)

Removes all quarantine files.

[section](#)[top](#)

aviron/ok?

```
(ok? scan-result)
```

Returns `true` if the scan result does not contain any detected viruses else `false`

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/ok? (aviron/scan-path "/data/document.pdf"))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/virus?](#)

Returns true if the scan result does contain any detected viruses else false

[aviron/viruses](#)

Returns a map of the detected viruses.

[section](#)[top](#)

aviron/ping

```
(ping client)
```

Sends a "PING" command to the ClamAV server.

Returns `true` if the server answers with a "PONG" else `false`

```
(do
  (load-module :aviron)
  (aviron/ping (aviron/create-client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/clamav-version](#)

Return the ClamAV version

[aviron/stats](#)

Returns the formatted statistics about the scan queue, contents of scan queue, and memory usage.

[aviron/reload-virus-databases](#)

Reload the virus databases.

[aviron/shutdown-server](#)

Shutdown the ClamAV server and perform a clean exit.

[section](#)

[top](#)

aviron/print-config

```
(print-config client)
(print-config client ps)
```

Print the client configuration in human readable form to a `:java.io.PrintStream`

If the passed stream is `nil` prints to stdout

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/print-config client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[section](#)

[top](#)

aviron/quarantine-active?

```
(quarantine-active? client)
```

Returns `true` if the quarantine is active else `false`

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/quarantine-active? client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/list-quarantine-files](#)

Returns a list of the quarantine files.

[aviron/remove-quarantine-file](#)

Removes a quarantine file.

[aviron/remove-all-quarantine-files](#)

Removes all quarantine files.

[section](#)

[top](#)

aviron/reachable?

```
(reachable? client)
(reachable? client timeout)
```

Tests if the ClamAV server is reachable. Uses a default timeout of 3'000ms.

Returns `true` if the server is reachable else `false` .

```
(do
  (load-module :aviron)
  (aviron/reachable? (aviron/create-client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[section](#)

[top](#)

aviron/reload-virus-databases

```
(reload-virus-databases client)
```

Reload the virus databases.

```
(do
  (load-module :aviron)
  (aviron/reload-virus-databases (aviron/create-client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ping](#)

Sends a "PING" command to the ClamAV server.

[aviron/clamav-version](#)

Return the ClamAV version

[aviron/stats](#)

Returns the formatted statistics about the scan queue, contents of scan queue, and memory usage.

[aviron/shutdown-server](#)

Shutdown the ClamAV server and perform a clean exit.

[section](#)

[top](#)

aviron/remove-all-quarantine-files

```
(remove-all-quarantine-files client)
```

Removes all quarantine files.

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/remove-all-quarantine-files client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/quarantine-active?](#)

Returns true if the quarantine is active else false

[aviron/list-quarantine-files](#)

Returns a list of the quarantine files.

[aviron/remove-quarantine-file](#)

Removes a quarantine file.

[section](#)

[top](#)

aviron/remove-quarantine-file

```
(remove-quarantine-file client f)
```

Removes a quarantine file.

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (if-let [f (first (aviron/list-quarantine-files client))]
      (aviron/remove-quarantine-file client f))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/quarantine-active?](#)

Returns true if the quarantine is active else false

[aviron/list-quarantine-files](#)

Returns a list of the quarantine files.

[aviron/remove-all-quarantine-files](#)

Removes all quarantine files.

[section](#)

[top](#)

aviron/scan-parallel

(scan-parallel client path)

Scans a single file or directory (recursively) using multiple threads.

Returns the scan result with the detected virus info

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/ok? (aviron/scan-parallel "/data")))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ok?](#)

Returns true if the scan result does not contain any detected viruses else false

[aviron/virus?](#)

Returns true if the scan result does contain any detected viruses else false

[aviron/viruses](#)

Returns a map of the detected viruses.

[section](#)

[top](#)

aviron/scan-path

(scan-path client path)

(scan-path client path continue)

Scans a single file or directory (recursively). Stops by default after the first file with a detected virus.

If 'continue' is `true` continues scanning upon detecting a virus in a file else stops after the first file with a detected virus.

Returns the scan result with the detected virus info

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/ok? (aviron/scan-path "/data/document.pdf")))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ok?](#)

Returns true if the scan result does not contain any detected viruses else false

[aviron/virus?](#)

Returns true if the scan result does contain any detected viruses else false

[aviron/viruses](#)

Returns a map of the detected viruses.

[section](#)[top](#)

aviron/scan-stream

```
(scan-stream is)
(scan-stream is chunk-size)
```

Scans a file's data passed in the 'is' stream. Uses a default chunk size of 2048 bytes.

Returns the scan result with the detected virus info

Note: The input stream must be closed by the caller!

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (try-with [is (io/file-in-stream "/data/document.pdf")]
      (aviron/ok? (aviron/scan-stream is))))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ok?](#)

Returns true if the scan result does not contain any detected viruses else false

[aviron/virus?](#)

Returns true if the scan result does contain any detected viruses else false

[aviron/viruses](#)

Returns a map of the detected viruses.

[section](#)[top](#)

aviron/shutdown-server

```
(shutdown-server client)
```

Shutdown the ClamAV server and perform a clean exit.

```
(do
  (load-module :aviron)
  (aviron/shutdown-server (aviron/create-client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ping](#)

Sends a "PING" command to the ClamAV server.

[aviron/clamav-version](#)

Return the ClamAV version

[aviron/stats](#)

Returns the formatted statistics about the scan queue, contents of scan queue, and memory usage.

[aviron/reload-virus-databases](#)

Reload the virus databases.

[section](#)

[top](#)

aviron/stats

```
(stats client)
```

Returns the formatted statistics about the scan queue, contents of scan queue, and memory usage.

```
(do
  (load-module :aviron)
  (aviron/stats (aviron/create-client)))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ping](#)

Sends a "PING" command to the ClamAV server.

[aviron/clamav-version](#)

Return the ClamAV version

[aviron/reload-virus-databases](#)

Reload the virus databases.

[aviron/shutdown-server](#)

Shutdown the ClamAV server and perform a clean exit.

[section](#)

[top](#)

aviron/version

```
(version)
```

Returns the version of the Aviron Java library.

```
(do
  (load-module :aviron)
  (aviron/version))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

aviron/virus?

(virus? scan-result)

Returns `true` if the scan result does contain any detected viruses else `false`

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (aviron/virus? (aviron/scan-path "/data/document.pdf"))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ok?](#)

Returns true if the scan result does not contain any detected viruses else false

[aviron/viruses](#)

Returns a map of the detected viruses.

aviron/viruses

(viruses scan-result)

Returns a map of the detected viruses.

The map's keys hold the file detected and the values contain a list of the virus types detected in the file.

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (println (aviron/viruses (aviron/scan-path "/data/document.pdf")))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[aviron/ok?](#)

Returns true if the scan result does not contain any detected viruses else false

[aviron/virus?](#)

Returns true if the scan result does contain any detected viruses else false

aviron/wait-for-operational-clamd

```
(wait-for-operational-clamd max-wait-time time-unit)
(wait-for-operational-clamd max-wait-time time-unit listener)
```

Wait for Clamd to get operational.

Returns `true` if the associated clamd daemon is operational else `false`.

Checks first if the clamd daemon process is running followed by ping command to verify that clamd is operational and ready to accept commands.

Clamd has pretty long startup time due the the time expensive virus database loading.

`time-unit` is one of `{:milliseconds, :seconds, :minutes, :hours}`

```
(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (println (aviron/wait-for-operational-clamd client 1 :minutes))))

(do
  (load-module :aviron)
  (let [client (aviron/create-client)]
    (println (aviron/wait-for-operational-clamd
              client
              1
              :minutes
              #(println "Clamd:" (. % :getStatus))))))
```

SEE ALSO

[aviron/create-client](#)

The ClamAV client provides access to the ClamAV daemon (clamd) functions like file scanning, updating the daemon's ClamAV virus databases, ...

[section](#)

[top](#)

await

`(await agents)`

Blocks the current thread (indefinitely) until all actions dispatched thus far (from this thread or agent) to the agents have occurred.

```
(do
  (def x1 (agent 100))
  (def x2 (agent {}))
  (send-off x1 + 5)
  (send-off x2 (fn [state]
                 (sleep 100)
                 (assoc state :done true))))
;; blocks till the agent actions are finished
(await x1 x2)
=> true
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[await-for](#)

Blocks the current thread until all actions dispatched thus far (from this thread or agent) to the agents have occurred, or the timeout ...

[section](#)

[top](#)

await-for

```
(await-for timeout-ms agents)
```

Blocks the current thread until all actions dispatched thus far (from this thread or agent) to the agents have occurred, or the timeout (in milliseconds) has elapsed. Returns logical false if returning due to timeout, logical true otherwise.

```
(do
  (def x1 (agent 100))
  (def x2 (agent {}))
  (send-off x1 + 5)
  (send-off x2 (fn [state]
                 (sleep 100)
                 (assoc state :done true))))
;; blocks till the agent actions are finished
(await-for 500 x1 x2))
=> true
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[await](#)

Blocks the current thread (indefinitely) until all actions dispatched thus far (from this thread or agent) to the agents have occurred.

[section](#)[top](#)

await-termination-agents

```
(shutdown-agents)
```

Blocks until all actions have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

```
(do
  (def x1 (agent 100))
  (def x2 (agent 100))
  (shutdown-agents)
  (await-termination-agents 1000))
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)[top](#)

await-termination-agents?

```
(await-termination-agents?)
```

Returns true if all tasks have been completed following agent shut down

```
(do
  (def x1 (agent 100))
  (def x2 (agent 100))
  (shutdown-agents)
  (await-termination-agents 1000)
  (sleep 300)
  (await-termination-agents?))
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)[top](#)

bases

```
(bases class)
```

Returns the immediate superclass and interfaces of class, if any.

```
(bases :java.util.ArrayList)
=> (:java.util.AbstractList :java.util.List :java.util.RandomAccess :java.lang.Cloneable :java.io.Serializable)
```

[section](#)[top](#)

benchmark/benchmark

```
(benchmark expr warmup-iterations iterations & options)
```

Benchmarks the given expression.

Note: All macros in the expression are expanded before running the benchmark phases.

Runs the benchmark in 4 phases:

1. Run the expression in a warm-up phase to allow the JIT compiler to do optimizations
2. Run the garbage collector to isolate timings from GC state prior to testing
3. Runs the expression benchmark
4. Analyzes and prints the benchmark statistics

Options:

:chart b	If true generates a chart and saves it to 'benchmark.png'. Defaults to false.
:steps n	the number of steps for the quantization, defaults to 100
:median b	show the median value in the chart {true/false}, defaults to false
:outlier b	show the outlier range in the chart {true/false}, defaults to false
:gc n	the number of GC runs

```
(do
  (load-module :benchmark ['benchmark :as 'b])

  (b/benchmark (+ 1 2) 120000 10000)

  (b/benchmark (+ 1 2) 120000 10000 :chart true :median true))
```

```
(b/benchmark (+ 1 2) 120000 10000 :chart true :outlier true)
```

```
(b/benchmark (+ 1 2) 120000 10000 :chart true :steps 100))
```

[section](#)[top](#)

bigint

```
(bigint x)
```

Converts to big integer.

```
(bigint 2000)
```

```
=> 2000N
```

```
(bigint 34897.65)
```

```
=> 34897N
```

```
(bigint 34897.65F)
```

```
=> 34897N
```

```
(bigint "56760000000000")
```

```
=> 56760000000000N
```

```
(bigint nil)
```

```
=> 0N
```

[section](#)[top](#)

binding

```
(binding [bindings*] exprs*)
```

Evaluates the expressions and binds the values to dynamic (thread-local) symbols

```
(do
  (binding [x 100]
    (println x)
    (binding [x 200]
      (println x))
    (println x)))
```

```
100
```

```
200
```

```
100
```

```
=> nil
```

;; binding-introduced bindings are thread-locally mutable:

```
(binding [x 1]
  (set! x 2)
```

```
  x)
```

```
=> 2
```



```
;; binding can use qualified names :
(binding [user/x 1]
  user/x)
=> 1
```

SEE ALSO

[def-dynamic](#)

Creates a dynamic variable that starts off as a global variable and can be bound with 'binding' to a new value on the local thread.

[let](#)

Evaluates the expressions and binds the values to symbols in the new local context.

[section](#)

[top](#)

boolean

```
(boolean x)
```

Converts to boolean. Everything except 'false' and 'nil' is true in boolean context.

```
(boolean false)
=> false
```

```
(boolean true)
=> true
```

```
(boolean nil)
=> false
```

```
(boolean 100)
=> true
```

[section](#)

[top](#)

boolean?

```
(boolean? n)
```

Returns true if n is a boolean

```
(boolean? true)
=> true
```

```
(boolean? false)
=> true
```

```
(boolean? nil)
=> false
```

```
(boolean? 0)
=> false
```

bound?

```
(bound? s)
```

Returns true if the symbol is bound to a value else false

```
(bound? 'test)
=> false
```

```
(let [test 100]
  (bound? 'test))
=> true
```

```
(do
  (def a 100)
  (bound? 'a))
=> true
```

SEE ALSO

[let](#)

Evaluates the expressions and binds the values to symbols in the new local context.

[def](#)

Creates a global variable.

[defonce](#)

Creates a global variable that can not be overwritten

butlast

```
(butlast coll)
```

Returns a collection with all but the last list element

```
(butlast nil)
=> nil
```

```
(butlast [])
=> []
```

```
(butlast [1])
=> []
```

```
(butlast [1 2 3])
=> [1 2]
```

```
(butlast '())
=> ()
```

```
(butlast '(1))
=> ()
```

```
(butlast '(1 2 3))  
=> (1 2)
```

```
(butlast "1234")  
=> (#\1 #\2 #\3)
```

SEE ALSO

[str/butlast](#)

Returns a possibly empty string of the characters without the last.

[section](#)

[top](#)

byte-order

```
(byte-order)
```

Returns the CPU's byte order.

```
(byte-order)  
=> :big-endian
```

[section](#)

[top](#)

bytebuf

```
(bytebuf x)
```

Converts x to bytebuf. x can be a bytebuf, a list/vector of longs, a string

```
(bytebuf [0 1 2])  
=> [0 1 2]
```

```
(bytebuf '(0 1 2))  
=> [0 1 2]
```

```
(bytebuf "abc")  
=> [97 98 99]
```

SEE ALSO

[io/bytebuf-out-stream](#)

Returns a new java.io.ByteArrayOutputStream.

[section](#)

[top](#)

bytebuf-allocate

```
(bytebuf-allocate length)  
(bytebuf-allocate length init-val)
```

The length can be specified as a number like `20000` or a number with a unit like `:20KB` or `:20MB`

$$\Rightarrow [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$$

```
=> [85 85 85 85 85 85 85 85 85 85 85 85 85 85 85 85 85 85 85 85 85]
```

[illegible][illegible]

[section](#) [top](#)

bytebuf-allocate-random

Allocates a new bytearray. The buffer will be preset with random bytes.

The length can be specified as a number like `20000` or a number with a unit like `:20KB` or `:20MB`

```
=> [174 193 76 124 128 203 84 216 253 75 65 104 226 228 171 172 25 209 224 214]
```

```
=> [185 238 189 221 6 135 146 239 5 189 58 229 40 127 239 9 106 135 188 235 167 75 12 56 198 206 19 165 65 251
67 35 185 207 138 164 247 94 182 67 22 44 219 107 92 1 14 80 174 245 65 79 49 12 198 213 26 44 187 154 144 105
198 122 141 56 7 150 83 17 94 204 134 44 84 249 171 6 112 104 46 158 231 85 176 61 138 165 127 208 62 74 75 237
40 207 125 191 103 133 ...]
```

[section](#) [top](#)

bytebuf-byte-order

Returns the bytearray's byte order.

```
=> :big-endian
```

bytebuf-byte-order!
Sets the bytebuf's byte order.

[section](#) [top](#)

bytebuf-byte-order!

```
(bytebuf-byte-order! buf endian)
```

Sets the bytebuf's byte order.

```
(-> (bytebuf-allocate 10)
    (bytebuf-byte-order! :big-endian)
    (bytebuf-byte-order))
=> :big-endian

(-> (bytebuf-allocate 10)
    (bytebuf-byte-order! :little-endian)
    (bytebuf-byte-order))
=> :little-endian
```

SEE ALSO

[bytebuf-byte-order](#)

Returns the bytebuf's byte order.

[section](#)

[top](#)

bytebuf-capacity

```
(bytebuf-capacity buf)
```

Returns the capacity of a bytebuf.

```
(bytebuf-capacity (bytebuf-allocate 100))
=> 100
```

SEE ALSO

[bytebuf-remaining](#)

Returns the number of bytes between the current position and the limit.

[bytebuf-limit](#)

Returns the limit of a bytebuf.

[bytebuf-pos](#)

Returns the buffer's current position.

[bytebuf-ensure-free-capacity!](#)

Ensure that the bytebuf has a free capacity. Returns the widened bytebuf.

[bytebuf-limit!](#)

Set a new limit for the buffer. The new limit must not be larger than the capacity.

[section](#)

[top](#)

bytebuf-ensure-free-capacity!

```
(bytebuf-ensure-capacity! buf capacity)
```

Ensure that the bytearray has a free capacity. Returns the widened bytearray.

[illegible]

SEE ALSO

bytebuf-remaining

Returns the number of bytes between the current position and the limit.

bytebuf-capacity

Returns the capacity of a bytearray.

bytebuf-limit

Returns the limit of a bytearray.

bytebuf-pos

Returns the buffer's current position.

bytebuf-limit!

Set a new limit for the buffer. The new limit must not be larger than the capacity.

[section](#) [top](#)

bytebuf-from-string

```
(bytebuf-from-string s)
(bytebuf-from-string s encoding)
(bytebuf-from-string s encoding buf-length fillbyte)
```

Converts a string to a bytearray using an optional encoding. The encoding defaults to :UTF-8

```
(bytebuf-from-string "abcdef")  
=> [97 98 99 100 101 102]  
  
(bytebuf-from-string "abcdef" :UTF-8)  
=> [97 98 99 100 101 102]  
  
(bytebuf-from-string "abcdef" :UTF-8  
=> [97 98 99 100 101 102 0 0 0 0 0 0
```

SEE ALSO

bytebuf-to-string

Converts a bytearray to a string using an optional encoding. The encoding defaults to :UTF-8

[section](#) [top](#)

bytebuf-get-byte

```
(bytebuf-get-byte buf)
(bytebuf-get-byte buf pos)
```

Reads a byte from the buffer. Without a pos reads from the current position and increments the position by one. With a position reads the byte from that position.

```
(-> (bytebuf-allocate 4)
    (bytebuf-put-byte! 1)
    (bytebuf-put-byte! 2)
    (bytebuf-get-byte 0))
=> 11
```

[section](#)[top](#)

bytebuf-get-double

```
(bytebuf-get-double buf)
(bytebuf-get-double buf pos)
```

Reads a double from the buffer. Without a pos reads from the current position and increments the position by eight. With a position reads the double from that position.

```
(-> (bytebuf-allocate 16)
    (bytebuf-put-double! 20.0)
    (bytebuf-put-double! 40.0)
    (bytebuf-get-double 0))
=> 20.0
```

[section](#)[top](#)

bytebuf-get-float

```
(bytebuf-get-float buf)
(bytebuf-get-float buf pos)
```

Reads a float from the buffer. Without a pos reads from the current position and increments the position by four. With a position reads the float from that position.

```
(-> (bytebuf-allocate 16)
    (bytebuf-put-float! 20.0)
    (bytebuf-put-float! 40.0)
    (bytebuf-get-float 0))
=> 20.0
```

[section](#)[top](#)

bytebuf-get-int

```
(bytebuf-get-int buf)
(bytebuf-get-int buf pos)
```

Reads an integer from the buffer. Without a pos reads from the current position and increments the position by four. With a position reads the integer from that position.

```
(-> (bytebuf-allocate 8)
    (bytebuf-put-int! 11)
    (bytebuf-put-int! 21)
    (bytebuf-get-int 0))
=> 11
```

bytebuf-get-long

```
(bytebuf-get-long buf)
(bytebuf-get-long buf pos)
```

Reads a long from the buffer. Without a pos reads from the current position and increments the position by eight. With a position reads the long from that position.

```
(> (bytebuf-allocate 16)
   (bytebuf-put-long! 20)
   (bytebuf-put-long! 40)
   (bytebuf-get-long 0))
=> 20
```

bytebuf-index-of

```
(bytebuf-index-of buf pattern)
(bytebuf-index-of buf pattern from-index)
(bytebuf-index-of buf pattern from-index to-index)
```

Returns the index within a byte buf of the first occurrence of the specified byte pattern.

The search is based on the Knuth-Morris-Pratt (KMP) pattern matching algorithm.

The KMP algorithm is an efficient method for finding the occurrence of a substring (a pattern) within a larger string (or in this case, a sequence of bytes)

```
(bytebuf-index-of (bytebuf [1 2 3 4 5]) (bytebuf [3 4]))
=> 2
```

```
(bytebuf-index-of (bytebuf [1 2 3 4 5 3 4]) (bytebuf [3 4]) 4)
=> 5
```

SEE ALSO

[bytebuf](#)

Converts x to bytebuf. x can be a bytebuf, a list/vector of longs, a string

bytebuf-limit

```
(bytebuf-limit buf)
```

Returns the limit of a bytebuf.

```
(bytebuf-limit (bytebuf-allocate 100))
=> 100
```


SEE ALSO

[bytebuf-remaining](#)

Returns the number of bytes between the current position and the limit.

[bytebuf-capacity](#)

Returns the capacity of a bytebuf.

[bytebuf-pos](#)

Returns the buffer's current position.

[bytebuf-ensure-free-capacity!](#)

Ensure that the bytebuf has a free capacity. Returns the widened bytebuf.

[bytebuf-limit!](#)

Set a new limit for the buffer. The new limit must not be larger than the capacity.

[section](#)

[top](#)

bytebuf-limit!

```
(bytebuf-limit! buf new-limit)
```

Set a new limit for the buffer. The new limit must not be larger than the capacity.

Returns the new limit of a bytebuf.

```
(bytebuf-limit! (bytebuf-allocate 100) 50)  
=> 50
```

SEE ALSO

[bytebuf-remaining](#)

Returns the number of bytes between the current position and the limit.

[bytebuf-capacity](#)

Returns the capacity of a bytebuf.

[bytebuf-limit](#)

Returns the limit of a bytebuf.

[bytebuf-pos](#)

Returns the buffer's current position.

[bytebuf-ensure-free-capacity!](#)

Ensure that the bytebuf has a free capacity. Returns the widened bytebuf.

[section](#)

[top](#)

bytebuf-merge

```
(bytebuf-merge buffers)
```

Merges bytebufs.

```
(bytebuf-merge (bytebuf [1 2]) (bytebuf [3 4]))  
=> [1 2 3 4]
```

SEE ALSO

bytebuf

Converts x to bytebuf. x can be a bytebuf, a list/vector of longs, a string

[section](#)

[top](#)

bytebuf-pos

(bytebuf-pos buf)

Returns the buffer's current position.

```
(bytebuf-pos (bytebuf-allocate 10))  
=> 0
```

SEE ALSO

[bytebuf-capacity](#)

Returns the capacity of a bytebuf.

[bytebuf-remaining](#)

Returns the number of bytes between the current position and the limit.

[bytebuf-limit](#)

Returns the limit of a bytebuf.

[bytebuf-ensure-free-capacity!](#)

Ensure that the bytebuf has a free capacity. Returns the widened bytebuf.

[bytebuf-limit!](#)

Set a new limit for the buffer. The new limit must not be larger than the capacity.

[section](#)

[top](#)

bytebuf-pos!

(bytebuf-pos! buf pos)

Sets the buffer's position.

```
(-> (bytebuf-allocate 10)  
    (bytebuf-pos! 4)  
    (bytebuf-put-byte! 1)  
    (bytebuf-pos! 8)  
    (bytebuf-put-byte! 2))  
=> [0 0 0 0 1 0 0 0 2 0]
```

[section](#)

[top](#)

bytebuf-put-buf!

(bytebuf-put-buf! dst src src-offset length)

This method transfers bytes from the src to the dst buffer at the current position, and then increments the position by length.

```
(-> (bytebuf-allocate 10)
    (bytebuf-pos! 4)
    (bytebuf-put-buf! (bytebuf [1 2 3]) 0 2))
=> [0 0 0 0 1 2 0 0 0 0]
```

[section](#)[top](#)

bytebuf-put-byte!

```
(bytebuf-put-byte! buf b)
```

Writes a byte to the buffer at the current position, and then increments the position by one.

```
(-> (bytebuf-allocate 4)
    (bytebuf-put-byte! 1)
    (bytebuf-put-byte! 2))
=> [1 2 0 0]
```

[section](#)[top](#)

bytebuf-put-double!

```
(bytebuf-put-double! buf d)
```

Writes a double (8 bytes) to buffer at the current position, and then increments the position by eight.

```
(-> (bytebuf-allocate 16)
    (bytebuf-put-double! 64.0)
    (bytebuf-put-double! 200.0))
=> [64 80 0 0 0 0 0 0 64 105 0 0 0 0 0 0]
```

[section](#)[top](#)

bytebuf-put-float!

```
(bytebuf-put-float! buf d)
```

Writes a float (4 bytes) to buffer at the current position, and then increments the position by four.

```
(-> (bytebuf-allocate 8)
    (bytebuf-put-float! 64.0)
    (bytebuf-put-float! 200.0))
=> [66 128 0 0 67 72 0 0]
```

[section](#)[top](#)

bytebuf-put-int!

```
(bytebuf-put-int! buf i)
```

Writes an integer (4 bytes) to buffer at the current position, and then increments the position by four.

```
(-> (bytebuf-allocate 8)
    (bytebuf-put-int! 4I)
    (bytebuf-put-int! 8I))
=> [0 0 0 4 0 0 0 8]
```

[section](#)

[top](#)

bytebuf-put-long!

```
(bytebuf-put-long! buf 1)
```

Writes a long (8 bytes) to buffer at the current position, and then increments the position by eight.

```
(-> (bytebuf-allocate 16)
    (bytebuf-put-long! 4)
    (bytebuf-put-long! 8))
=> [0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 8]
```

[section](#)

[top](#)

bytebuf-remaining

```
(bytebuf-remaining buf)
```

Returns the number of bytes between the current position and the limit.

```
(bytebuf-capacity (bytebuf-allocate 100))
=> 100
```

SEE ALSO

[bytebuf-capacity](#)

Returns the capacity of a bytebuf.

[bytebuf-limit](#)

Returns the limit of a bytebuf.

[bytebuf-pos](#)

Returns the buffer's current position.

[bytebuf-ensure-free-capacity!](#)

Ensure that the bytebuf has a free capacity. Returns the widened bytebuf.

[bytebuf-limit!](#)

Set a new limit for the buffer. The new limit must not be larger than the capacity.

[section](#)

[top](#)

bytebuf-sub

```
(bytebuf-sub x start) (bytebuf-sub x start end)
```

Returns a byte buffer of the items in buffer from start (inclusive) to end (exclusive). If end is not supplied, defaults to (count bytebuffer)

```
(bytebuf-sub (bytebuf [1 2 3 4 5 6]) 2)
=> [3 4 5 6]
```

```
(bytebuf-sub (bytebuf [1 2 3 4 5 6]) 4)
=> [5 6]
```

[section](#)

[top](#)

bytebuf-to-list

```
(bytebuf-to-list buf)
```

Returns the bytebuf as lazy list of integers

```
(doall (bytebuf-to-list (bytebuf [97 98 99])))
=> (97I 98I 99I)
```

[section](#)

[top](#)

bytebuf-to-string

```
(bytebuf-to-string buf)
(bytebuf-to-string buf encoding)
```

Converts a bytebuf to a string using an optional encoding. The encoding defaults to :UTF-8

```
(bytebuf-to-string (bytebuf [97 98 99]) :UTF-8)
=> "abc"
```

SEE ALSO

[bytebuf-from-string](#)

Converts a string to a bytebuf using an optional encoding. The encoding defaults to :UTF-8

[section](#)

[top](#)

bytebuf?

```
(bytebuf? x)
```

Returns true if x is a bytebuf

```
(bytebuf? (bytebuf [1 2]))
=> true
```

```
(bytebuf? [1 2])
=> false
```

```
(bytebuf? nil)
=> false
```

[section](#)[top](#)

callstack

```
(callstack)
```

Returns the current callstack.

```
(do
  (defn f1 [x] (f2 x))
  (defn f2 [x] (f3 x))
  (defn f3 [x] (f4 x))
  (defn f4 [x] (callstack))
  (f1 100))
=> [{:fn-name "callstack" :file "example" :line 74 :col 18} {:fn-name "user/f4" :file "example" :line 73 :col 18}
{:fn-name "user/f3" :file "example" :line 72 :col 18} {:fn-name "user/f2" :file "example" :line 71 :col 18}
{:fn-name "user/f1" :file "example" :line 75 :col 5}]
```

[section](#)[top](#)

cancel

```
(cancel f)
```

Cancels a future or a promise

```
(do
  (def wait (fn [] (sleep 400) 100))
  (let [f (future wait)]
    (sleep 50)
    (printf "After 50ms: cancelled=%b\n" (cancelled? f))
    (cancel f)
    (sleep 100)
    (printf "After 150ms: cancelled=%b\n" (cancelled? f))))
```

After 50ms: cancelled=false

After 150ms: cancelled=true

```
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[done?](#)

Returns true if the future or promise is done otherwise false

[cancelled?](#)

Returns true if the future or promise is cancelled otherwise false

[section](#)[top](#)

cancelled?

```
(cancelled? f)
```

Returns true if the future or promise is cancelled otherwise false

```
(cancelled? (future (fn [] 100)))  
=> false
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[promise](#)

Returns a promise object that can be read with deref, and set, once only, with deliver. Calls to deref prior to delivery will block, ...

[done?](#)

Returns true if the future or promise is done otherwise false

[cancel](#)

Cancels a future or a promise

[section](#)[top](#)

cargo-arangodb/db-dump

```
(cargo-arangodb/db-dump cname db-name db-user db-passwd dump-name log)
```

Dumps an ArangoDB database.

The DB dump is written to container's directory `/var/lib/arangodb3/{dump-name}`. If the directory does not exist it is created automatically.

Example:

```
(do  
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])  
  
  ;; (cargo-arangodb/exists-db-dump? "db-test" "dump-001")  
  ;; (cargo-arangodb/remove-db-dump "db-test" "dump-001")  
  
  ;; Dump the 'people' database to 'dump-001'  
  (ca/db-dump "db-test" "people" "root" "xxx" "dump-001" nil))
```

Args:

cname	The container name
db-name	The name of the DB to dump
db-user	The DB user
db-passwd	The DB password
dump-name	The dump name. e.g "dump-001"
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "ArangoDB:" s))</code>

Dumps an ArangoDB database using this commands on the container:

```
mkdir /var/lib/arangodb3/dump-001  
  
arangodump  
  --output-directory /var/lib/arangodb3/dump-001  
  --overwrite true
```

```
--include-system-collections true
--server.database "people"
--server.endpoint tcp://127.0.0.1:8529
--server.username "root"
--server.password "xxx"
```

Open an interactive docker shell to check the dump:

```
docker exec -it {container-id} sh
```

ZIP a dump

```
docker exec -it {container-id}
    zip -r
        /var/lib/arangodb3/dump-001.zip
        /var/lib/arangodb3/dump-001
```

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])

  (ca/db-dump "db-test" "people" "root" "xxx" "dump-001" nil))
```

SEE ALSO

[cargo-arangodb/db-restore](#)

Restores an ArangoDB database from a dump

[cargo-arangodb/exists-db-dump?](#)

Returns true if the dump with the given name exists otherwise false.

[cargo-arangodb/remove-db-dump](#)

Removes an existing DB dump.

[cargo-arangodb/download-db-dump](#)

Downloads an existing the DB dump 'dump-name' from the container to the local filesystem. The export directory in the local filesystem ...

[cargo-arangodb/upload-db-dump](#)

Uploads an existing DB dump with the name 'dump-name' from the local filesystem to the container. The import directory on local filesystem ...

[section](#)

[top](#)

cargo-arangodb/db-restore

```
(cargo-arangodb/db-restore cname db-name db-user db-passwd dump-name log)
```

Restores an ArangoDB database from a dump

The DB dump is read from container's directory "/var/lib/arangodb3/{dump-name}".

Example:

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])

  ;; (cargo-arangodb/exists-db-dump? "db-test" "dump-001")
  ;; (cargo-arangodb/remove-db-dump "db-test" "dump-001")

  ;; Restore the 'people' database from 'dump-001'
  (ca/db-restore "db-test" "people" "root" "xxx" "dump-001" nil))
```

Args:

cname	The container name
db-name	The name of the DB to dump
db-user	The DB user

db-passwd	The DB password
dump-name	The dump name. e.g "dump-001"
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "ArangoDB:" s))</code>

Restores an ArangoDB database using this command on the container:

```
arangorestore
--input-directory /var/lib/arangodb3/dump-001
--force-same-database
--create-database true
--include-system-collections true
--server.database "people"
--server.endpoint tcp://127.0.0.1:8529
--server.username "root"
--server.password "xxx"
```

Open an interactive docker shell to check the dump:

```
docker exec -it {container-id} sh
```

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/db-restore "db-test" "people" "root" "xxx" "dump-001" nil))
```

SEE ALSO

[cargo-arangodb/db-dump](#)

Dumps an ArangoDB database.

[cargo-arangodb/exists-db-dump?](#)

Returns true if the dump with the given name exists otherwise false.

[cargo-arangodb/remove-db-dump](#)

Removes an existing DB dump.

[cargo-arangodb/download-db-dump](#)

Downloads an existing the DB dump 'dump-name' from the container to the local filesystem. The export directory in the local filesystem ...

[cargo-arangodb/upload-db-dump](#)

Uploads an existing DB dump with the name 'dump-name' from the local filesystem to the container. The import directory on local filesystem ...

[section](#)

[top](#)

cargo-arangodb/download-db-dump

```
(cargo-arangodb/download-db-dump cname dump-name export-dir log)
```

Downloads an existing the DB dump 'dump-name' from the container to the local filesystem. The export directory in the local filesystem must be an existing directory.

Args:

cname	The container name
dump-name	The dump name
export-dir	The export dir. E.g: <code>(io/file (io/user-home-dir) "dump")</code>
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "ArangoDB:" s))</code>

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])

  ;; create a DB dump
```

```
(ca/db-dump "db-test" "people" "root" "xxx" "dump-001" nil)

;; downloads the DB dump to the local filesystem
(let [dir (io/user-home-dir)]
  (ca/download-db-dump "db-test" "dump-001" dir nil)))
```

SEE ALSO

[cargo-arangodb/upload-db-dump](#)

Uploads an existing DB dump with the name 'dump-name' from the local filesystem to the container. The import directory on local filesystem ...

[cargo-arangodb/db-dump](#)

Dumps an ArangoDB database.

[cargo-arangodb/db-restore](#)

Restores an ArangoDB database from a dump

[cargo-arangodb/exists-db-dump?](#)

Returns true if the dump with the given name exists otherwise false.

[section](#)

[top](#)

cargo-arangodb/exists-db-dump?

```
(cargo-arangodb/exists-db-dump? cname dump-name)
```

Returns true if the dump with the given name exists otherwise false.

Args:

cname	The container name
dump-name	The dump name

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/exists-db-dump? "db-test" "dump-001"))
```

SEE ALSO

[cargo-arangodb/db-dump](#)

Dumps an ArangoDB database.

[cargo-arangodb/db-restore](#)

Restores an ArangoDB database from a dump

[cargo-arangodb/remove-db-dump](#)

Removes an existing DB dump.

[section](#)

[top](#)

cargo-arangodb/list-db-dumps

```
(cargo-arangodb/list-db-dumps cname)
```

List the created DB dumps.

Args:

cname	The container name

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/list-db-dumps "db-test"))
```

SEE ALSO

[cargo-arangodb/db-dump](#)

Dumps an ArangoDB database.

[cargo-arangodb/db-restore](#)

Restores an ArangoDB database from a dump

[cargo-arangodb/exists-db-dump?](#)

Returns true if the dump with the given name exists otherwise false.

[section](#)

[top](#)

cargo-arangodb/logs

```
(cargo-arangodb/logs cname)
(cargo-arangodb/logs cname lines)
```

Prints the ArangoDB docker container logs

Args:

cname A unique container name

lines The number of tail lines

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/logs "db-test"))
```

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/logs "db-test" 100))
```

SEE ALSO

[cargo-arangodb/start](#)

Starts an ArangoDB container.

[cargo-arangodb/running?](#)

Returns true if a container with the specified name is running.

[section](#)

[top](#)

cargo-arangodb/remove-db-dump

```
(cargo-arangodb/remove-db-dump cname dump-name)
```

Removes an existing DB dump.

Args:

cname The container name

dump-name The dump name

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/remove-db-dump "db-test" "dump-001"))
```

SEE ALSO

[cargo-arangodb/db-dump](#)

Dumps an ArangoDB database.

[cargo-arangodb/db-restore](#)

Restores an ArangoDB database from a dump

[cargo-arangodb/exists-db-dump?](#)

Returns true if the dump with the given name exists otherwise false.

[section](#)

[top](#)

cargo-arangodb/running?

```
(cargo-arangodb/running? cname)
```

Returns true if a container with the specified name is running.

Args:

cname A unique container name

```
;; Test if ArangoDB container is running
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])
  (ca/running? "db-test"))
```

SEE ALSO

[cargo-arangodb/logs](#)

Prints the ArangoDB docker container logs

[cargo-arangodb/start](#)

Starts an ArangoDB container.

[cargo-arangodb/stop](#)

Stops an ArangoDB container

[section](#)

[top](#)

cargo-arangodb/start

```
(cargo-arangodb/start cname version mapped-port root-passwd memory cores log)
(cargo-arangodb/start cname version volumes mapped-port root-passwd memory cores log)
```

Starts an ArangoDB container.

Start rules:

- If a container with another version exists for the container name remove the container and the image
- Pull the image if not yet locally available
- If the container already runs - use it
- If the container is available but does not run - start it `(docker/start ...)`
- If the container is not available - run it `(docker/run ...)`

- Finally check for a successful startup. The container log must contain the string *".is ready for business. Have fun."* on the last line.

Args:

cname	A unique container name
version	The ArangoDB version to use. E.g.: 3.11.4
mapped-port	The published (mapped) ArangoDB port on the host
root-passwd	The ArangoDB root password
memory	The detected memory ArangoDB is to use. E.g.: 8GB, 8000MB
cores	The detected number of cores ArangoDB is to use
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "ArangoDB:" s))</code>

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])

  ;; Run an ArangoDB container labeled as "db-test"
  (ca/start "db-test" "3.11.4" 8500 "test" "8GB" 1 nil))
```

SEE ALSO

[cargo-arangodb/logs](#)

Prints the ArangoDB docker container logs

[cargo-arangodb/stop](#)

Stops an ArangoDB container

[cargo-arangodb/running?](#)

Returns true if a container with the specified name is running.

section top

cargo-arangodb/stop

```
(cargo-arangodb/stop cname)
(cargo-arangodb/stop cname log)
```

Stops an ArangoDB container

Args:

cname	A unique container name
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "ArangoDB:" s))</code>

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])

  ;; Stop the ArangoDB container labeled as "db-test"
  (ca/stop "db-test" nil))
```

SEE ALSO

[cargo-arangodb/start](#)

Starts an ArangoDB container.

[cargo-arangodb/running?](#)

Returns true if a container with the specified name is running.

section top

cargo-arangodb/upload-db-dump

```
(cargo-arangodb/upload-db-dump cname dump-name import-dir log)
```

Uploads an existing DB dump with the name 'dump-name' from the local filesystem to the container. The import directory on local filesystem must be an existing non empty directory.

Args:

cname	The container name
dump-name	The dump name
import-dir	The import dir. E.g.: <code>(io/file (io/user-home-dir) "dump")</code>
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "ArangoDB:" s))</code>

```
(do
  (load-module :cargo-arangodb ['cargo-arangodb :as 'ca])

  ;; upload the dump to the container's filesystem
  (let [dir (io/file (io/user-home-dir) "dump-001")]
    (ca/upload-db-dump "db-test" "dump-001" dir nil))

  ;; restore the DB dump
  (ca/db-restore "db-test" "people" "root" "xxx" "dump-001" nil))
```

SEE ALSO

[cargo-arangodb/download-db-dump](#)

Downloads an existing the DB dump 'dump-name' from the container to the local filesystem. The export directory in the local filesystem ...

[cargo-arangodb/db-dump](#)

Dumps an ArangoDB database.

[cargo-arangodb/db-restore](#)

Restores an ArangoDB database from a dump

[cargo-arangodb/exists-db-dump?](#)

Returns true if the dump with the given name exists otherwise false.

[section](#)

[top](#)

cargo-postgresql/logs

```
(cargo-postgresql/logs cname)
(cargo-postgresql/logs cname lines)
```

Prints the PostgreSQL docker container logs

Args:

cname	A unique container name
lines	The number of tail lines

```
(do
  (load-module :cargo-postgresql ['cargo-postgresql :as 'pg])
  (pg/logs "postgres"))
```

```
(do
  (load-module :cargo-postgresql ['cargo-postgresql :as 'pg])
  (pg/logs "postgres" 100))
```

SEE ALSO

[cargo-postgresql/start](#)

Starts a PostgreSQL container.

[cargo-postgresql/running?](#)

Returns true if a container with the specified name is running.

[section](#)

[top](#)

cargo-postgresql/running?

```
(cargo-postgresql/running? cname)
```

Returns true if a container with the specified name is running.

Args:

cname A unique container name

```
;; Test if PostgreSQL container is running
(do
  (load-module :cargo-postgresql ['cargo-postgresql :as 'pg])
  (pg/running? "postgres"))
```

SEE ALSO

[cargo-postgresql/logs](#)

Prints the PostgreSQL docker container logs

[cargo-postgresql/start](#)

Starts a PostgreSQL container.

[cargo-postgresql/stop](#)

Stops a PostgreSQL container

[section](#)

[top](#)

cargo-postgresql/start

```
(cargo-postgresql/start cname version storage-dir)
(cargo-postgresql/start cname version storage-dir user password)
(cargo-postgresql/start cname version storage-dir user password log)
(cargo-postgresql/start cname version mapped-port storage-dir user password log)
```

Starts a PostgreSQL container.

Start rules:

- If a container with another version exists for the container name remove the container and the image
- Pull the image if not yet locally available
- If the container already runs - use it
- If the container is available but does not run - start it (docker/start ...)

- If the container is not available - run it `(docker/run ...)`
- Finally check for a successful startup. The container error log must contain the string *"database system is ready to accept connections."* on the last few lines.

Args:

cname	A unique container name
version	The PostgreSQL version to use. E.g.: "16.2", "16"
mapped-port	The published (mapped) port on the host. Defaults to 5432
storage-dir	Directory where PostgreSQL persists all the data.
user	A user. Defaults to "postgres"
password	A password. Defaults to "postgres"
log	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "PostgreSQL:" s))</code>

```
(do
  (load-module :cargo-postgresql ['cargo-postgresql :as 'pg])

  ;; Run a PostgreSQL container labeled as "postgres"
  (pg/start "postgres" "16.2" "./postgres-storage"))
```

SEE ALSO

[cargo-postgresql/stop](#)

Stops a PostgreSQL container

[cargo-postgresql/running?](#)

Returns true if a container with the specified name is running.

[cargo-postgresql/logs](#)

Prints the PostgreSQL docker container logs

section

top

cargo-postgresql/stop

```
(cargo-postgresql/stop cname)
(cargo-postgresql/stop cname log)
```

Stops a PostgreSQL container

Args:

cname	A unique container name

```
(do
  (load-module :cargo-postgresql ['cargo-postgresql :as 'pg])

  ;; Stop the PostgreSQL container labeled as "postgres"
  (pg/stop "postgres"))
```

SEE ALSO

[cargo-postgresql/start](#)

Starts a PostgreSQL container.

[cargo-postgresql/running?](#)

Returns true if a container with the specified name is running.

section

top

cargo-qdrant/logs

```
(cargo-qdrant/logs cname)
(cargo-qdrant/logs cname lines)
```

Prints the Qdrant docker container logs

Args:

cname A unique container name
lines The number of tail lines

```
(do
  (load-module :cargo-qdrant ['cargo-qdrant :as 'cq])
  (cq/logs "qdrant"))

(do
  (load-module :cargo-qdrant ['cargo-qdrant :as 'cq])
  (cq/logs "qdrant" 100))
```

SEE ALSO

[cargo-qdrant/start](#)

Starts a Qdrant container.

[cargo-qdrant/running?](#)

Returns true if a container with the specified name is running.

[section](#)

[top](#)

cargo-qdrant/running?

```
(cargo-qdrant/running? cname)
```

Returns true if a container with the specified name is running.

Args:

cname A unique container name

```
;; Test if Qdrant container is running
(do
  (load-module :cargo-qdrant ['cargo-qdrant :as 'cq])
  (cq/running? "qdrant"))
```

SEE ALSO

[cargo-qdrant/logs](#)

Prints the Qdrant docker container logs

[cargo-qdrant/start](#)

Starts a Qdrant container.

[cargo-qdrant/stop](#)

Stops a Qdrant container

[section](#)

[top](#)

cargo-qdrant/start

```
(cargo-qdrant/start cname version storage-dir)
(cargo-qdrant/start cname version storage-dir config-file log)
(cargo-qdrant/start cname version mapped-rest-port mapped-grpc-port storage-dir config-file log)
```

Starts a Qdrant container.

Qdrant is vector database often used for LLM embeddings.

Telemetry reporting is disabled by setting the env variable QDRANT__TELEMETRY_DISABLED to `true`.

Start rules:

- If a container with another version exists for the container name remove the container and the image
- Pull the image if not yet locally available
- If the container already runs - use it
- If the container is available but does not run - start it `(docker/start ...)`
- If the container is not available - run it `(docker/run ...)`
- Finally check for a successful startup. The container log must contain the string `".Qdrant HTTP listening on."` on the last line.

Args:

<code>cname</code>	A unique container name
<code>version</code>	The Qdrant version to use. E.g.: "1.8.3"
<code>mapped-rest-port</code>	The published (mapped) Qdrant REST port on the host. Defaults to 6333
<code>mapped-grpc-port</code>	The published (mapped) Qdrant GRPC port on the host. Defaults to 6334
<code>storage-dir</code>	Directory where Qdrant persists all the data.
<code>config-file</code>	An optional custom configuration yaml file
<code>log</code>	A log function, may be <i>nil</i> . E.g: <code>(fn [s] (println "Qdrant:" s))</code>

```
(do
  (load-module :cargo-qdrant ['cargo-qdrant :as 'cq])

  ;; Run a Qdrant container labeled as "qdrant"
  (cq/start "qdrant" "1.8.3" "./qdrant-storage"))
```

SEE ALSO

[cargo-qdrant/stop](#)

Stops a Qdrant container

[cargo-qdrant/running?](#)

Returns true if a container with the specified name is running.

[cargo-qdrant/logs](#)

Prints the Qdrant docker container logs

[section](#)

[top](#)

cargo-qdrant/stop

```
(cargo-qdrant/stop cname)
(cargo-qdrant/stop cname log)
```

Stops a Qdrant container

Args:

cname A unique container name

```
(do
  (load-module :cargo-qdrant ['cargo-qdrant :as 'cq])

  ;; Stop the Qdrant container labeled as "qdrant"
  (cq/stop "qdrant"))
```

SEE ALSO

[cargo-qdrant/start](#)

Starts a Qdrant container.

[cargo-qdrant/running?](#)

Returns true if a container with the specified name is running.

[section](#)

[top](#)

cargo/purge

```
(cargo/purge cname)
```

Removes a container and its image. The container must not be running.

Args:

cname A unique container name

```
;; Purge an ArangoDB container
(cargo/purge "arangodb-test")
```

SEE ALSO

[cargo/start](#)

Starts a container.

[cargo/stop](#)

Stops a container

[cargo/running?](#)

Returns true if a container with the specified name is running.

[section](#)

[top](#)

cargo/running?

```
(cargo/running? cname)
```

Returns true if a container with the specified name is running.

Args:

cname A unique container name

```
;; Test if ArangoDB container is running
(cargo/running? "arangodb-test")
```

SEE ALSO

[cargo/start](#)

Starts a container.

[cargo/stop](#)

Stops a container

[cargo/purge](#)

Removes a container and its image. The container must not be running.

[section](#)

[top](#)

cargo/start

```
(cargo/start cname repo version publish envs args ready? log)
```

```
(cargo/start cname repo version publish envs args ready? log wait-after-start-secs ready-check-max-secs)
```

Starts a container.

Start rules:

- If a container with the passed name exists or is running in another version, stop that container and remove it together with the image
- Pull the image if it is not yet locally available
- If the container runs with the requested version already - use it
- If the container is available but does not run - start it using `(docker/start ...)`
- If the container is not available - run it using `(docker/run ...)`
- Finally check for a successful startup using the supplied `ready?` function. E.g.: `ready?` may scan the container logs for a successful startup message.

Args:

cname	A unique container name
repo	The image repository
version	The image version
publish	Publish a container's ports to the host. To expose port 8080 inside the container to port 3000 outside the container, pass <code>["3000:8080"]</code>
envs	A vector of env variables
vols	A vector of volume mounts
args	A vector of arguments for the process run in the container
ready?	A function to decide if the container is ready (may be <i>nil</i>). The function takes the unique container name as its single argument. It returns true if the container is ready else false
log	A log function (may be <i>nil</i>). The function takes a single string argument
wait-after-start-secs	Wait n seconds after starting the container (may be <i>nil</i>)
ready-check-max-secs	Try max n seconds for ready check (defaults to 30s if <i>nil</i>)

```
;; Run an ArangoDB container
```

```
(cargo/start "arangodb-test"
  "arangodb/arangodb"
  "3.11.4"
  ["8500:8529"]
  ["ARANGO_ROOT_PASSWORD=test"
   "ARANGODB_OVERRIDE_DETECTED_TOTAL_MEMORY=8GB"
   "ARANGODB_OVERRIDE_DETECTED_NUMBER_OF_CORES=1"]
  []
  ["--server.endpoint tcp://0.0.0.0:8529"]
  (fn [cname]
    (-> (docker/container-logs cname :tail 1)
```

```
        (str/trim)
        (match? #".*is ready for business. Have fun.*"))))
(fn [s] (println "ArangoDB:" s))
3
30)
```

SEE ALSO

[cargo/stop](#)

Stops a container

[cargo/running?](#)

Returns true if a container with the specified name is running.

[cargo/purge](#)

Removes a container and its image. The container must not be running.

[section](#)

[top](#)

cargo/stop

```
(cargo/stop cname log)
```

Stops a container

Args:

cname A unique container name

log A log function (may be *nil*). The function takes a single string argument

```
;; Stop an ArangoDB container
(cargo/stop "arangodb-test"
  (fn [s] (println "ArangoDB:" s)))
```

SEE ALSO

[cargo/start](#)

Starts a container.

[cargo/running?](#)

Returns true if a container with the specified name is running.

[cargo/purge](#)

Removes a container and its image. The container must not be running.

[section](#)

[top](#)

cartesian-product

```
(cartesian-product coll1 coll2 coll*)
```

Returns the cartesian product of two or more collections.

Removes all duplicate items in the collections before computing the cartesian product.

```
(cartesian-product [1 2 3] [1 2 3])
=> ((1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3))
```

```
(cartesian-product [0 1] [0 1] [0 1])
=> ((0 0 0) (0 0 1) (0 1 0) (0 1 1) (1 0 0) (1 0 1) (1 1 0) (1 1 1))
```

SEE ALSO

[combinations](#)

All the unique ways of taking n different elements from the items in the collection

[section](#)

[top](#)

case

```
(case expr & clauses)
```

Takes an expression and a set of clauses. Each clause takes the form of test-constant result-expr

```
(case (+ 1 9)
  10 :ten
  20 :twenty
  30 :thirty
  :dont-know)
=> :ten
```

SEE ALSO

[cond](#)

Takes a set of test/expr pairs. It evaluates each test one at a time. If a test returns logical true, cond evaluates and returns the ...

[condp](#)

Takes a binary predicate, an expression, and a set of clauses.

[section](#)

[top](#)

cast

```
(cast class object)
```

Casts a Java object to a specific type

Note: Casting a Java object will change the object's *formal type*. See the `formal-type` function for detailed information.

```
(do
  (import :java.awt.Point)
  (import :java.awt.geom.Point2D)

  ;; upcasting :java.awt.Point to :java.awt.geom.Point2D
  ;; Point2D does not define the translate method!
  (let [p1 (. :Point new 1.0 1.0)
        p2 (cast :Point2D p1)]
    (println "p1 ->" p1)
    (println "p2 ->" p2)
    (println "Formal type p1 ->" (formal-type p1))
    (println "Formal type p2 ->" (formal-type p2))
    (println "p1' ->" (doto p1 (. :translate 2.0 2.0)))
    ;; the translate method is not defined by Point2D
    ;; and will fail with a JavaMethodInvocationException!
    ;; (doto p2 (. :translate 2.0 2.0))
```

```
)  
p1 -> java.awt.Point[x=1,y=1]  
p2 -> java.awt.Point[x=1,y=1]  
Formal type p1 -> :java.awt.Point  
Formal type p2 -> :java.awt.geom.Point2D  
p1' -> java.awt.Point[x=3,y=3]  
=> nil
```

SEE ALSO

[formal-type](#)

Returns the formal type of a Java object.

[remove-formal-type](#)

Removes the formal type from a Java object.

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[section](#)

[top](#)

ceil

```
(ceil x)
```

Returns the largest integer that is greater than or equal to x

```
(ceil 1.4)  
=> 2.0
```

```
(ceil -1.4)  
=> -1.0
```

```
(ceil 1.23M)  
=> 2.00M
```

```
(ceil -1.23M)  
=> -1.00M
```

SEE ALSO

[floor](#)

Returns the largest integer that is less than or equal to x

[section](#)

[top](#)

char

```
(char c)
```

Converts a number or s single char string to a char.

```
(char 65)  
=> #\A
```

```
(char "A")
=> #\A

(long (char "A"))
=> 65

(str/join (map char [65 66 67 68]))
=> "ABCD"

(map #(- (long %) (long (char "0")))) (str/chars "123456")
=> (1 2 3 4 5 6)
```

SEE ALSO

[char?](#)

Returns true if s is a char.

[section](#)

[top](#)

char-escaped

(char-escaped c)

Returns the ASCII escaped character for c.

- `\'` single quote
- `\"` double quote
- `\\` backslash
- `\n` new line
- `\r` carriage return
- `\t` tab
- `\b` backspace
- `\f` form feed
- `\0` null character
- in all other cases returns the character c

```
(char-escaped #\n)
=> #\newline
```

```
(char-escaped #\a)
=> #\a
```

SEE ALSO

[char](#)

Converts a number or s single char string to a char.

[char?](#)

Returns true if s is a char.

[section](#)

[top](#)

char-literals

(char-literals)

Returns all defined char literals.

Char Literal	Unicode	Char
#\space	\u0020	#\space
#\newline	\u000A	#\newline
#\tab	\u0009	#\tab
#\formfeed	\u000C	#\formfeed
#\return	\u000D	#\return
#\backspace	\u0008	#\backspace
#\lparen	\u0028	#\l
#\rparen	\u0029	#\r
#\quote	\u0022	#\"
#\backslash	\u005C	#\backslash
#\pildcrow	\u00B6	#\¶
#\middle-dot	\u00B7	#\·
#\right-guillemet	\u00BB	#\»
#\left-guillemet	\u00AB	#\«
#\copyright	\u00A9	#\©
#\bullet	\u2022	#\•
#\horz-ellipsis	\u2026	#\…
#\per-mille-sign	\u2030	#\‰
#\diameter-sign	\u2300	#\∅
#\check-mark	\u2713	#\✓
#\cross-mark	\u2717	#\✗
#\pi	\u03C0	#\π
#\nbsp	\u00A0	#\
#\en-space	\u2002	#\
#\em-space	\u2003	#\
#\three-per-em-space	\u2004	#\
#\four-per-em-space	\u2005	#\
#\six-per-em-space	\u2006	#\

([char-literals](#))

SEE ALSO

[char](#)

Converts a number or s single char string to a char.

[char?](#)

Returns true if s is a char.

[section](#) [top](#)

char?

(char? s)

Returns true if s is a char.

([char?](#) #\a)

=> true

SEE ALSO

[char](#)

Converts a number or s single char string to a char.

[section](#)

[top](#)

charset-default-encoding

(charset-default-encoding)

Returns the default charset of this Java virtual machine.

```
(charset-default-encoding)
```

```
=> :UTF-8
```

[section](#)

[top](#)

chinook-postgresql/download-data

(download-data)

Download the Chinook dataset for PostgreSQL.

The data set is downloaded from [GitHub/lerocha](#)

The data set is published under the [License](#)

```
(do
  (load-module :chinook-postgresql ['chinook-postgresql :as 'chinook])
  (chinook/download-data))
```

SEE ALSO

[chinook-postgresql/show-data-model](#)

Opens a browser to show the Chinook data model (<https://github.com/lerocha/chinook-database/tree/master#data-model>)

[chinook-postgresql/load-data](#)

Load the Chinook dataset to a PostgreSQL database.

[section](#)

[top](#)

chinook-postgresql/load-data

(load-data)

Load the Chinook dataset to a PostgreSQL database.

The data set is loaded from [GitHub/lerocha](#)

[Data Model](#) published under the [License](#)

The Chinook sample database has 11 tables as follows:

employees	stores employee data such as id, last name, first name, etc. It also has a field named ReportsTo to specify who reports to whom

customers	stores customer data
invoices	stores invoice header data
invoice_items	stores the invoice line items data
artists	stores artist data. It is a simple table that contains the id and name
albums	stores data about a list of tracks. Each album belongs to one artist. However, one artist may have multiple albums
media_types	stores media types such as MPEG audio and AAC audio files
genres	stores music types such as rock, jazz, metal, etc.
tracks	stores the data of songs. Each track belongs to one album
playlists	stores data about playlists. Each playlist contains a list of tracks. Each track may belong to multiple playlists. The relationship between the playlists and tracks tables is many-to-many. The playlist_track table is used to reflect this relationship
playlist_track	reflect the many-to-many relationship between playlist and tracks

Start the PostgreSQL docker container:

```
(do
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (jdbp/start "postgres" "16.2" 5432 "./postgres-storage" "postgres" "postgres"))
```

Note: The storage directory (e.g. `./postgres-storage`) must exist!

```
(do
  (load-module :chinook-postgresql ['chinook-postgresql :as 'chinook])
  (chinook/load-data "localhost" 5432 "postgres" "postgres"))

(do
  (load-module :chinook-postgresql ['chinook-postgresql :as 'chinook])
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost"
                                          5432
                                          chinook/database
                                          "postgres"
                                          "postgres")])
  (-> (jdbc/execute-query conn "SELECT * FROM album WHERE title LIKE '%Mozart%'"
    (jdbc/print-query-result))))
```

SEE ALSO

[chinook-postgresql/show-data-model](#)

Opens a browser to show the Chinook data model (<https://github.com/lerocha/chinook-database/tree/master#data-model>)

[chinook-postgresql/download-data](#)

Download the Chinook dataset for PostgreSQL.

section

top

chinook-postgresql/show-data

```
(show-data)
```

Opens a browser to show the [Chinook data](#)

```
(do
  (load-module :chinook-postgresql ['chinook-postgresql :as 'chinook])
  (chinook/show-data))
```

SEE ALSO

[chinook-postgresql/show-data-model](#)

Opens a browser to show the Chinook data model (<https://github.com/lerocha/chinook-database/tree/master#data-model>)

[chinook-postgresql/download-data](#)

Download the Chinook dataset for PostgreSQL.

[chinook-postgresql/load-data](#)

Load the Chinook dataset to a PostgreSQL database.

[section](#)

[top](#)

chinook-postgresql/show-data-model

```
(show-data-model)
```

Opens a browser to show the [Chinook data model](#)

```
(do
  (load-module :chinook-postgresql ['chinook-postgresql :as 'chinook])
  (chinook/show-data-model))
```

SEE ALSO

[chinook-postgresql/show-data](#)

Opens a browser to show the Chinook data ([https://raw.githubusercontent.com/lerocha/chinook-database/master/ChinookDatabase/DataSources/Chin ...](https://raw.githubusercontent.com/lerocha/chinook-database/master/ChinookDatabase/DataSources/Chin...))

[chinook-postgresql/download-data](#)

Download the Chinook dataset for PostgreSQL.

[chinook-postgresql/load-data](#)

Load the Chinook dataset to a PostgreSQL database.

[section](#)

[top](#)

cidr/end-inet-addr

```
(cidr/end-inet-addr cidr)
```

Returns the end inet address of a CIDR IP block.

```
(cidr/end-inet-addr "222.192.0.0/11")
=> /222.223.255.255
```

```
(cidr/end-inet-addr "2001:0db8:85a3:08d3:1319:8a2e:0370:7347/64")
=> /2001:db8:85a3:8d3:ffff:ffff:ffff:ffff
```

```
(cidr/end-inet-addr (cidr/parse "222.192.0.0/11"))
=> /222.223.255.255
```

[section](#)

[top](#)

cidr/in-range?

```
(cidr/in-range? ip cidr)
```

Returns true if the ip address is within the ip range of the cidr else false. ip may be a string or a :java.net.InetAddress, cidr may be a string or a CIDRJava object obtained from 'cidr/parse'.

```
(cidr/in-range? "222.220.0.0" "222.220.0.0/11")  
=> true
```

```
(cidr/in-range? (inet/inet-addr "222.220.0.0") "222.220.0.0/11")  
=> true
```

```
(cidr/in-range? "222.220.0.0" (cidr/parse "222.220.0.0/11"))  
=> true
```

[section](#)[top](#)

cidr/insert

```
(cidr/insert trie cidr value)
```

Insert a new CIDR / value relation into trie. Works with IPv4 and IPv6. Please keep IPv4 and IPv6 CIDRs in different tries.

```
(do  
  (let [trie (cidr/trie)]  
    (cidr/insert trie  
                  (cidr/parse "192.16.10.0/24")  
                  "Germany")  
    (cidr/lookup trie "192.16.10.15")))  
=> "Germany"
```

[section](#)[top](#)

cidr/lookup

```
(cidr/lookup trie ip)
```

Lookup the associated value of a CIDR in the trie. A cidr "192.16.10.0/24" or an inet address "192.16.10.15" can be passed as ip.

```
(do  
  (let [trie (cidr/trie)]  
    (cidr/insert trie  
                  (cidr/parse "192.16.10.0/24")  
                  "Germany")  
    (cidr/lookup trie "192.16.10.15")))  
=> "Germany"
```

[section](#)[top](#)

cidr/lookup-reverse

```
(cidr/lookup-reverse trie ip)
```

Reverse lookup a CIDR in the trie given an IP address

```
(do
  (let [trie (cidr/trie)]
    (cidr/insert trie
      (cidr/parse "192.16.10.0/24")
      "Germany")
    (cidr/lookup-reverse trie "192.16.10.15")))
=> 192.16.10.0/24: [/192.16.10.0 .. /192.16.10.255]
```

[section](#)

[top](#)

cidr/parse

```
(cidr/parse cidr)
```

Parses CIDR IP blocks to an IP address range. Supports both IPv4 and IPv6.

```
(cidr/parse "222.192.0.0/11")
=> 222.192.0.0/11: [/222.192.0.0 .. /222.223.255.255]

(cidr/parse "2001:0db8:85a3:08d3:1319:8a2e:0370:7347/64")
=> 2001:0db8:85a3:08d3:1319:8a2e:0370:7347/64: [/2001:db8:85a3:8d3:0:0:0:0 .. /2001:db8:85a3:8d3:ffff:ffff:ffff:ffff]
```

[section](#)

[top](#)

cidr/size

```
(cidr/size trie)
```

Returns the size of the trie.

```
(do
  (let [trie (cidr/trie)]
    (cidr/insert trie
      (cidr/parse "192.16.10.0/24")
      "Germany")
    (cidr/size trie)))
=> 1
```

[section](#)

[top](#)

cidr/start-inet-addr

```
(cidr/start-inet-addr cidr)
```

Returns the start inet address of a CIDR IP block.

```
(cidr/start-inet-addr "222.192.0.0/11")
=> /222.192.0.0
```

```
(cidr/start-inet-addr "2001:0db8:85a3:08d3:1319:8a2e:0370:7347/64")
=> /2001:db8:85a3:8d3:0:0:0:0
```

```
(cidr/start-inet-addr (cidr/parse "222.192.0.0/11"))
=> /222.192.0.0
```

[section](#)[top](#)

cidr/trie

```
(cidr/trie)
```

Create a new mutable concurrent CIDR trie.

```
(do
  (let [trie (cidr/trie)]
    (cidr/insert trie
      (cidr/parse "192.16.10.0/24")
      "Germany")
    (cidr/lookup trie "192.16.10.15")))
=> "Germany"
```

[section](#)[top](#)

circular-buffer

```
(circular-buffer capacity)
```

Creates a new circular buffer.

A circular buffer stores the N most recently inserted elements. If the circular buffer is already full, the oldest element (the head) will be evicted, and then the new element added at the tail.

The circular buffer does not permit `nil` elements.

```
(let [q (circular-buffer 3)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (put! q 4)
  (println q)
  (println (take! q)))
(2 3 4)
2
=> nil
```

SEE ALSO

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[empty](#)

Returns an empty collection of the same category as coll, or nil if coll is nil. If the collection is mutable clears the collection ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[circular-buffer?](#)

Returns true if coll is a circular buffer

[section](#)[top](#)

circular-buffer?

```
(circular-buffer? coll)
```

Returns true if coll is a circular buffer

```
(circular-buffer? (circular-buffer 20))  
=> true
```

[section](#)[top](#)

clamp

```
(clamp x min max)
```

Restricts a given value between a lower and upper bound. In this way, it acts like a combination of the `min` and `max` functions.

```
(clamp 1 10 20)  
=> 10
```

```
(clamp 1I 10I 20I)  
=> 10I
```

```
(clamp 1.0 10.0 20.0)  
=> 10.0
```

SEE ALSO

[min](#)

Returns the smallest of the values

[max](#)

Returns the greatest of the values

[section](#)[top](#)

class

```
(class name)
```


Returns the Java class for the given name. Throws an exception if the class is not found.

```
(class :java.util.ArrayList)
=> class java.util.ArrayList
```

SEE ALSO

[class-of](#)

Returns the Java class of a value.

[class-name](#)

Returns the Java class name of a class.

[class-version](#)

Returns the major version of a Java class.

[cast](#)

Casts a Java object to a specific type

[formal-type](#)

Returns the formal type of a Java object.

[remove-formal-type](#)

Removes the formal type from a Java object.

[section](#)

[top](#)

class-name

```
(class-name class)
```

Returns the Java class name of a class.

```
(class-name (class :java.util.ArrayList))
=> "java.util.ArrayList"
```

SEE ALSO

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[class-of](#)

Returns the Java class of a value.

[class-version](#)

Returns the major version of a Java class.

[section](#)

[top](#)

class-of

```
(class-of x)
```

Returns the Java class of a value.

```
(class-of 100)
=> class com.github.jlangch.venice.impl.types.VncLong
```

```
(class-of (. :java.awt.Point :new 10 10))  
=> class java.awt.Point
```

SEE ALSO

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[class-name](#)

Returns the Java class name of a class.

[class-version](#)

Returns the major version of a Java class.

[section](#)[top](#)

class-version

```
(class-version class)
```

Returns the major version of a Java class.

Java major versions:

- Java 8 uses major version 52
- Java 9 uses major version 53
- Java 10 uses major version 54
- Java 11 uses major version 55
- Java 12 uses major version 56
- Java 13 uses major version 57
- Java 14 uses major version 58
- Java 15 uses major version 59

```
(class-version :com.github.jlangch.venice.Venice)  
=> 52
```

SEE ALSO

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[class-of](#)

Returns the Java class of a value.

[class-name](#)

Returns the Java class name of a class.

[section](#)[top](#)

classloader

```
(classloader)  
(classloader type)
```

Returns the classloader.

```
;; Returns the current classloader  
(classloader)  
=> class sun.misc.Launcher$AppClassLoader
```

```
;; Returns the system classloader
(classloader :system)
=> sun.misc.Launcher$AppClassLoader@4e0e2f2a

;; Returns the classloader which loaded the Venice classes
(classloader :application)
=> sun.misc.Launcher$AppClassLoader@4e0e2f2a

;; Returns the thread-context classloader
(classloader :thread-context)
=> sun.misc.Launcher$AppClassLoader@4e0e2f2a
```

SEE ALSO

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[classloader-of](#)

Returns the classloader of a value or a Java class.

[section](#)[top](#)

classloader-of

```
(classloader-of x)
```

Returns the classloader of a value or a Java class.

Note:

Some Java VM implementations may use 'null' to represent the bootstrap class loader. This method will return 'nil' in such implementations if this class was loaded by the bootstrap class loader.

```
(classloader-of (class :java.awt.Point))
=> nil

(classloader-of (. :java.awt.Point :new 10 10))
=> nil

(classloader-of (class-of "abcdef"))
=> sun.misc.Launcher$AppClassLoader@4e0e2f2a

(classloader-of "abcdef")
=> sun.misc.Launcher$AppClassLoader@4e0e2f2a
```

SEE ALSO

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[classloader](#)

Returns the classloader.

[section](#)[top](#)

clear-taps

```
(clear-taps)
```

Removes all tap sets.

```
(do
  (add-tap prn)
  (clear-taps))
=> nil
```

SEE ALSO

[remove-tap](#)

Remove f from the tap set.

[add-tap](#)

adds f, a fn of one argument, to the tap set. This function will be called with anything sent via tap>.

[tap>](#)

Sends x to any taps. Will not block. Returns true if there was room in the queue, false if not (x is dropped).

[section](#)

[top](#)

coalesce

```
(coalesce args*)
```

Returns nil if all of its arguments are nil, otherwise it returns the first non nil argument. The arguments are evaluated lazy.

```
(coalesce)
=> nil
```

```
(coalesce 2)
=> 2
```

```
(coalesce nil 1 2)
=> 1
```

[section](#)

[top](#)

coll?

```
(coll? coll)
```

Returns true if coll is a collection

```
(coll? {:a 1})
=> true
```

```
(coll? [1 2])
=> true
```

[section](#)

[top](#)

combinations

```
(combinations coll n)
```

All the unique ways of taking n different elements from the items in the collection

```
(combinations [0 1 2 3] 1)
=> ([0] [1] [2] [3])
```

```
(combinations [0 1 2 3] 2)
=> ([0 1] [0 2] [0 3] [1 2] [1 3] [2 3])
```

```
(combinations [0 1 2 3] 3)
=> ([0 1 2] [0 1 3] [1 2 3])
```

```
(combinations [0 1 2 3] 4)
=> ([0 1 2 3])
```

SEE ALSO

[cartesian-product](#)

Returns the cartesian product of two or more collections.

[section](#)

[top](#)

comment

```
(comment & body)
```

Ignores body, yields nil

```
(comment
  (println 1)
  (println 5))
=> nil
```

[section](#)

[top](#)

comp

```
(comp f*)
```

Takes a set of functions and returns a function that is the composition of those functions. The returned function takes a variable number of args, applies the rightmost of the functions to the args, applies the next function (right-to-left) to the result, etc.

Functions composition builds complex functions by combining simpler ones. Functions are composed by passing the output of one function as the input to another.

```
h(x) = (g ° f)(x) = g(f(x))
```

The composition operator `°` can be understood as *after*. In other words, the function `g` is applied after the function `f` has been applied to `x`.

If you have two functions `f` and `g`, function composition allows you to create a new function `h` such that `h(x) = g(f(x))`.

```
((comp str +) 8 8 8)
=> "24"

(map (comp - (partial + 3) (partial * 2)) [1 2 3 4])
=> (-5 -7 -9 -11)

((reduce comp [(partial + 1) (partial * 2) (partial + 3)]) 100)
=> 207

(filter (comp not zero?) [0 1 0 2 0 3 0 4])
=> (1 2 3 4)

(do
  (def fifth (comp first rest rest rest rest))
  (fifth [1 2 3 4 5]))
=> 5
```

[section](#)[top](#)

compare

```
(compare x y)
```

Comparator. Returns -1, 0, or 1 when x is logically 'less than', 'equal to', or 'greater than' y. For list and vectors the longer sequence is always 'greater' regardless of its contents. For sets and maps only the size of the collection is compared.

```
(compare nil 0)
=> -1

(compare 0 nil)
=> 1

(compare 1 0)
=> 1

(compare 1 1)
=> 0

(compare 1M 2M)
=> -1

(compare 1 nil)
=> 1

(compare nil 1)
=> -1

(compare "aaa" "bbb")
=> -1

(compare [0 1 2] [0 1 2])
=> 0

(compare [0 1 2] [0 9 2])
=> -1
```

```
(compare [0 9 2] [0 1 2])
=> 1

(compare [1 2 3] [0 1 2 3])
=> -1

(compare [0 1 2] [3 4])
=> 1
```

[section](#)[top](#)

compare-and-set!

```
(compare-and-set! atom oldval newval)
```

Atomically sets the value of atom to newval if and only if the current value of the atom is identical to oldval. Returns true if set happened, else false.

```
(do
  (def counter (atom 2))
  (compare-and-set! counter 2 4)
  @counter)
=> 4
```

SEE ALSO

[atom](#)

Creates an atom with the initial value x.

[section](#)[top](#)

complement

```
(complement f)
```

Takes a fn f and returns a fn that takes the same arguments as f, has the same effects, if any, and returns the opposite truth value.

```
(complement even?)
=> anonymous-db4ee45c-ff9c-43d4-bc9b-18aeac5f57f8

(filter (complement even?) '(1 2 3 4))
=> (1 3)
```

[section](#)[top](#)

complete-on-timeout

```
(complete-on-timeout p value time time-unit)
```

Completes the promise with the given value if not otherwise completed before the given timeout.

```

(-> (promise (fn [] (sleep 100) "The quick brown fox"))
      (complete-on-timeout "The fox did not jump" 500 :milliseconds)
      (deref))
=> "The quick brown fox"

(-> (promise (fn [] (sleep 500) "The quick brown fox"))
      (complete-on-timeout "The fox did not jump" 100 :milliseconds)
      (deref))
=> "The fox did not jump"

(-> (promise (fn [] (sleep 500) "The quick brown fox"))
      (complete-on-timeout "The fox did not jump" 100 :milliseconds)
      (then-apply str/upper-case)
      (deref))
=> "THE FOX DID NOT JUMP"

(-> (promise (fn [] (sleep 50) 100))
      (complete-on-timeout 888 100 :milliseconds)
      (then-apply #(do (sleep 200) (* % 3)))
      (complete-on-timeout 999 220 :milliseconds)
      (deref))
=> 999

```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[then-compose](#)

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completest normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[section](#)

[top](#)

component/Component

Defines a protocol for components.

Definition:


```
(defprotocol Component
  (start [component] component)
  (stop [component] component))
```

Function `start` :

Begins operation of this component. Synchronous, does not return until the component is started. Returns an updated version of this component.

Function `stop` :

Ceases operation of this component. Synchronous, does not return until the component is stopped. Returns an updated version of this component.

[section](#)

[top](#)

component/dep

```
(dep c k)
```

Returns a dependency given by its key 'k' from the component 'c' dependencies.

```
(do
  (load-module :component ['component :as 'c])

  (deftype :server []
    c/Component
    (start [this] (println "Store: " (c/dep this :store)) this)
    (stop [this] this))

  (deftype :database []
    c/Component
    (start [this] this)
    (stop [this] this))

  (defn create-system []
    (-> (c/system-map
        "test"
        :server (server. )
        :store (database. ))
      (c/system-using {:server [:store]})))

  (-> (create-system)
    (c/start)
    (c/stop))

  nil)
Store: {:custom-type* :user/database}
=> nil
```

SEE ALSO

[component/deps](#)

Returns the dependencies of the component 'c' or nil if there aren't any dependencies.

[component/id](#)

Returns id of the component 'c'.

[section](#)

[top](#)

component/deps

```
(deps c)
```

Returns the dependencies of the component 'c' or `nil` if there aren't any dependencies.

```
(do
  (load-module :component ['component :as 'c])

  (deftype :server []
    c/Component
    (start [this] (println "Dependencies: " (c/deps this)) this)
    (stop [this] this))

  (deftype :database []
    c/Component
    (start [this] this)
    (stop [this] this))

  (defn create-system []
    (-> (c/system-map
        "test"
        :server (server. )
        :store (database. ))
      (c/system-using {:server [:store]})))

  (-> (create-system)
    (c/start)
    (c/stop))

  nil)
Dependencies:  {:store {:custom-type* :user/database} :component-info {:custom-type* :component/component-info :
id :server :system-name test :components {}}}
=> nil
```

SEE ALSO

[component/dep](#)

Returns a dependency given by its key 'k' from the component 'c' dependencies.

[component/id](#)

Returns id of the component 'c'.

[section](#)

[top](#)

component/id

```
(id c)
```

Returns id of the component 'c'.

```
(do
  (load-module :component ['component :as 'c])

  (deftype :server []
    c/Component
    (start [this] (println "ID: " (c/id this)) this)
    (stop [this] this))

  (defn create-system []
```

```

(-> (c/system-map
      "test"
      :server (server. ))
     (c/system-using {:server []})))

(-> (create-system)
     (c/start)
     (c/stop))

nil)
ID: :server
=> nil

```

SEE ALSO

[component/dep](#)

Returns a dependency given by its key 'k' from the component 'c' dependencies.

[component/deps](#)

Returns the dependencies of the component 'c' or nil if there aren't any dependencies.

[section](#) [top](#)

component/system-map

```
(system-map name keyval*)
```

Returns a system constructed of components given as key/value pairs. The 'key' is a `keyword` (the component's id) referencing the component given as 'value'.

The system has default implementations of the Lifecycle 'start' and 'stop' methods which recursively starts/stops all components in the system.

Note:

`system-map` just creates a raw system without any dependencies between the components. Use `system-using` after creating the system map to establish the dependencies.

```

(do
  (load-module :component ['component :as 'c])

  (deftype :server [port :long]
    c/Component
    (start [this] (println "server started") this)
    (stop [this] (println "server stopped") this))

  (deftype :database [user      :string
                     password :string]
    c/Component
    (start [this] (println "database started") this)
    (stop [this] (println "database stopped") this))

  (c/system-map
    "test"
    :server (server. 4600)
    :store (database. "foo" "123"))

  nil)

```

SEE ALSO

[component/system-using](#)

Associates a component dependency graph with the 'system' that has been created through a call to `system-map`. 'dependency-map' is a ...

component/system-using

```
(system-using system dependency-map)
```

Associates a component dependency graph with the 'system' that has been created through a call to `system-map`. 'dependency-map' is a map of keys to maps or vectors specifying the dependencies of the component at that key in the system.

Throws an exception if a component dependency circle is detected.

The system is started and stopped calling the lifecycle `start` or `stop` method on the system component.

Upon successfully starting a component the flag `{:started true}` is added to the component's meta data. It's up to the components lifecycle `start` method to decide what to do with multiple start requests. The lifecycle `start` method can for instance simply return the unaltered component if it has already been started.

Upon successfully stopping a component the flag `{:started false}` is added to the component's meta data. It's up to the components lifecycle `stop` method to decide what to do with multiple stop requests. The lifecycle `stop` method can for instance simply return the unaltered component if it has not been started or has already been stopped.

```
(do
  (load-module :component ['component :as 'c])

  (deftype :server [port :long]
    c/Component
    (start [this]
      (let [store1 (-> (c/dep this :store1) :name)
            store2 (-> (c/dep this :store2) :name)]
        (println "server started. using the stores" store1 "," store2))
      this)
    (stop [this]
      (println "server stopped")
      this))

  (deftype :database [name      :string
                     user       :string
                     password   :string]
    c/Component
    (start [this]
      (println "database" (:name this) "started")
      this)
    (stop [this]
      (println "database" (:name this) "stopped")
      this))

  (defn create-system []
    (-> (c/system-map
        "test"
        :server (server. 4600)
        :store1 (database. "store1" "foo" "123")
        :store2 (database. "store2" "foo" "123"))
        (c/system-using {:server [:store1 :store2]))))

  (defn start []
    (-> (create-system)
        (c/start)))

  (let [system (start)
        server (-> system :components :server)]
    ; access server component
    (println "Accessing the system...")
    (c/stop system))
```

```
nil)
database store1 started
database store2 started
server started. using the stores store1 , store2
Accessing the system...
server stopped
database store2 stopped
database store1 stopped
=> nil
```

SEE ALSO

[component/system-map](#)

Returns a system constructed of components given as key/value pairs. The 'key' is a keyword (the component's id) referencing the component ...

[section](#)

[top](#)

concat

```
(concat coll)
(concat coll & colls)
```

Returns a list of the concatenation of the elements in the supplied collections.

```
(concat [1 2])
=> (1 2)
```

```
(concat [1 2] [4 5 6])
=> (1 2 4 5 6)
```

```
(concat '(1 2))
=> (1 2)
```

```
(concat '(1 2) [4 5 6])
=> (1 2 4 5 6)
```

```
(concat {:a 1})
=> ([:a 1])
```

```
(concat {:a 1} {:b 2 :c 3})
=> ([:a 1] [:b 2] [:c 3])
```

```
(concat "abc")
=> (#\a #\b #\c)
```

```
(concat "abc" "def")
=> (#\a #\b #\c #\d #\e #\f)
```

SEE ALSO

[concatv](#)

Returns a vector of the concatenation of the elements in the supplied collections.

[into](#)

Returns a new coll consisting of to coll with all of the items of from coll conjoined.

[merge](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from ...

concatv

```
(concatv coll)
(concatv coll & colls)
```

Returns a vector of the concatenation of the elements in the supplied collections.

```
(concatv [1 2])
=> [1 2]
```

```
(concatv [1 2] [4 5 6])
=> [1 2 4 5 6]
```

```
(concatv '(1 2))
=> [1 2]
```

```
(concatv '(1 2) [4 5 6])
=> [1 2 4 5 6]
```

```
(concatv {:a 1})
=> [[:a 1]]
```

```
(concatv {:a 1} {:b 2 :c 3})
=> [[:a 1] [:b 2] [:c 3]]
```

```
(concatv "abc")
=> [#\a #\b #\c]
```

```
(concatv "abc" "def")
=> [#\a #\b #\c #\d #\e #\f]
```

SEE ALSO

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[into](#)

Returns a new coll consisting of to coll with all of the items of from coll conjoined.

[merge](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from ...

cond

```
(cond & clauses)
```

Takes a set of test/expr pairs. It evaluates each test one at a time. If a test returns logical true, cond evaluates and returns the value of the corresponding expr and doesn't evaluate any of the other tests or exprs. (cond) returns nil.

```
(let [n 5]
  (cond
    (< n 0) "negative"
    (> n 0) "positive"
    :else "zero"))
=> "positive"
```

SEE ALSO

[condp](#)

Takes a binary predicate, an expression, and a set of clauses.

[case](#)

Takes an expression and a set of clauses. Each clause takes the form of test-constant result-expr

[section](#)

[top](#)

cond->

(cond-> expr & clauses)

Takes an expression and a set of test/form pairs. Threads expr (via ->) through each form for which the corresponding test expression is true. Note that, unlike cond branching, `cond->` threading does not short circuit after the first true test expression.

It is useful in situations where you want selectively assoc, update, or dissoc something from a map.

```
(cond-> m
  (some-pred? q) (assoc :key :value))
```

```
(let [a 5]
  (cond-> []
    (odd? a) (conj (* a 2)) ;; => [10]
    (even? a) (conj (* a 3))))
=> [10]
```

SEE ALSO

[cond->>](#)

Takes an expression and a set of test/form pairs. Threads expr (via ->>) through each form for which the corresponding test expression ...

[section](#)

[top](#)

cond->>

(cond->> expr & clauses)

Takes an expression and a set of test/form pairs. Threads expr (via ->>) through each form for which the corresponding test expression is true. Note that, unlike cond branching, `cond->>` threading does not short circuit after the first true test expression.

```
(let [a 12]
  (cond->> 3
    (odd? a) (/ a)
    (even? a) (/ a))) ;; => 4
=> 4
```

SEE ALSO

[cond->](#)

Takes an expression and a set of test/form pairs. Threads expr (via ->) through each form for which the corresponding test expression ...

[section](#)

[top](#)

condp

(condp pred expr & clauses)

Takes a binary predicate, an expression, and a set of clauses.

Each clause can take the form of either:

test-expr result-expr

test-expr :>> result-fn

Note :>> is an ordinary keyword.

For each clause, (pred test-expr expr) is evaluated. If it returns logical true, the clause is a match. If a binary clause matches, the result-expr is returned, if a ternary clause matches, its result-fn, which must be a unary function, is called with the result of the predicate as its argument, the result of that call being the return value of condp. A single default expression can follow the clauses, and its value will be returned if no clause matches. If no default expression is provided and no clause matches, a VncException is thrown.

```
(condp some [1 2 3 4]
  #{0 6 7} :>> inc
  #{4 5 9} :>> dec
  #{1 2 3} :>> #(* % 10))
=> 3
```

```
(condp some [-10 -20 0 10]
  pos? 1
  neg? -1
  (constantly true) 0)
=> 1
```

SEE ALSO

[cond](#)

Takes a set of test/expr pairs. It evaluates each test one at a time. If a test returns logical true, cond evaluates and returns the ...

[case](#)

Takes an expression and a set of clauses. Each clause takes the form of test-constant result-expr

[section](#)

[top](#)

config/build

(build & parts)

Merges given configuration parts and returns it as a map.

Configuration parts:

- JSON classpath resource file
- JSON file
- Environment variables
- System properties

Example:


```

(do
  (load-module :config)

  (def cfg (config/build
    (config/env "java")
    (config/env-var "SERVER_PORT" [:http :port] "8080")))

  (println "home:" (-> cfg :11 :zulu :home))
  ; => home: /Library/Java/JavaVirtualMachines/zulu-11.jdk/Contents/Home

  (println "port:" (-> cfg :http :port)))
  ; => port: 8080

```

```

;; -----
;; Example I) Configuration builder
(do
  (load-module :config ['config :as 'cfg])

  (cfg/build
    (cfg/resource "config-defaults.json" :key-fn keyword)
    (cfg/file "./config-local.json" :key-fn keyword)
    (cfg/env-var "SERVER_PORT" [:http :port])
    (cfg/env-var "SERVER_THREADS" [:http :threads])
    (cfg/property-var "MASTER_PWD" [:app :master-pwd])))

;; -----
;; Example II) Using configurations with the component module
(do
  (load-module :config ['config :as 'cfg])
  (load-module :component ['component :as 'cmp])

  ;; define the server component
  (deftype :server []
    cmp/Component
    (start [this]
      (let [config (cmp/dep this :config)
            port (get-in config [:server :port])]
        (println (cmp/id this) "started at port" port)
        this))
    (stop [this]
      (println (cmp/id this) "stopped")
      this))

  ;; note that the configuration is a plain vanilla Venice map and
  ;; does not implement the protocol 'Component'
  (defn create-system []
    (-> (cmp/system-map
      "test"
      :config (cfg/build
        (cfg/env-var "SERVER_PORT" [:server :port] "8800"))
      :server (server. ))
      (cmp/system-using
        {:server [:config]})))

  (-> (create-system)
    (cmp/start)
    (cmp/stop))

  nil)

```

SEE ALSO

[config/file](#)

Reads a JSON configuration part from given file `f`.

[config/resource](#)

Reads a JSON configuration part from given path in classpath.

[config/env-var](#)

Reads a configuration value from an environment variable and associates it to the given path in a map.

[config/property-var](#)

Reads a configuration value from a system property and associates it to the given path in a map.

[config/env](#)

Reads configuration part from environment variables, filtered by a prefix. `nil` may be passed as prefix to get env vars.

[config/properties](#)

Reads configuration part from system properties, filtered by a prefix. `nil` may be passed as prefix to get property vars.

[section](#)

[top](#)

config/env

(`env prefix`)

Reads configuration part from environment variables, filtered by a prefix. `nil` may be passed as prefix to get env vars.

The reader splits the environment variable names on the underscores to build a map.

```
(base) $ env | grep JAVA_
JAVA_11_OPENJDK_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home
JAVA_11_ZULU_HOME=/Library/Java/JavaVirtualMachines/zulu-11.jdk/Contents/Home
JAVA_11_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home
JAVA_8_ZULU_HOME=/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home
JAVA_8_OPENJDK_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home
JAVA_8_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home
JAVA_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home

venice> (config/env "java")
=> {
  :11 {
    :zulu { :home "/Library/Java/JavaVirtualMachines/zulu-11.jdk/Contents/Home" }
    :openjdk { :home "/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home" }
    :home "/Library/Java/JavaVirtualMachines/adoptopenjdk-11.jdk/Contents/Home"
  }

  :8 {
    :zulu { :home "/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home" }
    :openjdk { :home "/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home" }
    :home "/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home"
  }

  :home "/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home"
}
```

(`config/env "DATABASE_"`)

SEE ALSO

[config/env-var](#)

Reads a configuration value from an environment variable and associates it to the given path in a map.

[config/properties](#)

Reads configuration part from system properties, filtered by a prefix. `nil` may be passed as prefix to get property vars.

[config/build](#)

Merges given configuration parts and returns it as a map.

[section](#)

[top](#)

config/env-var

```
(env-var name path)
(env-var name path default-val)
```

Reads a configuration value from an environment variable and associates it to the given path in a map.

```
(config/env-var "JAVA_HOME" [:java-home])
=> {:java-home "/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home"}
```

```
(config/env-var "SERVER_PORT" [:http :port])
=> nil
```

```
(config/env-var "SERVER_PORT" [:http :port] "8080")
=> {:http {:port "8080"}}
```

SEE ALSO

[config/property-var](#)

Reads a configuration value from a system property and associates it to the given path in a map.

[config/env](#)

Reads configuration part from environment variables, filtered by a prefix. nil may be passed as prefix to get env vars.

[config/build](#)

Merges given configuration parts and returns it as a map.

[section](#)

[top](#)

config/file

```
(file f)
(file f reader-opts)
```

Reads a JSON configuration part from given file f.

f may be a:

- string file path, e.g. "/temp/foo.json"
- `java.io.File`, e.g. `(io/file "/temp/foo.json")`
- `java.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `java.net.URI`

The optional 'reader-opts' are defined by `json/read-str`.

E.g.: `:key-fn keyword` will convert all config keys to keywords

```
(config/file "/foo/app/config-production.json" :key-fn keyword)
```

```
(do
  (def cfg-json ""
    { "db" : {
      "classname" : "com.mysql.jdbc.Driver",
      "subprotocol" : "mysql",
      "subname" : "//127.0.0.1:3306/test",
      "user" : "test",
      "password" : "123"
    }
  }
  ""))

(-> (io/buffered-reader cfg-json)
  (config/file :key-fn keyword)))
```

SEE ALSO

[config/resource](#)

Reads a JSON configuration part from given path in classpath.

[config/build](#)

Merges given configuration parts and returns it as a map.

[json/read-str](#)

Reads a JSON string and returns it as a Venice datatype.

[section](#)

[top](#)

config/properties

(properties prefix)

Reads configuration part from system properties, filtered by a prefix. `nil` may passed as prefix to get property vars.

The reader splits the property names on the underscores to build a map.

```
(config/properties "DATABASE_")
```

SEE ALSO

[config/property-var](#)

Reads a configuration value from an system property and associates it to the given path in a map.

[config/build](#)

Merges given configuration parts and returns it as a map.

[section](#)

[top](#)

config/property-var

(property-var name path)

(property-var name path default-val)

Reads a configuration value from an system property and associates it to the given path in a map.

```
(config/property-var "java.vendor" [:java :vendor])
=> {:java {:vendor "Azul Systems, Inc."}}
```

```
(config/property-var "java.version" [:java :version])
=> {:java {:version "1.8.0_462"}}

(config/property-var "SERVER_PORT" [:http :port])
=> nil

(config/property-var "SERVER_PORT" [:http :port] "8080")
=> {:http {:port "8080"}}
```

SEE ALSO

[config/env-var](#)

Reads a configuration value from an environment variable and associates it to the given path in a map.

[config/properties](#)

Reads configuration part from system properties, filtered by a prefix. nil may be passed as prefix to get property vars.

[config/build](#)

Merges given configuration parts and returns it as a map.

[section](#)

[top](#)

config/resource

```
(resource path)
(resource path reader-opts)
```

Reads a JSON configuration part from given path in classpath.

The optional 'reader-opts' are defined by `json/read-str`.

E.g.: `:key-fn keyword` will convert all config keys to keywords

```
(config/resource "com/github/jlangch/venice/examples/database-config.json"
  :key-fn keyword)
=> {:db {:classname "com.mysql.jdbc.Driver" :subname "//127.0.0.1:3306/test" :user "test" :subprotocol "mysql" :
password "123"}}
```

SEE ALSO

[config/file](#)

Reads a JSON configuration part from given file f.

[config/build](#)

Merges given configuration parts and returns it as a map.

[json/read-str](#)

Reads a JSON string and returns it as a Venice datatype.

[section](#)

[top](#)

conj

```
(conj)
(conj x)
(conj coll x)
(conj coll x & xs)
```

Returns a new collection with the x, xs 'added'. `(conj nil item)` returns `(item)` and `(conj item)` returns `item`.

For ordered collections like list, vectors and ordered sets/maps the value is added at the end. For all other collections the position is undefined.

```
(conj [1 2 3] 4)
=> [1 2 3 4]

(conj [1 2 3] 4 5)
=> [1 2 3 4 5]

(conj [1 2 3] [4 5])
=> [1 2 3 [4 5]]

(conj '(1 2 3) 4)
=> (1 2 3 4)

(conj '(1 2 3) 4 5)
=> (1 2 3 4 5)

(conj '(1 2 3) '(4 5))
=> (1 2 3 (4 5))

(conj (set 1 2 3) 4)
=> #{1 2 3 4}

(conj {:a 1 :b 2} [:c 3])
=> {:a 1 :b 2 :c 3}

(conj {:a 1 :b 2} {:c 3})
=> {:a 1 :b 2 :c 3}

(conj {:a 1 :b 2} (map-entry :c 3))
=> {:a 1 :b 2 :c 3}

(conj)
=> []

(conj 4)
=> 4
```

SEE ALSO

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[into](#)

Returns a new coll consisting of to coll with all of the items of from coll conjoined.

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[list*](#)

Creates a new list containing the items prepended to the rest, the last of which will be treated as a collection.

[vector*](#)

Creates a new vector containing the items prepended to the rest, the last of which will be treated as a collection.

[section](#)

[top](#)

conj!

```
(conj!)
(conj! x)
(conj! coll x)
(conj! coll x & xs)
```

Returns a new mutable collection with the `x`, `xs` 'added'. `(conj! nil item)` returns `(item)` and `(conj! item)` returns `item`.

For mutable ordered collections like lists the value is added at the end. For all other mutable collections the position is undefined.

```
(conj! (mutable-list 1 2 3) 4)
=> (1 2 3 4)

(conj! (mutable-list 1 2 3) 4 5)
=> (1 2 3 4 5)

(conj! (mutable-list 1 2 3) '(4 5))
=> (1 2 3 (4 5))

(conj! (mutable-set 1 2 3) 4)
=> #{1 2 3 4}

(conj! (mutable-map :a 1 :b 2) [:c 3])
=> {:a 1 :b 2 :c 3}

(conj! (mutable-map :a 1 :b 2) {:c 3})
=> {:a 1 :b 2 :c 3}

(conj! (mutable-map :a 1 :b 2) (map-entry :c 3))
=> {:a 1 :b 2 :c 3}

(conj! (stack) 1 2 3)
=> (3 2 1)

(conj! (queue) 1 2 3)
=> (1 2 3)

(conj!)
=> ()

(conj! 4)
=> 4
```

[section](#)

[top](#)

cons

```
(cons x coll)
```

Returns a new collection where `x` is the first element and `coll` is the rest.

For ordered collections like list, vectors and ordered sets/maps the value is added at the beginning. For all other collections the position is undefined.

```
(cons 1 '(2 3 4 5 6))
=> (1 2 3 4 5 6)
```

```

(cons 1 nil)
=> (1)

(cons [1 2] [4 5 6])
=> [[1 2] 4 5 6]

(cons 3 (set 1 2))
=> #{1 2 3}

(cons {:c 3} {:a 1 :b 2})
=> {:a 1 :b 2 :c 3}

(cons (map-entry :c 3) {:a 1 :b 2})
=> {:a 1 :b 2 :c 3}

; cons a value to a lazy sequence
(->> (cons -1 (lazy-seq 0 #(+ % 1)))
      (take 5)
      (doall))
=> (-1 0 1 2 3)

; recursive lazy sequence (fibonacci example)
(do
  (defn fib
    ([ ] (fib 1 1))
    ([a b] (cons a (fn [] (fib b (+ a b))))))

    (doall (take 6 (fib))))
=> (1 1 2 3 5 8)

```

SEE ALSO

[conj](#)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item) and (conj item) returns item.

[list*](#)

Creates a new list containing the items prepended to the rest, the last of which will be treated as a collection.

[vector*](#)

Creates a new vector containing the items prepended to the rest, the last of which will be treated as a collection.

[section](#)
[top](#)

cons!

```
(cons! x coll)
```

Adds x to the mutable collection coll.

For mutable ordered collections like lists the value is added at the beginning. For all other mutable collections the position is undefined.

```

(cons! 1 (mutable-list 2 3))
=> (1 2 3)

(cons! 3 (mutable-set 1 2))
=> #{1 2 3}

(cons! {:c 3} (mutable-map :a 1 :b 2))
=> {:a 1 :b 2 :c 3}

```



```
(cons! (map-entry :c 3) (mutable-map :a 1 :b 2))  
=> {:a 1 :b 2 :c 3}
```

```
(cons! 1 (stack))  
=> (1)
```

```
(cons! 1 (queue))  
=> (1)
```

[section](#)[top](#)

constantly

```
(constantly x)
```

Returns a function that takes any number of arguments and returns always the value x.

```
(do  
  (def fix (constantly 10))  
  (fix 1 2 3)  
  (fix 1)  
  (fix ))  
=> 10
```

SEE ALSO

[repeat](#)

Returns a lazy sequence of x values or a collection with the value x repeated n times.

[repeatedly](#)

Takes a function of no args, presumably with side effects, and returns a collection of n calls to it

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[section](#)[top](#)

contains?

```
(contains? coll key)
```

Returns true if key is present in the given collection, otherwise returns false.

Note: To test if a value is in a vector or list use `any?`

```
(contains? #{:a :b} :a)  
=> true
```

```
(contains? {:a 1 :b 2} :a)  
=> true
```

```
(contains? [10 11 12] 1)  
=> true
```

```
(contains? [10 11 12] 5)
=> false
```

```
(contains? "abc" 1)
=> true
```

```
(contains? "abc" 5)
=> false
```

SEE ALSO

[not-contains?](#)

Returns true if key is not present in the given collection, otherwise returns false.

[any?](#)

Returns true if the predicate is true for at least one collection item, false otherwise.

[section](#)[top](#)

count

```
(count coll)
```

Returns the number of items in the collection. `(count nil)` returns 0. Also works on strings, and Java Collections

```
(count {:a 1 :b 2})
=> 2
```

```
(count [1 2])
=> 2
```

```
(count "abc")
=> 3
```

[section](#)[top](#)

cpus

```
(cpus)
```

Returns the number of available processors or number of hyperthreads if the CPU supports hyperthreads.

```
(cpus)
=> 8
```

[section](#)[top](#)

cron/schedule-at

```
(schedule-at-round-times-in-day fn sync-period schedule-at)
```

Creates and executes a one-shot scheduled task that becomes enabled at a given time.

This scheduled task is not prone to clock shifts.

Returns a future. `(deref f)`, `(future? f)`, `(cancel f)`, and `(done? f)` will work on the returned future.

This function is built on the [CronScheduler](#) project.

```
(let [sync-period (. :java.time.Duration :ofMinutes 10)
      at          (time/plus (time/local-date-time) :seconds 2)
      task        (fn [] (println "Task:" (time/local-date-time)) 100)
      sched       (cron/schedule-at task sync-period at)]
  (deref sched))
```

SEE ALSO

[cron/schedule-at-fixed-rate](#)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period.

[cron/schedule-at-round-times-in-day](#)

Submits a periodic task that becomes enabled at round clock times within a day, with the given period.

[section](#)

[top](#)

cron/schedule-at-fixed-rate

```
(schedule-at-fixed-rate fn sync-period initial-delay period time-unit)
(schedule-at-fixed-rate fn sync-period initial-delay period time-unit skipping-to-latest)
```

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period.

This scheduled task is not prone to clock shifts.

Returns a future. `(deref f)`, `(future? f)`, `(cancel f)`, and `(done? f)` will work on the returned future.

This function is built on the [CronScheduler](#) project.

```
(let [sync-period (. :java.time.Duration :ofMinutes 10)
      task        (fn [] (println "Task:" (time/local-date-time)))
      sched       (cron/schedule-at-fixed-rate task sync-period 1 2 :seconds)]
  (sleep 16 :seconds)
  (cancel sched))
```

SEE ALSO

[cron/schedule-at-round-times-in-day](#)

Submits a periodic task that becomes enabled at round clock times within a day, with the given period.

[cron/schedule-at](#)

Creates and executes a one-shot scheduled task that becomes enabled at a given time.

[section](#)

[top](#)

cron/schedule-at-round-times-in-day

```
(schedule-at-round-times-in-day fn sync-period schedule-period)
(schedule-at-round-times-in-day fn sync-period schedule-period skipping-to-latest)
```

Submits a periodic task that becomes enabled at round clock times within a day, with the given period.

This scheduled task is not prone to clock shifts.

Returns a future. `(deref f)`, `(future? f)`, `(cancel f)`, and `(done? f)` will work on the returned future.

This function is built on the [CronScheduler](#) project.

```
(let [sync-period      (. :java.time.Duration :ofMinutes 10)
      schedule-period  (. :java.time.Duration :ofHours 4)
      task              (fn [] (println "Task:" (time/local-date-time)))
      sched             (cron/schedule-at-round-times-in-day task sync-period schedule-period)]
  (sleep 24 :hours)
  (cancel sched))
```

SEE ALSO

[cron/schedule-at-fixed-rate](#)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period.

[cron/schedule-at](#)

Creates and executes a one-shot scheduled task that becomes enabled at a given time.

[section](#)

[top](#)

crypt/add-bouncy-castle-provider

```
(crypt/add-bouncy-castle-provider)
```

Adds the BouncyCastle provider to the Java security.

```
(do
  (load-module :crypt)
  (crypt/add-bouncy-castle-provider))
```

SEE ALSO

[crypt/provider?](#)

Returns true if the Java security provider with name exists, else false.

[section](#)

[top](#)

crypt/ciphers

```
(crypt/ciphers)
(crypt/ciphers opt)
```

Returns a list of the ciphers the Java VM supports

Argument opt

:default returns the names of the cipher suites which are enabled by default.

:available returns the names of the cipher suites which could be enabled for use on an SSL connection created by this `SSLServerSocketFactory`.

```
(do
  (load-module :crypt)
  (crypt/ciphers :default))
```

```
(do
  (load-module :crypt)
  (crypt/ciphers :available))

(do
  (load-module :crypt)
  (docoll println (crypt/ciphers :available)))
```

section top

crypt/encryptor-aes-256-gcm

```
(crypt/encryptor-aes-256-gcm passphrase & opts)
```

Returns an encryptor function for AES 256 GCM encrypting/decrypting binary buffers or files.

Options:

:key-salt arr	An optional salt to generate the key. E.g.: (bytebuf [43 195 99 118 231 225 142 76 132 194 129 237 158 12 12 203])
:key-iterations n	An optional iteration count to generate the key. Defaults to 3000.
:efficient-mem-use b	If <code>true</code> efficiently use memory in encryption (less memory allocation). Defaults to <code>false</code> .

IV is random and unique for every call of the encryptor function.

While encrypting data the random *IV* is written to the start of the encrypted data and read before decrypting the data

```
AES256-GCM
AES/GCM/NoPadding
+-----+
|      iv  (12)      |
+-----+
|      data  (n)      |
+-----+
```

```
;; File
(do
  (load-module :crypt)
  (load-module :hexdump)
  (let [file-in      (io/temp-file "test-", ".data")
        file-encoded (io/temp-file "test-", ".data.enc")
        file-decoded (io/temp-file "test-", ".data.dec")
        encryptor    (crypt/encryptor-aes-256-gcm "secret")]
    (io/delete-file-on-exit file-in file-encoded file-decoded)
    (io/spit file-in "1234567890")
    (encryptor :encrypt file-in file-encoded true)
    (encryptor :decrypt file-encoded file-decoded true)
    (println (hexdump/dump (io/slurp file-encoded :binary true)))))

;; Byte buffer
(do
  (load-module :crypt)
  (let [buf-in      (bytebuf-from-string "1234567890")
        encryptor    (crypt/encryptor-aes-256-gcm "secret")
        buf-encoded  (encryptor :encrypt buf-in)
        buf-decoded  (encryptor :decrypt buf-encoded)]
    (println (bytebuf-to-string buf-decoded))))
```

```
;; Text (Base64 encoded as either :Standard or :UrlSafe)
(do
  (load-module :crypt)
  (let [text      "1234567890"
        encryptor (crypt/encryptor-aes-256-gcm "secret")
        encoded    (encryptor :encrypt text :Standard)
        decoded    (encryptor :decrypt encoded :Standard)]
    (println decoded)))
```

SEE ALSO

[crypt/encryptor-chacha20-bouncycastle](#)

Returns an encryptor function for ChaCha20 BouncyCastle encrypting/decrypting binary buffers or files.

[crypt/encryptor-chacha20](#)

Returns an encryptor function for ChaCha20 encrypting/decrypting binary buffers or files.

[section](#)

[top](#)

crypt/encryptor-chacha20

```
(crypt/encryptor-chacha20 passphrase & opts)
```

Returns an encryptor function for ChaCha20 encrypting/decrypting binary buffers or files.

Requires Java11+.

Options:

:key-salt arr	An optional salt to generate the key. E.g.: (bytebuf [43 195 99 118 231 225 142 76 132 194 129 237 158 12 12 203])
:key-iterations n	An optional iteration count to generate the key. Defaults to 3000.

Warning: files encrypted with ChaCha20 BouncyCastle cannot be by ChaCha20 from the Java VM (and vice versa) due to different initial counter handling and IV sizes

The ChaCha family of ciphers are an order of magnitude more efficient on servers that do not provide hardware acceleration. Apple Silicon does not seem to have AES hardware acceleration probably due to its RISC nature.

IV, *Nonce* and/or *Counter* are random and unique for every call of the encryptor function.

While encrypting a file the *IV*, *Nonce* and *Counter* are written to the start of the encrypted data and read before decrypting the data:

```

ChaCha20
+-----+
|      iv (12)      |
+-----+
|     nonce (12)     |
+-----+
|    counter (4)     |
+-----+
|     data (n)       |
+-----+
```

```
;; File
(do
  (load-module :crypt)
  (load-module :hexdump)
  (let [file-in      (io/temp-file "test-", ".data")
        file-encoded (io/temp-file "test-", ".data.enc")
        file-decoded (io/temp-file "test-", ".data.dec")
        encryptor    (crypt/encryptor-chacha20 "secret")]
```

```

(io/delete-file-on-exit file-in file-encoded file-decoded)
(io/spit file-in "1234567890")
(encryptor :encrypt file-in file-encoded true)
(encryptor :decrypt file-encoded file-decoded true)
(println (hexdump/dump (io/slurp file-encoded :binary true))))

;; Byte buffer
(do
  (load-module :crypt)
  (let [buf-in (bytebuf-from-string "1234567890")
        encryptor (crypt/encryptor-chacha20 "secret")
        buf-encoded (encryptor :encrypt buf-in)
        buf-decoded (encryptor :decrypt buf-encoded)]
    (println (bytebuf-to-string buf-decoded))))

;; Text (Base64 ecoded as either :Standard or :UrlSafe)
(do
  (load-module :crypt)
  (let [text "1234567890"
        encryptor (crypt/encryptor-chacha20 "secret")
        encoded (encryptor :encrypt text :Standard)
        decoded (encryptor :decrypt encoded :Standard)]
    (println decoded)))

```

SEE ALSO

[crypt/encryptor-aes-256-gcm](#)

Returns an encryptor function for AES 256 GCM encrypting/decrypting binary buffers or files.

[crypt/encryptor-chacha20-bouncycastle](#)

Returns an encryptor function for ChaCha20 BouncyCastle encrypting/decrypting binary buffers or files.

[section](#)

[top](#)

crypt/encryptor-chacha20-bouncycastle

```
(crypt/encryptor-chacha20-bouncycastle passphrase & opts)
```

Returns an encryptor function for ChaCha20 BouncyCastle encrypting/decrypting binary buffers or files.

Requires the BouncyCastle libraries on the classpath.

Options:

:key-salt arr	An optional salt to generate the key. E.g.: (bytebuf [43 195 99 118 231 225 142 76 132 194 129 237 158 12 12 203])
:key-iterations n	An optional iteration count to generate the key. Defaults to 3000.

Warning: files encrypted with ChaCha20 BouncyCastle cannot be by ChaCha20 from the Java VM (and vice versa) due to different initial counter handling and IV sizes

The ChaCha family of ciphers are an order of magnitude more efficient on servers that do not provide hardware acceleration. Apple Silicon does not seem to have AES hardware acceleration probably due to its RISC nature.

IV, *Nonce* and *Counter* are random and unique for every call of encryptor function.

While encrypting a file the *IV*, *Nonce* and *Counter* are written to the start of the encrypted data and read before decrypting the data:

```

ChaCha20 Bouncy Castle
+-----+
|      iv (8)      |
+-----+
|    nonce (12)    |

```

```
+-----+
|   counter (4)   |
+-----+
|   data (n)      |
+-----+
```

```
;; File
(do
  (load-module :crypt)
  (load-module :hexdump)
  (let [file-in      (io/temp-file "test-", ".data")
        file-encoded (io/temp-file "test-", ".data.enc")
        file-decoded (io/temp-file "test-", ".data.dec")
        encryptor    (crypt/encryptor-chacha20-bouncycastle "secret")]
    (io/delete-file-on-exit file-in file-encoded file-decoded)
    (io/spit file-in "1234567890")
    (encryptor :encrypt file-in file-encoded true)
    (encryptor :decrypt file-encoded file-decoded true)
    (println (hexdump/dump (io/slurp file-encoded :binary true)))))

;; Byte buffer
(do
  (load-module :crypt)
  (let [buf-in      (bytebuf-from-string "1234567890")
        encryptor   (crypt/encryptor-chacha20-bouncycastle "secret")
        buf-encoded (encryptor :encrypt buf-in)
        buf-decoded (encryptor :decrypt buf-encoded)]
    (println (bytebuf-to-string buf-decoded))))

;; Text
;; - encrypt text to Base64 encoded data
;; - decrypt Base64 encoded data to text
;; Supports Base64 :Standard (RFC4648) or :UrlSafe (RFC4648_URLSAFE) format
(do
  (load-module :crypt)
  (let [text      "1234567890"
        encryptor (crypt/encryptor-chacha20-bouncycastle "secret")
        encoded   (encryptor :encrypt text :Standard)
        decoded   (encryptor :decrypt encoded :Standard)]
    (println decoded)))
```

SEE ALSO

[crypt/encryptor-aes-256-gcm](#)

Returns an encryptor function for AES 256 GCM encrypting/decrypting binary buffers or files.

[crypt/encryptor-chacha20](#)

Returns an encryptor function for ChaCha20 encrypting/decrypting binary buffers or files.

[crypt/add-bouncy-castle-provider](#)

Adds the BouncyCastle provider to the Java security.

[crypt/provider?](#)

Returns true if the Java security provider with name exists, else false.

[section](#)

[top](#)

crypt/hash-file

(crypt/hash-file algorithm salt file)

Computes a hash for a file. The hash is used together with the function `crypt/verify-file-hash` to detect file modifications.

Returns the hash Base64 encoded.

The functions uses the fast MD5 hash algorithm.

The arg 'file' may be a:

- string file path, e.g: `"/temp/foo.json"`
- `bytebuf`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.io.InputStream`

Supported hash algorithms:

- MD5 (default)
- SHA-1
- SHA-512

MD5 is the fastest hash algorithm and precise enough to detect file changes.

```
(do
  (load-module :crypt)
  (let [file (io/temp-file "test-", ".data")
        data (bytebuf-allocate-random 1000)]
    (io/delete-file-on-exit file)
    (io/spit file data)
    (crypt/hash-file "SHA-256" "-salt-" file)))
```

SEE ALSO

[crypt/verify-file-hash](#)

Verifies a file against a hash (Base64 encoded). Returns true if the file's actual hash is equal to the given hash otherwise false.

[section](#)

[top](#)

crypt/max-key-size

`(crypt/max-key-size algorithm)`

Returns the max allowed key size

```
(do
  (load-module :crypt)
  (crypt/max-key-size "AES"))
```

[section](#)

[top](#)

crypt/md5-hash

`(crypt/md5-hash data)`
`(crypt/md5-hash data salt)`

Hashes a string or a bytebuf using MD5 with an optional salt.

Note: MD5 is not safe any more use PBKDF2 to hash passwords!

```
(do
  (load-module :crypt)
  (-> (crypt/md5-hash "hello world")
    (str/bytebuf-to-hex :upper)))
=> "5EB63BBE01EEED093CB22BB8F5ACDC3"

(do
  (load-module :crypt)
  (-> (crypt/md5-hash "hello world" "-salt-")
    (str/bytebuf-to-hex :upper)))
=> "C40C4EAC3C1B87B6877E21FEBA087D0A"

(do
  (load-module :crypt)
  (-> (crypt/md5-hash "hello world")
    (str/encode-base64)))
=> "XrY7u+Ae7tCTyyK7j1rNww=="
```

SEE ALSO

[crypt/sha1-hash](#)

Hashes a string or a bytebuf using SHA1 with an optional salt.

[crypt/sha512-hash](#)

Hashes a string or a bytebuf using SHA512 with an optional salt.

[crypt/pbkdf2-hash](#)

Hashes a string using PBKDF2. iterations defaults to 1000, key-length defaults to 256.

[section](#)

[top](#)

crypt/pbkdf2-hash

```
(crypt/pbkdf2-hash data salt)
(crypt/pbkdf2-hash data salt iterations key-length)
```

Hashes a string using PBKDF2. iterations defaults to 1000, key-length defaults to 256.

```
(do
  (load-module :crypt)
  (-> (crypt/pbkdf2-hash "hello world" "-salt-")
    (str/bytebuf-to-hex :upper)))
=> "54F2B4411E8817C2A0743B2A7DD7EAE5AA3F748D1DDDC00766380914AFFE995"

(do
  (load-module :crypt)
  (-> (crypt/pbkdf2-hash "hello world" "-salt-" 1000 256)
    (str/bytebuf-to-hex :upper)))
=> "54F2B4411E8817C2A0743B2A7DD7EAE5AA3F748D1DDDC00766380914AFFE995"

(do
  (load-module :crypt)
  (-> (crypt/pbkdf2-hash "hello world" "-salt-")
    (str/encode-base64)))
=> "VPK0QR6IF8KgDsqdfq5ao/dI0d3c4Adm0AkUr/6ZU="
```

SEE ALSO

[crypt/md5-hash](#)

Hashes a string or a bytebuf using MD5 with an optional salt.

[crypt/sha1-hash](#)

Hashes a string or a bytebuf using SHA1 with an optional salt.

[crypt/sha512-hash](#)

Hashes a string or a bytebuf using SHA512 with an optional salt.

[section](#)[top](#)

crypt/provider?

```
(crypt/provider? name)
```

Returns true if the Java security provider with name exists, else false.

```
(do
  (load-module :crypt)

  ;; BouncyCastle provider available?
  (crypt/provider? "BC"))
```

SEE ALSO

[crypt/add-bouncy-castle-provider](#)

Adds the BouncyCastle provider to the Java security.

[section](#)[top](#)

crypt/sha1-hash

```
(crypt/sha1-hash data)
(crypt/sha1-hash data salt)
```

Hashes a string or a bytebuf using SHA1 with an optional salt.

```
(do
  (load-module :crypt)
  (-> (crypt/sha1-hash "hello world")
    (str/bytebuf-to-hex :upper)))
=> "2AAE6C35C94FCFB415DBE95F408B9CE91EE846ED"

(do
  (load-module :crypt)
  (-> (crypt/sha1-hash "hello world" "-salt-")
    (str/bytebuf-to-hex :upper)))
=> "90AECEDB9423CC9BC5BB7CBAFB88380BE5745B3D"

(do
  (load-module :crypt)
  (-> (crypt/sha1-hash "hello world")
    (str/encode-base64)))
=> "Kq5sNc1Pz7QV2+lfQIuc6R7oRu0="
```

SEE ALSO

[crypt/md5-hash](#)

Hashes a string or a bytebuf using MD5 with an optional salt.

[crypt/sha512-hash](#)

Hashes a string or a bytebuf using SHA512 with an optional salt.

[crypt/pbkdf2-hash](#)

Hashes a string using PBKDF2. iterations defaults to 1000, key-length defaults to 256.

[section](#)[top](#)

crypt/sha512-hash

```
(crypt/sha512-hash data)
(crypt/sha512-hash data salt)
```

Hashes a string or a bytebuf using SHA512 with an optional salt.

```
(do
  (load-module :crypt)
  (let [s (-> (crypt/sha512-hash "hello world")
              (str/bytebuf-to-hex :upper))]
    (str/truncate s 64 "... " :middle)))
=> "309ECC489C12D6EB4CC40F50C902F2B...30E81F605DCF7DC5542E93AE9CD76F"

(do
  (load-module :crypt)
  (let [s (-> (crypt/sha512-hash "hello world" "-salt-")
              (str/bytebuf-to-hex :upper))]
    (str/truncate s 64 "... " :middle)))
=> "316EBB70239D9480E91089D5D5BC642...095F186B19FC33C93D60282F3314A2"

(do
  (load-module :crypt)
  (let [s (-> (crypt/sha512-hash "hello world")
              (str/encode-base64))]
    (str/truncate s 64 "... " :middle)))
=> "MJ7MSJwS1utMxA9QyQLytNDtd+5RGnx...VbRbDP2DDoH2Bdz33FVC6TrpzXbw=="
```

SEE ALSO

[crypt/md5-hash](#)

Hashes a string or a bytebuf using MD5 with an optional salt.

[crypt/sha1-hash](#)

Hashes a string or a bytebuf using SHA1 with an optional salt.

[crypt/pbkdf2-hash](#)

Hashes a string using PBKDF2. iterations defaults to 1000, key-length defaults to 256.

[section](#)[top](#)

crypt/verify-file-hash

```
(crypt/verify-file-hash algorithm salt file hash)
```

Verifies a file against a hash (Base64 encoded). Returns true if the file's actual hash is equal to the given hash otherwise false.

The arg 'file' may be a:

- string file path, e.g: "/temp/foo.json"

- `bytebuf`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.io.InputStream`

Supported hash algorithms:

- MD5
- SHA-1
- SHA-512

Warning: The MD5 hash function's security is considered to be severely compromised. Collisions can be found within seconds, and they can be used for malicious purposes.

```
(do
  (load-module :crypt)
  (let [file (io/temp-file "test-", ".data")
        data (bytebuf-allocate-random 1000)
        salt "salt"]
    (io/delete-file-on-exit file)
    (io/spit file data)
    (let [hash (crypt/hash-file "SHA-256" "-salt-" file)]
      (crypt/verify-file-hash "SHA-256" "-salt-" file hash))))
```

SEE ALSO

[crypt/hash-file](#)

Computes a hash for a file. The hash is used together with the function `crypt/verify-file-hash` to detect file modifications.

[section](#)

[top](#)

csv/read

`(csv/read source & options)`

Reads CSV-data from a source.

The source may be a:

- `string`
- `bytebuf`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.Path`, `
- `java.io.InputStream`
- `java.io.Reader`

Options:

`:encoding enc` used when reading from a binary data source e.g `:encoding :utf-8`, defaults to `:utf-8`
`:separator val` e.g. `","`, defaults to a comma
`:quote val` e.g. `""`, defaults to a double quote

```
(csv/read ""1,"ab",false"")
=> (("1" "ab" "false"))

(csv/read "1|||'ab'|false" :separator "|" :quote "'")
=> (("1" nil nil "ab" "false"))
```

csv/write

(csv/write sink records & options)

Spits data to a sink in CSV format.

The sink may be a:

- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.Path`
- `java.io.OutputStream`
- `java.io.Writer`

Options:

:separator val e.g. ",", defaults to a comma

:quote val e.g. "\"", defaults to a double quote

:newline val :lf (default) or :cr+lf

:encoding enc used when writing to a binary data sink. e.g :encoding :utf-8, defaults to :utf-8

```
(csv/write (io/file "test.csv") [[1 "AC" false] [2 "WS" true]])
```

csv/write-str

(csv/write-str records & options)

Writes data to a string in CSV format.

All fields containing a quote char, a separator char or a newline are quoted.

Options:

:separator val e.g. ",", defaults to a comma

:quote val e.g. "\"", defaults to a double quote

:newline val :lf (default) or :cr+lf

```
(csv/write-str [[1 "AC" false] [2 "WS" true]])
```

```
=> "1,AC,false\n2,WS,true"
```

```
(csv/write-str [[1 "AC" false] [2 "WS, '-1'" true]]
```

```
  :quote "\""
```

```
  :separator ","
```

```
  :newline :cr+lf)
```

```
=> "1,AC,false\r\n2,WS, '-1'',true"
```

current-time-millis

(current-time-millis)

Returns the current time in milliseconds.

```
(current-time-millis)
=> 1780586456210
```

SEE ALSO

[nano-time](#)

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

[section](#)

[top](#)

cycle

```
(cycle coll)
```

Returns a lazy (infinite!) sequence of repetitions of the items in coll.

```
(doall (take 5 (cycle [1 2])))
=> (1 2 1 2 1)
```

SEE ALSO

[repeat](#)

Returns a lazy sequence of x values or a collection with the value x repeated n times.

[repeatedly](#)

Takes a function of no args, presumably with side effects, and returns a collection of n calls to it

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[constantly](#)

Returns a function that takes any number of arguments and returns always the value x.

[section](#)

[top](#)

dag/add-edges

```
(add-edges edges*)
```

Add edges to a DAG. Returns a new DAG with added edges.

An edge is a vector of two nodes forming a parent/child relationship. Any *Venice* value can be used for a node.

Note: The graph is reconstructed after adding edges. To have best performance pass the edges with a single `add-edges` call to the DAG.

```
(dag/add-edges (dag/dag) ["A" "B"] ["B" "C"])
=> (["A" "B"] ["B" "C"])
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/topological-sort](#)

Topological sort of a DAG using Kahn's algorithm (https://en.wikipedia.org/wiki/Topological_sorting)

dag/add-nodes

```
(add-nodes nodes*)
```

Add nodes to a DAG. Returns a new DAG with added nodes.

Any *Venice* value can be used for a node.

Note: The graph is reconstructed after adding nodes. To have best performance pass the nodes with a single `add-nodes` call to the DAG.

```
(dag/add-nodes (dag/dag) "A")
=> ("A")
```

```
(> (dag/dag)
    (dag/add-nodes "A")
    (dag/add-edges ["A" "B"]))
=> (["A" "B"])
```

```
(> (dag/dag)
    (dag/add-nodes "A")
    (dag/add-edges ["B" "C"]))
=> ("A" ["B" "C"])
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/topological-sort](#)

Topological sort of a DAG using Kahn's algorithm (https://en.wikipedia.org/wiki/Topological_sorting)

dag/child-of?

```
(child-of? dag c v)
```

Returns `true` if `c` is a transitive child of `v`

```
(> (dag/dag ["A", "B"] ; A E
      ["B", "C"] ; | |
      ["C", "D"] ; B F
      ["E", "F"] ; | / \
      ["F", "C"] ; C G
      ["F", "G"] ; \ /
      ["G", "D"] ; D
    (dag/child-of? "G" "E"))
=> true
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/children](#)

Returns the transitive child nodes

[dag/parent-of?](#)

Returns true if p is a transitive parent of v

[section](#)

[top](#)

[dag/children](#)

(children dag node)

Returns the transitive child nodes

```
(dag/children (dag/dag ["A" "B"] ["B" "C"]) "A")
=> ("B" "C")
```

```
(-> (dag/dag ["A", "B"] ; A E
      ["B", "C"] ; | |
      ["C", "D"] ; B F
      ["E", "F"] ; | / \
      ["F", "C"] ; C G
      ["F", "G"] ; \ /
      ["G", "D"] ; D
    (dag/children "F"))
=> ("C" "G" "D")
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/direct-children](#)

Returns the direct child nodes

[dag/parents](#)

Returns the transitive parent nodes

[dag/direct-parents](#)

Returns the direct parent nodes

[dag/roots](#)

Returns the root nodes of a DAG

[section](#)

[top](#)

[dag/compare-fn](#)

(compare-fn dag)

Returns a comparator fn which produces a topological sort based on the dependencies in the graph. Nodes not present in the graph will sort after nodes in the graph.

```
(let [g (dag/dag ["A", "B"] ; A E
      ["B", "C"] ; | |
      ["C", "D"] ; B F
      ["E", "F"] ; | / \
      ["F", "C"] ; C G
      ["F", "G"] ; \ /
```

```
      ["G", "D"])] ;      D
(sort (dag/compare-fn g) ["D" "F" "A" "Z"]))
=> ["F" "A" "D" "Z"]
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/topological-sort](#)

Topological sort of a DAG using Kahn's algorithm (https://en.wikipedia.org/wiki/Topological_sorting)

[section](#)

[top](#)

[dag/dag](#)

```
(dag)
(dag edges*)
```

Creates a new DAG (directed acyclic graph) built from edges

An edge is a vector of two nodes forming a parent/child relationship.

```
(dag/dag)
=> ()
```

```
(dag/dag ["A" "B"] ["B" "C"])
=> (["A" "B"] ["B" "C"])
```

```
(dag/dag ["A", "B"] ; A E
["B", "C"] ; | |
["C", "D"] ; B F
["E", "F"] ; | / \
["F", "C"] ; C G
["F", "G"] ; \ /
["G", "D"] ; D
=> (["A" "B"] ["B" "C"] ["C" "D"] ["E" "F"] ["F" "C"] ["F" "G"] ["G" "D"])
```

SEE ALSO

[dag/dag?](#)

Returns true if coll is a DAG

[dag/add-edges](#)

Add edges to a DAG. Returns a new DAG with added edges.

[dag/add-nodes](#)

Add nodes to a DAG. Returns a new DAG with added nodes.

[dag/topological-sort](#)

Topological sort of a DAG using Kahn's algorithm (https://en.wikipedia.org/wiki/Topological_sorting)

[dag/edges](#)

Returns the edges of a DAG

[dag/edge?](#)

Returns true if the edge given by its parent and child node is part of the DAG

[dag/nodes](#)

Returns the nodes of a DAG

[dag/node?](#)

Returns true if v is a node in the DAG

[dag/roots](#)

Returns the root nodes of a DAG

[dag/children](#)

Returns the transitive child nodes

[dag/direct-children](#)

Returns the direct child nodes

[dag/child-of?](#)

Returns true if c is a transitive child of v

[dag/parents](#)

Returns the transitive parent nodes

[dag/direct-parents](#)

Returns the direct parent nodes

[dag/parent-of?](#)

Returns true if p is a transitive parent of v

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)

[top](#)

[dag/dag?](#)

```
(dag? coll)
```

Returns true if coll is a DAG

```
(dag/dag? (dag/dag))  
=> true
```

[section](#)

[top](#)

[dag/direct-children](#)

```
(direct-children dag node)
```

Returns the direct child nodes

```
(-> (dag/dag ["A", "B"] ; A E  
          ["B", "C"] ; | |  
          ["C", "D"] ; B F  
          ["E", "F"] ; | / \  
          ["F", "C"] ; C G  
          ["F", "G"] ; \ /  
          ["G", "D"] ; D  
    (dag/direct-children "F"))  
=> ("C" "G")
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/children](#)

Returns the transitive child nodes

[dag/parents](#)

Returns the transitive parent nodes

[dag/direct-parents](#)

Returns the direct parent nodes

[dag/roots](#)

Returns the root nodes of a DAG

[section](#)

[top](#)

dag/direct-parents

(direct-parents dag node)

Returns the direct parent nodes

```
(dag/parents (dag/dag ["A" "B"] ["B" "C"]) "C")  
=> ("B" "A")
```

```
(-> (dag/dag ["A", "B"] ; A E  
      ["B", "C"] ; | |  
      ["C", "D"] ; B F  
      ["E", "F"] ; | / \  
      ["F", "C"] ; C G  
      ["F", "G"] ; \ /  
      ["G", "D"]) ; D  
  (dag/direct-parents "C"))  
=> ("B" "F")
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/parents](#)

Returns the transitive parent nodes

[dag/children](#)

Returns the transitive child nodes

[dag/direct-children](#)

Returns the direct child nodes

[dag/roots](#)

Returns the root nodes of a DAG

[section](#)

[top](#)

dag/edge?

(edge? dag parent child)

Returns `true` if the edge given by its parent and child node is part of the DAG

```
(-> (dag/dag ["A", "B"] ; A E
      ["B", "C"] ; | |
      ["C", "D"] ; B F
      ["E", "F"] ; | / \
      ["F", "C"] ; C G
      ["F", "G"] ; \ /
      ["G", "D"] ; D
    (dag/edge? "C", "D"))
=> true
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/edges](#)

Returns the edges of a DAG

[section](#)

[top](#)

dag/edges

```
(edges dag)
```

Returns the edges of a DAG

```
(dag/edges (dag/dag ["A" "B"] ["B" "C"]))
=> (["A" "B"] ["B" "C"])
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/add-edges](#)

Add edges to a DAG. Returns a new DAG with added edges.

[dag/nodes](#)

Returns the nodes of a DAG

[section](#)

[top](#)

dag/node?

```
(node? dag v)
```

Returns `true` if `v` is a node in the DAG

```
(-> (dag/dag ["A", "B"] ; A E
      ["B", "C"] ; | |
      ["C", "D"] ; B F
      ["E", "F"] ; | / \
      ["F", "C"] ; C G
      ["F", "G"] ; \ /
      ["G", "D"] ; D
    (dag/node? "G"))
=> true
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/nodes](#)

Returns the nodes of a DAG

[section](#)

[top](#)

dag/nodes

```
(nodes dag)
```

Returns the nodes of a DAG

```
(dag/nodes (dag/dag ["A" "B"] ["B" "C"]))  
=> ("A" "B" "C")
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/node?](#)

Returns true if v is a node in the DAG

[dag/add-edges](#)

Add edges to a DAG. Returns a new DAG with added edges.

[dag/edges](#)

Returns the edges of a DAG

[section](#)

[top](#)

dag/parent-of?

```
(parent-of? dag p v)
```

Returns `true` if p is a transitive parent of v

```
(-> (dag/dag ["A", "B"] ; A E  
        ["B", "C"] ; | |  
        ["C", "D"] ; B F  
        ["E", "F"] ; | / \  
        ["F", "C"] ; C G  
        ["F", "G"] ; \ /  
        ["G", "D"]) ; D  
    (dag/parent-of? "E" "G"))  
=> true
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/parents](#)

Returns the transitive parent nodes

[dag/child-of?](#)

Returns true if c is a transitive child of v

[section](#)

[top](#)

[dag/parents](#)

(parents dag node)

Returns the transitive parent nodes

```
(dag/parents (dag/dag ["A" "B"] ["B" "C"]) "C")  
=> ("B" "A")
```

```
(-> (dag/dag ["A", "B"] ; A E  
      ["B", "C"] ; | |  
      ["C", "D"] ; B F  
      ["E", "F"] ; | / \  
      ["F", "C"] ; C G  
      ["F", "G"] ; \ /  
      ["G", "D"] ; D  
      (dag/parents "C"))  
=> ("B" "F" "A" "E")
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/direct-parents](#)

Returns the direct parent nodes

[dag/children](#)

Returns the transitive child nodes

[dag/direct-children](#)

Returns the direct child nodes

[dag/roots](#)

Returns the root nodes of a DAG

[section](#)

[top](#)

[dag/roots](#)

(roots dag)

Returns the root nodes of a DAG

```
(dag/roots (dag/dag ["A" "B"] ["B" "C"]))  
=> ("A")
```

```
(-> (dag/dag ["A", "B"] ; A E  
      ["B", "C"] ; | |  
      ["C", "D"] ; B F  
      ["E", "F"] ; | / \  
      ["F", "G"] ; \ /  
      ["G", "D"] ; D  
      (dag/roots "C"))  
=> ("A")
```

```
      ["F", "C"] ;      C      G
      ["F", "G"] ;      \    /
      ["G", "D"]) ;      D

(dag/roots))
=> ("A" "E")
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/parents](#)

Returns the transitive parent nodes

[dag/children](#)

Returns the transitive child nodes

[section](#)

[top](#)

dag/topological-sort

```
(topological-sort dag)
```

Topological sort of a DAG using [Kahn's algorithm](#)

```
(dag/topological-sort (dag/dag ["A" "B"] ["B" "C"]))
=> ["A" "B" "C"]
```

```
(-> (dag/dag ["A", "B"] ;      A      E
      ["B", "C"] ;      |      |
      ["C", "D"] ;      B      F
      ["E", "F"] ;      |    /    \
      ["F", "C"] ;      C      G
      ["F", "G"] ;      \    /
      ["G", "D"]) ;      D

(dag/topological-sort))
=> ["E" "F" "G" "A" "B" "C" "D"]
```

SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph) built from edges

[dag/compare-fn](#)

Returns a comparator fn which produces a topological sort based on the dependencies in the graph. Nodes not present in the graph will ...

[dag/add-edges](#)

Add edges to a DAG. Returns a new DAG with added edges.

[section](#)

[top](#)

dec

```
(dec x)
```

Decrements the number x


```
(dec 10)
=> 9
```

```
(dec 10I)
=> 9I
```

```
(dec 10.1)
=> 9.1
```

```
(dec 10.12M)
=> 9.12M
```

SEE ALSO

[inc](#)

Increments the number x

[section](#)

[top](#)

dec/add

```
(dec/add x y scale rounding-mode)
```

Adds two decimals and scales the result. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, or `:UP`

```
(dec/add 2.44697M 1.79882M 3 :HALF_UP)
=> 4.246M
```

SEE ALSO

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, or `:UP`

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, or `:UP`

[dec/scale](#)

Scales a decimal. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, or `:UP`

[section](#)

[top](#)

dec/div

```
(dec/div x y scale rounding-mode)
```

Divides x by y and scales the result. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, or `:UP`

```
(dec/div 2.44697M 1.79882M 5 :HALF_UP)
=> 1.36032M
```

SEE ALSO

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

dec/mul

```
(dec/mul x y scale rounding-mode)
```

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

```
(dec/mul 2.44697M 1.79882M 5 :HALF_UP)
=> 4.40166M
```

SEE ALSO

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

dec/scale

```
(dec/scale x scale rounding-mode)
```

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

```
(dec/scale 2.44697M 0 :HALF_UP)
```

```
=> 2M
```

```
(dec/scale 2.44697M 1 :HALF_UP)
```

```
=> 2.4M
```

```
(dec/scale 2.44697M 2 :HALF_UP)
```

```
=> 2.45M
```

```
(dec/scale 2.44697M 3 :HALF_UP)
```

```
=> 2.447M
```

```
(dec/scale 2.44697M 10 :HALF_UP)
```

```
=> 2.4469700000M
```

SEE ALSO

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/sub](#)

Subtract y from x and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[section](#)

[top](#)

dec/sub

```
(dec/sub x y scale rounding-mode)
```

Subtract y from x and scales the result. rounding-mode is one of `:CEILING`, `:DOWN`, `:FLOOR`, `:HALF_DOWN`, `:HALF_EVEN`, `:HALF_UP`, `:UNNECESSARY`, or `:UP`

```
(dec/sub 2.44697M 1.79882M 3 :HALF_UP)
```

```
=> 0.648M
```

SEE ALSO

[dec/add](#)

Adds two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/mul](#)

Multiplies two decimals and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, ...

[dec/div](#)

Divides x by y and scales the result. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

[dec/scale](#)

Scales a decimal. rounding-mode is one of :CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, or :UP

decimal

```
(decimal x) (decimal x scale rounding-mode)
```

Converts to decimal. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, :UP)

```
(decimal 2)
```

```
=> 2M
```

```
(decimal 2 3 :HALF_UP)
```

```
=> 2.000M
```

```
(decimal 2.5787 3 :HALF_UP)
```

```
=> 2.579M
```

```
(decimal 2.5787F 3 :HALF_UP)
```

```
=> 2.579M
```

```
(decimal 2.5787M 3 :HALF_UP)
```

```
=> 2.579M
```

```
(decimal "2.5787" 3 :HALF_UP)
```

```
=> 2.579M
```

```
(decimal nil)
```

```
=> 0M
```

decimal?

```
(decimal? n)
```

Returns true if n is a decimal

```
(decimal? 4.0M)
```

```
=> true
```

```
(decimal? 4.0)
```

```
=> false
```

```
(decimal? 3)
```

```
=> false
```

```
(decimal? 3I)
```

```
=> false
```

dedupe

```
(dedupe coll)
```

Returns a collection with all consecutive duplicates removed.
Returns a stateful transducer when no collection is provided.

```
(dedupe [1 2 2 2 3 4 4 2 3])  
=> [1 2 3 4 2 3]
```

```
(dedupe '(1 2 2 2 3 4 4 2 3))  
=> (1 2 3 4 2 3)
```

SEE ALSO

[distinct](#)

Returns a collection with all duplicates removed.

[section](#)

[top](#)

def

```
(def name expr)
```

Creates a global variable.

```
(def x 5)  
=> user/x
```

```
(def sum (fn [x y] (+ x y)))  
=> user/sum
```

```
(def ^{:private true} x 100)  
=> user/x
```

SEE ALSO

[def](#)

Creates a global variable.

[def-](#)

Same as def, yielding non-public def

[defonce](#)

Creates a global variable that can not be overwritten

[def-dynamic](#)

Creates a dynamic variable that starts off as a global variable and can be bound with 'binding' to a new value on the local thread.

[set!](#)

Sets a global or thread-local variable to the value of the expression.

[section](#)

[top](#)

def-

```
(def- name expr)
```

Same as `def` , yielding non-public `def`

```
(def- x 100)
```

```
(do
  (ns foo)
  (def- x 100)
  (ns bar)
  foo/x) ; Illegal access of private symbol
```

SEE ALSO

[def](#)

Creates a global variable.

[section](#)

[top](#)

def-dynamic

```
(def-dynamic name expr)
```

Creates a dynamic variable that starts off as a global variable and can be bound with 'binding' to a new value on the local thread.

```
(do
  (def-dynamic x 100)
  (println x)
  (binding [x 200]
    (println x))
  (println x))
100
200
100
=> nil

(def-dynamic ^{:private true} x 100)
=> user/x
```

SEE ALSO

[binding](#)

Evaluates the expressions and binds the values to dynamic (thread-local) symbols

[def](#)

Creates a global variable.

[defonce](#)

Creates a global variable that can not be overwritten

[set!](#)

Sets a global or thread-local variable to the value of the expression.

[section](#)

[top](#)

defmacro

```
(defmacro name [params*] body)
```

Macro definition

```
(defmacro unless [pred a b]
  `(if (not ~pred) ~a ~b))
=> user/unless
```

SEE ALSO

[macroexpand](#)

If form represents a macro form, returns its expansion, else returns form.

[macroexpand-all](#)

Recursively expands all macros in the form.

[section](#)

[top](#)

defmethod

```
(defmethod multifn-name dispatch-val & fn-tail)
```

Creates a new method for a multimethod associated with a dispatch-value.

```
(do
  ;;defmulti with dispatch function
  (defmulti salary (fn [amount] (amount :t)))

  ;;defmethod provides a function implementation for a particular value
  (defmethod salary "com"      [amount] (+ (:b amount) (/ (:b amount) 2)))
  (defmethod salary "bon"      [amount] (+ (:b amount) 99))
  (defmethod salary :default   [amount] (:b amount))

  [(salary {:t "com" :b 1000})
   (salary {:t "bon" :b 1000})
   (salary {:t "xxx" :b 1000})]
)
=> [1500 1099 1000]
```

SEE ALSO

[defmulti](#)

Creates a new multimethod with the associated dispatch function.

[section](#)

[top](#)

defmulti

```
(defmulti name dispatch-fn)
```

Creates a new multimethod with the associated dispatch function.

```
(do
  ;;defmulti with dispatch function
  (defmulti salary (fn [amount] (amount :t)))
```

```

;;defmethod provides a function implementation for a particular value
(defmethod salary "com"      [amount] (+ (:b amount) (/ (:b amount) 2)))
(defmethod salary "bon"      [amount] (+ (:b amount) 99))
(defmethod salary :default   [amount] (:b amount))

[(salary {:t "com" :b 1000})
 (salary {:t "bon" :b 1000})
 (salary {:t "xxx" :b 1000})]
)
=> [1500 1099 1000]

(do
  ;;dispatch on type
  (defmulti test (fn [x] (type x)))

  (defmethod test :core/number [x] [x :number])
  (defmethod test :core/string [x] [x :string])
  (defmethod test :core/boolean [x] [x :boolean])
  (defmethod test :default     [x] [x :default])

  [(test 1)
   (test 1.0)
   (test 1.0M)
   (test "abc")
   (test [1])])
)
=> [[1 :number] [1.0 :number] [1.0M :number] ["abc" :string] [[1] :default]]

```

SEE ALSO

[defmethod](#)

Creates a new method for a multimethod associated with a dispatch-value.

[section](#)

[top](#)

defn

```

(defn name [args*] condition-map? expr*)
(defn name ([args*] condition-map? expr*)+)

```

Same as `(def name (fn name [args*] condition-map? expr*))` or `(def name (fn name ([args*] condition-map? expr*)+))`

```

(defn sum [x y] (+ x y))
=> user/sum

(defn sum [x y] { :pre [(> x 0)] } (+ x y))
=> user/sum

(defn sum
  ([] 0)
  ([x] x)
  ([x y] (+ x y)))
=> user/sum

```

SEE ALSO

[defn-](#)

Same as defn, yielding non-public def

fn

Defines an anonymous function.

def

Creates a global variable.

[section](#)

[top](#)

defn-

```
(defn- name [args*] condition-map? expr*)
(defn- name ([args*] condition-map? expr*)+)
```

Same as `defn`, yielding non-public `def`

```
(defn- sum [x y] (+ x y))
=> user/sum
```

SEE ALSO

defn

Same as (def name (fn name [args*] condition-map? expr*)) or (def name (fn name ([args*] condition-map? expr*)+))

fn

Defines an anonymous function.

def

Creates a global variable.

[section](#)

[top](#)

defonce

```
(defonce name expr)
```

Creates a global variable that can not be overwritten

```
(defonce x 5)
=> user/x
```

```
(defonce ^{:private true} x 5)
=> user/x
```

SEE ALSO

def

Creates a global variable.

def-dynamic

Creates a dynamic variable that starts off as a global variable and can be bound with 'binding' to a new value on the local thread.

[section](#)

[top](#)

defprotocol

```
(defprotocol protocol fn-spec*)
```

Defines a new protocol with the supplied function specs.

Formats:

- `(defprotocol P (foo [x]))`
- `(defprotocol P (foo [x] [x y]))`
- `(defprotocol P (foo [x] [x y] nil))`
- `(defprotocol P (foo [x] [x y] 100))`
- `(defprotocol P (foo [x]) (bar [x] [x y]))`

```
(do
  (ns foo)
  (deftype :complex [re :long, im :long])
  (defprotocol XMath (+ [x y]
                       (- [x y]))
    (extend :foo/complex XMath
      (+ [x y] (complex. (core/+ (:re x) (:re y))
                          (core/+ (:im x) (:im y))))
      (- [x y] (complex. (core/- (:re x) (:re y))
                          (core/- (:im x) (:im y)))))

    (extend :core/long XMath
      (+ [x y] (core/+ x y))
      (- [x y] (core/- x y)))
    (foo/+ (complex. 1 1) (complex. 4 5)))
=> {:custom-type* :foo/complex :re 5 :im 6}
```

```
(do
  (ns foo)
  (defprotocol Lifecycle (start [c]) (stop [c]))
  (deftype :component [name :string]
    Lifecycle (start [c] (println "'~(:name c)' started"))
              (stop [c] (println "'~(:name c)' stopped")))
  (let [c (component. "test")
        lifecycle? (extends? (type c) Lifecycle)]
    (println "'~(:name c)' extends Lifecycle protocol: ~{lifecycle?}")
    (start c)
    (stop c)))
'test' extends Lifecycle protocol: true
'test' started
'test' stopped
=> nil
```

SEE ALSO

[extend](#)

Extends protocol for type with the supplied functions.

[extends?](#)

Returns true if the type extends the protocol.

[defmulti](#)

Creates a new multimethod with the associated dispatch function.

[section](#)

[top](#)

deftype

```
(deftype name fields)
```

```
(deftype name fields validator)
```

Defines a new custom *record* type for the name with the fields.

The optional validator is a single arg function receiving the value as the argument and throwing an an exception if the value is not valid.

Venice implicitly creates a builder and a type check function suffixed with a dot and a question mark:

```
(deftype :point [x :long, y :long])

(point. 200 300)           ; builder
(point? (point. 200 300)) ; type check
```

The builder accepts values of any subtype of the field's type.

Validation example:

```
(deftype :point
  [x :long, y :long]
  (fn [t]
    (assert (pos? (:x t)) "x must be positive!"))))
```

```
(do
  (ns foo)
  (deftype :point [x :long, y :long])
  ; explicitly creating a custom type value
  (def x (. :point 100 200))
  ; Venice implicitly creates a builder function
  ; suffixed with a '.'
  (def y (point. 200 300))
  ; ... and a type check function
  (point? y)
  y)
=> {:custom-type* :foo/point :x 200 :y 300}
```

```
(do
  (ns foo)
  (deftype :point [x :long, y :long])
  (def x (point. 100 200))
  (type x))
=> :foo/point
```

```
(do
  (ns foo)
  (deftype :point [x :long, y :long]
    (fn [p]
      (assert (pos? (:x p)) "x must be positive")
      (assert (pos? (:y p)) "y must be positive"))))
  (def p (point. 100 200))
  [(:x p) (:y p)])
=> [100 200]
```

```
(do
  (ns foo)
  (deftype :named [name :string, value :any])
  (def x (named. "count" 200))
  (def y (named. "seq" [1 2]))
  [x y])
=> [{:custom-type* :foo/named :name "count" :value 200} {:custom-type* :foo/named :name "seq" :value [1 2]}]
```

;; modifying a custom type field

```
(do
  (deftype :point [x :long, y :long])
  (def p (point. 0 0))
  (def q (assoc p :x 1 :y 2)) ; q is a 'point'
```

```
(pr-str q))
=> "{:custom-type* :user/point :x 1 :y 2}"

;; removing a custom type field
(do
  (deftype :point [x :long, y :long])
  (def p (point. 100 200))
  (def q (dissoc p :x)) ; q is just a map now
  (pr-str q))
=> "{:y 200}"
```

SEE ALSO

[deftype?](#)

Returns true if type is a custom type else false.

[deftype-of](#)

Defines a new custom wrapper type based on a base type.

[deftype-or](#)

Defines a new custom choice type.

[.:](#)

Instantiates a custom type.

[deftype-describe](#)

Describes a custom type.

[Object](#)

Defines a protocol to customize the toString and/or the compareTo function of custom datatypes.

[assoc](#)

When applied to a map, returns a new map of the same type, that contains the mapping of key(s) to val(s). When applied to a vector, ...

[dissoc](#)

Returns a new coll of the same type, that does not contain a mapping for key(s)

[section](#)

[top](#)

deftype-describe

```
(deftype-describe type)
```

Describes a custom type.

```
(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (deftype-describe :complex))

=> {:type :foo/complex :custom-type :record :field-defs ({:name :real :type :core/long :index 0I :nillable
false} {:name :imaginary :type :core/long :index 1I :nillable false}) :validation-fn nil}

(do
  (ns foo)
  (deftype-of :port :long)
  (deftype-describe :port))

=> {:custom-type :wrapping :base-type :core/long :type :foo/port :validation-fn nil}

(do
  (ns foo)
```

```
(deftype-or :digit 0 1 2 3 4 5 6 7 8 9)
(deftype-describe :digit))
=> {:type :foo/digit :custom-type :choice :values #{0 1 2 3 4 5 6 7 8 9}}
```

SEE ALSO

[deftype](#)

Defines a new custom record type for the name with the fields.

[deftype?](#)

Returns true if type is a custom type else false.

[deftype-or](#)

Defines a new custom choice type.

[deftype-of](#)

Defines a new custom wrapper type based on a base type.

∴

Instantiates a custom type.

[section](#)

[top](#)

deftype-of

```
(deftype-of name base-type)
(deftype-of name base-type validator)
```

Defines a new custom *wrapper* type based on a base type.

Venice implicitly creates a builder and a type check function suffixed with a dot and a question mark:

```
(deftype-of :port :long)

(port. 8080)           ; builder
(port? (port. 8080))  ; type check
```

```
(do
  (ns foo)
  (deftype-of :email-address :string)
  ; explicitly creating a wrapper type value
  (def x (. :email-address "foo@foo.org"))
  ; Venice implicitly creates a builder function
  ; suffixed with a '.'
  (def y (email-address. "foo@foo.org"))
  ; ... and a type check function
  (email-address? y)
  y)
=> "foo@foo.org"
```

```
(do
  (ns foo)
  (deftype-of :email-address :string)
  (str "Email: " (email-address. "foo@foo.org")))
=> "Email: foo@foo.org"
```

```
(do
  (ns foo)
  (deftype-of :email-address :string)
  (def x (email-address. "foo@foo.org"))
  [(type x) (supertype x)])
=> [:foo/email-address :core/string]
```

```
(do
  (ns foo)
  (deftype-of :email-address
    :string
    str/valid-email-addr?)
  (email-address. "foo@foo.org"))
=> "foo@foo.org"
```

```
(do
  (ns foo)
  (deftype-of :contract-id :long)
  (contract-id. 100000))
=> 100000
```

```
(do
  (ns foo)
  (deftype-of :my-long :long)
  (+ 10 (my-long. 100000)))
=> 100010
```

SEE ALSO

[deftype](#)

Defines a new custom record type for the name with the fields.

[deftype?](#)

Returns true if type is a custom type else false.

[deftype-or](#)

Defines a new custom choice type.

[..](#)

Instantiates a custom type.

[deftype-describe](#)

Describes a custom type.

[section](#)

[top](#)

deftype-or

```
(deftype-or name val*)
```

Defines a new custom *choice* type.

Venice implicitly creates a builder and a type check function suffixed with a dot and a question mark:

```
(deftype-or :color :red :green :blue)

(color. :blue)           ; builder
(color? (color. :blue)) ; type check
```

```
(do
  (ns foo)
  (deftype-or :color :red :green :blue)
  ; explicitly creating a wrapper type value
  (def x (. :color :red))
  ; Venice implicitly creates a builder function
  ; suffixed with a '.'
  (def y (color. :blue))
  ; ... and a type check function
```

```
(color? y)
y)
=> "blue"

(do
  (ns foo)
  (deftype-or :digit 0 1 2 3 4 5 6 7 8 9)
  (digit. 1))
=> 1

(do
  (ns foo)
  (deftype-or :long-or-double :long :double)
  (long-or-double. 1000))
=> 1000
```

SEE ALSO

[deftype](#)

Defines a new custom record type for the name with the fields.

[deftype?](#)

Returns true if type is a custom type else false.

[deftype-of](#)

Defines a new custom wrapper type based on a base type.

[::](#)

Instantiates a custom type.

[deftype-describe](#)

Describes a custom type.

[section](#)

[top](#)

deftype?

```
(deftype? type)
```

Returns true if `type` is a custom type else false.

```
(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (deftype? :complex))
=> true

(do
  (ns foo)
  (deftype-of :email-address :string)
  (deftype? :email-address))
=> true

(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (def x (complex. 100 200))
  (deftype? (type x)))
=> true
```

SEE ALSO

[deftype](#)

Defines a new custom record type for the name with the fields.

[deftype-of](#)

Defines a new custom wrapper type based on a base type.

[deftype-or](#)

Defines a new custom choice type.

[::](#)

Instantiates a custom type.

[deftype-describe](#)

Describes a custom type.

[section](#)[top](#)

delay

(delay & body)

Takes a body of expressions and yields a Delay object that will invoke the body only the first time it is forced (with `force` or `deref` / `@`), and will cache the result and return it on all subsequent force calls.

```
(do
  (def x (delay (println "working...") 100))
  (deref x))
working...
=> 100
```

SEE ALSO

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[force](#)

If x is a delay, returns its value, else returns x

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[delay?](#)

Returns true if x is a Delay created with delay

[memoize](#)

Returns a memoized version of a referentially transparent function.

[section](#)[top](#)

delay-queue

(delay-queue)

Creates a new delay queue.

A delay-queue is an unbounded blocking queue of delayed elements, in which an element can only be taken when its delay has expired. The head of the queue is that delayed element whose delay expired furthest in the past. If no delay has expired there is no head and `poll!` will return nil. Unexpired elements cannot be removed using `take!` or `poll!`, they are otherwise treated as normal elements. For example, the `count` method returns the count of both expired and unexpired elements. This queue does not permit `nil` elements.

Example rate limiter:

```
(do
  (defprotocol RateLimiter (init [x]) (acquire [x]))

  (deftype :rate-limiter [queue                :core/delay-queue
                        limit-for-period      :long
                        limit-refresh-period :long]

    RateLimiter
    (init [this] (let [q (:queue this)
                      n (:limit-for-period this)]
                  (empty q)
                  (repeatedly n #(put! q :token 0))
                  this))
    (acquire [this] (let [q (:queue this)
                        p (:limit-refresh-period this)]
                     (take! q)
                     (put! q :token p)))))

;; create a limiter with a limit of 5 actions within a 2s period
(def limiter (init (rate-limiter. (delay-queue) 5 2000)))

;; test the limiter
(doseq [x (range 1 26)]
  (acquire limiter)
  (sleep (rand-long 100)) ;; func simulation
  (printf "%s: run %2d%n" (time/local-date-time) x)))
```

```
(let [q (delay-queue)]
  (put! q 1 100)
  (put! q 1 200)
  (take! q))
=> 1
```

SEE ALSO

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value `:indefinite`.

[empty](#)

Returns an empty collection of the same category as coll, or nil if coll is nil. If the collection is mutable clears the collection ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[delay-queue?](#)

Returns true if coll is a delay-queue

delay-queue?

```
(delay-queue? coll)
```

Returns true if coll is a delay-queue

```
(delay-queue? (delay-queue))  
=> true
```

[section](#)[top](#)

delay?

```
(delay? x)
```

Returns true if x is a Delay created with delay

```
(do  
  (def x (delay (println "working...") 100))  
  (delay? x))  
=> true
```

SEE ALSO

[delay](#)

Takes a body of expressions and yields a Delay object that will invoke the body only the first time it is forced (with force or deref ...

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[section](#)[top](#)

deliver

```
(deliver ref value)
```

Delivers the supplied value to the promise, releasing any pending derefs. A subsequent call to deliver on a promise will have no effect.

```
(do  
  (def p (promise))  
  (deliver p 10)  
  (deliver p 20) ; no effect  
  @p)  
=> 10
```

SEE ALSO

[deliver-ex](#)

Delivers the supplied exception to the promise, releasing any pending derefs. A subsequent call to deliver on a promise will have no effect.

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[section](#)

[top](#)

deliver-ex

```
(deliver-ex ref ex)
```

Delivers the supplied exception to the promise, releasing any pending derefs. A subsequent call to `deliver` on a promise will have no effect.

```
(do
  (def p (promise))
  (deliver-ex p (ex :VncException "error"))
  (deliver p 20) ; no effect
  (try
    @p
    (catch :VncException e (ex-message e))))
=> "error"
```

SEE ALSO

[deliver](#)

Delivers the supplied value to the promise, releasing any pending derefs. A subsequent call to `deliver` on a promise will have no effect.

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[section](#)

[top](#)

deque

```
(deque)
(deque capacity)
```

Creates a new mutable threadsafe bounded or unbounded deque.

The deque can be turned into a synchronous deque when using the functions `put!` and `take!`. `put!` waits until the value be added and `take!` waits until a value is available from queue thus synchronizing the producer and consumer.

```
; unbounded deque
(let [q (deque)]
  (offer! q 1)
  (offer! q 2)
  (offer! q 3)
  (poll! q)
  q)
=> (2 3)
```

```
; unbounded deque
(let [q (conj! (deque) 1 2 3)]
  (offer! q 4)
  (poll! q))
```

```

q)
=> (2 3 4)

; bounded deque
(let [q (deque 10)]
  (offer! q 1000 1)
  (offer! q 1000 2)
  (offer! q 1000 3)
  (poll! q 1000)
  q)
=> (2 3)

; bounded deque
(let [q (conj! (deque 10) 1 2 3)]
  (offer! q 4)
  (poll! q)
  q)
=> (2 3 4)

; synchronous unbounded deque
(let [q (deque)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (take! q)
  q)
=> (2 3)

; synchronous bounded deque
(let [q (deque 10)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (take! q)
  q)
=> (2 3)

```

SEE ALSO

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[put-head!](#)

Puts an item to the head of a deque. The operation is synchronous, it waits indefinitely until the value can be placed on the queue.

[take-tail!](#)

Retrieves and removes the tail value of the deque, waiting if necessary until a value becomes available.

[offer-head!](#)

Offers an item to the head of a deque with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll-tail!](#)

Polls an item from the tail of a deque with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[empty](#)

Returns an empty collection of the same category as coll, or nil if coll is nil. If the collection is mutable clears the collection ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[deque?](#)

Returns true if coll is a deque

[reduce](#)

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

[transduce](#)

Reduce with a transformation of a reduction function f (xf). If init is not supplied, (f) will be called to produce it. f should be ...

[docoll](#)

Applies f to the items of the collection presumably for side effects. Returns nil.

[into!](#)

Adds all of the items of 'from' conjoined to the mutable 'to' collection

[conj!](#)

Returns a new mutable collection with the x, xs 'added'. (conj! nil item) returns (item) and (conj! item) returns item.

[section](#)[top](#)

deque?

```
(deque? coll)
```

Returns true if coll is a deque

```
(deque? (deque))  
=> true
```

[section](#)[top](#)

deref

```
(deref x)  
(deref x timeout-ms timeout-val)
```

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will block if computation is not complete. The variant taking a timeout can be used for futures and will return `timeout-val` if the timeout (in milliseconds) is reached before a value is available. If a future is deref'd and the waiting thread is interrupted the futures are cancelled.

```
(do  
  (def counter (atom 10))  
  (deref counter))  
=> 10
```

```
(do  
  (def counter (atom 10))  
  @counter)  
=> 10
```

```
(do
  (defn task [] 100)
  (let [f (future task)]
    (deref f)))
=> 100

(do
  (defn task [] 100)
  (let [f (future task)]
    @f))
=> 100

(do
  (defn task [] 100)
  (let [f (future task)]
    (deref f 300 :timeout)))
=> 100

(do
  (def x (delay (println "working...") 100))
  @x)
working...
=> 100

(do
  (def p (promise))
  (deliver p 10)
  @p)
=> 10

(do
  (def x (agent 100))
  @x)
=> 100

(do
  (def counter (volatile 10))
  @counter)
=> 10
```

[section](#)[top](#)

deref?

```
(deref? x)
```

Returns true if x is dereferencable.

```
(deref? (atom 10))
=> true
```

```
(deref? (delay 100))
=> true
```

```
(deref? (promise))
=> true
```

```
(deref? (future (fn [] 10)))  
=> true
```

```
(deref? (volatile 100))  
=> true
```

```
(deref? (agent 100))  
=> true
```

```
(deref? (just 100))  
=> true
```

[section](#)[top](#)

difference

```
(difference s1)  
(difference s1 s2)  
(difference s1 s2 & sets)
```

Return a set that is the first set without elements of the remaining sets

```
(difference (set 1 2 3))  
=> #{1 2 3}
```

```
(difference (set 1 2) (set 2 3))  
=> #{1}
```

```
(difference (set 1 2) (set 1) (set 1 4) (set 3))  
=> #{2}
```

SEE ALSO

[union](#)

Return a set that is the union of the input sets

[intersection](#)

Return a set that is the intersection of the input sets

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[conj](#)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item) and (conj item) returns item.

[disj](#)

Returns a new set with the x, xs removed.

[section](#)[top](#)

digits

```
(digits x)
```

Returns the number of digits of x. The number x must be of type integer, long, or bigint

```
(digits 124)
=> 3

(digits -10)
=> 2

(digits 111111111111111111111111111111N)
=> 32
```

[section](#)[top](#)

disj

```
(disj set x)
(disj set x & xs)
```

Returns a new set with the x, xs removed.

```
(disj (set 1 2 3) 3)
=> #{1 2}
```

[section](#)[top](#)

dissoc

```
(dissoc coll key)
(dissoc coll key & ks)
```

Returns a new coll of the same type, that does not contain a mapping for key(s)

```
(dissoc {:a 1 :b 2 :c 3} :b)
=> {:a 1 :c 3}
```

```
(dissoc {:a 1 :b 2 :c 3} :c :b)
=> {:a 1}
```

```
(dissoc [1 2 3] 0)
=> [2 3]
```

```
(do
  (deftype :complex [real :long, imaginary :long])
  (def x (complex. 100 200))
  (def y (dissoc x :real))
  (pr-str y))
=> "{:imaginary 200}"
```

SEE ALSO

[assoc](#)

When applied to a map, returns a new map of the same type, that contains the mapping of key(s) to val(s). When applied to a vector, ...

[update](#)

Updates a value in an associative structure, where k is a key and f is a function that will take the old value and any supplied fargs ...

dissoc!

```
(dissoc! coll key)
(dissoc! coll key & ks)
```

Dissociates keys from a mutable map, returns the map

```
(dissoc! (mutable-map :a 1 :b 2 :c 3) :b)
=> {:a 1 :c 3}

(dissoc! (mutable-map :a 1 :b 2 :c 3) :c :b)
=> {:a 1}

(dissoc! (mutable-vector 1 2 3) 0)
=> [2 3]
```

SEE ALSO

[assoc!](#)

Associates key/vals with a mutable map, returns the map

[update!](#)

Updates a value in a mutable associative structure, where k is a key and f is a function that will take the old value and any supplied ...

dissoc-in

```
(dissoc-in m ks)
```

Dissociates an entry in a nested associative structure, where ks is a sequence of keys and returns a new nested structure.

```
(do
  (def users [ {:name "James" :age 26}
                {:name "John" :age 43} ])
  (dissoc-in users [1]))
=> [{:name "James" :age 26}]

(do
  (def users [ {:name "James" :age 26}
                {:name "John" :age 43} ])
  (dissoc-in users [1 :age]))
=> [{:name "James" :age 26} {:name "John"}]
```

distinct

```
(distinct coll)
```

Returns a collection with all duplicates removed.
Returns a stateful transducer when no collection is provided.

```
(distinct [1 2 3 4 2 3 4])  
=> [1 2 3 4]
```

```
(distinct '(1 2 3 4 2 3 4))  
=> (1 2 3 4)
```

SEE ALSO

[dedupe](#)

Returns a collection with all consecutive duplicates removed.

[distinct?](#)

Returns true if no two of the arguments are equal

[section](#)

[top](#)

distinct?

```
(distinct? x) (distinct? x y) (distinct? x y & more)
```

Returns true if no two of the arguments are equal

```
(distinct? 1 2 3)  
=> true
```

```
(distinct? 1 2 3 3)  
=> false
```

```
(distinct? 1 2 3 1)  
=> false
```

SEE ALSO

[distinct](#)

Returns a collection with all duplicates removed.

[section](#)

[top](#)

do

```
(do exprs)
```

Evaluates the expressions in order and returns the value of the last.

```
(do (println "Test...") (+ 1 1))  
Test...  
=> 2
```

[section](#)

[top](#)

doall

```
(doall coll)
(doall n coll)
```

When lazy sequences are produced `doall` can be used to force any effects and realize the lazy sequence. Returns the realized items in a list!

```
(->> (lazy-seq #(rand-long 100))
      (take 4)
      (doall))
=> (50 27 49 26)
```

```
(->> (lazy-seq #(rand-long 100))
      (doall 4))
=> (60 42 60 20)
```

SEE ALSO

lazy-seq

Creates a new lazy sequence.

cycle

Returns a lazy (infinite!) sequence of repetitions of the items in coll.

repeat

Returns a lazy sequence of x values or a collection with the value x repeated n times.

cons

Returns a new collection where x is the first element and coll is the rest.

section

top

dobench

```
(dobench iterations expr)
(dobench warm-up-iterations gc-runs iterations expr)
```

Runs the expr iterations times in the most effective way and returns a list of elapsed nanoseconds for each invocation. It's main purpose is supporting benchmark tests.

Note: For best performance enable `macroexpand-on-load` !

[illegible][illegible]

section

top

doc

(doc)

```
(doc x)
```

Prints documentation for a var or special form given `x` as its name. Prints the definition of custom types.

Displays the source of a module if `x` is a module: `(doc :ansi)`

If the var could not be found, searches for a similar var with the **Levenshtein distance** 1.

E.g:

```
> (doc dac)
Symbol 'dac' not found!

Did you mean?
  dag/dag
  dec
```

If you search for functions provided by a module use:

```
(finder "ipc/*")
```

```
;; documentation of function '+'
(doc +)
```

```
;; documentation of special form 'def'
(doc def)
```

```
;; source code of module ':ascii-table'
(doc :ascii-table)
```

```
;; definition/structure of a complex type
(do
  (deftype :complex [real :long, imaginary :long])
  (doc :complex))
```

SEE ALSO

[ns-list](#)

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

[modules](#)

Lists the available Venice modules

[finder](#)

Finds symbols that match one more glob patterns or regular expressions.

[section](#)

[top](#)

docker/cmd

```
(docker/cmd & args)
```

Runs any Docker command.

```
(println (docker/cmd "ps" "--all"))
```

```
;; a single string argument works as well
(println (docker/cmd "ps --all"))
```

```
;; run a command with JSON output and parse the JSON output into
;; Venice data
;; use `apply` to apply a vector of arguments
```

```
(-<> (apply docker/cmd ["ps" "--all" "--format" "json"])\n  (:out <>)\n  (str/split-lines <>)\n  (str/join "," <>)\n  (str "[" <> "]" )\n  (json/read-str <>))
```

[section](#)[top](#)

docker/container-exec-by-name

(docker/container-exec-by-name name command)

Execute a command in the running container with the specified name (always in non detached mode).

Returns the captured stdout text if the command succeeds.

Throws `ShellException` if the command fails. The exception carries the exit code and the captured stderr text.

```
(docker/container-exec-by-name "myapp" "touch /tmp/execWorks")
```

SEE ALSO

[docker/container-exec-by-name&](#)

Execute a command in the running container with the specified name (always in detached mode).

[docker/run](#)

Create and run a new container from an image.

[docker/container-running-with-name?](#)

Checks if there is container with the specified name in 'running' state.

[docker/container-exec-by-name](#)

Execute a command in the running container with the specified name (always in non detached mode).

[docker/container-logs](#)

Returns the container logs.

[section](#)[top](#)

docker/container-exec-by-name&

(docker/container-exec-by-name& name command)

Execute a command in the running container with the specified name (always in detached mode).

Returns always an empty string because the command is run in detached mode. To get the commands captured stdout text use `docker /container-exec-by-name` instead.

Throws a `ShellException` if the command fails. The `ShellException` carries the exit code, stdout, and stderr text.

```
(docker/container-exec-by-name& "myapp" "touch /tmp/execWorks")
```

SEE ALSO

[docker/container-exec-by-name](#)

Execute a command in the running container with the specified name (always in non detached mode).

[docker/run](#)

Create and run a new container from an image.

[docker/container-running-with-name?](#)

Checks if there is container with the specified name in 'running' state.

[docker/container-exec-by-name](#)

Execute a command in the running container with the specified name (always in non detached mode).

[docker/container-logs](#)

Returns the container logs.

[section](#)

[top](#)

docker/container-exists-with-name?

```
(docker/container-exists-with-name? name)
```

Returns true if there is container with the specified name else false.

```
(docker/container-exists-with-name? "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-find-by-name](#)

Find all containers with a specified name

[section](#)

[top](#)

docker/container-find-by-name

```
(docker/container-find-by-name name)
```

Find all containers with a specified name

```
(docker/container-find-by-name "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-exists-with-name?](#)

Returns true if there is container with the specified name else false.

[section](#)

[top](#)

docker/container-image-info-by-name

```
(docker/container-image-info-by-name name)
```

Returns the image info for a container given by its name.

Returns a map (e.g.): `{ :image "arangodb/arangodb:3.10.10" :repo "arangodb/arangodb" :tag "3.10.10" }`

```
(docker/container-image-info-by-name "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-find-by-name](#)

Find all containers with a specified name

[docker/container-exists-with-name?](#)

Returns true if there is container with the specified name else false.

[docker/container-status-by-name](#)

Returns the status of container with the specified name.

[section](#)

[top](#)

docker/container-logs

```
(docker/container-logs name & options)
```

Returns the container logs.

Options:

:tail n	Number of lines to show from the end of the logs
:since ts	Show logs since timestamp or relative (e.g. "42m" for 42 minutes)
:until ts	Show logs until timestamp or relative (e.g. "42m" for 42 minutes)
:follow {true, false}	Follow log output
:details {true, false}	Show extra details provided to logs

```
(docker/container-logs "myapp")
```

```
(docker/container-logs "myapp" :since "2m")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/logs](#)

Get the logs of a container

[docker/container-running-with-name?](#)

Checks if there is container with the specified name in 'running' state.

[section](#)

[top](#)

docker/container-purge-by-name

```
(docker/container-purge-by-name name)
```

Removes a container and its image.

```
(docker/container-purge-by-name "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-find-by-name](#)

Find all containers with a specified name

[docker/container-exists-with-name?](#)

Returns true if there is container with the specified name else false.

[docker/container-status-by-name](#)

Returns the status of container with the specified name.

[section](#)

[top](#)

docker/container-remove-by-name

```
(docker/container-remove-by-name name)
```

Removes a container with the specified name.

```
(docker/container-remove-by-name "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-find-by-name](#)

Find all containers with a specified name

[docker/container-exists-with-name?](#)

Returns true if there is container with the specified name else false.

[docker/container-status-by-name](#)

Returns the status of container with the specified name.

[section](#)

[top](#)

docker/container-running-with-name?

```
(docker/container-running-with-name? name)
```

Checks if there is container with the specified name in 'running' state.

Returns true if running else false.

```
(docker/container-running-with-name? "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-start-by-name](#)

Starts a container with the specified name.

[docker/container-stop-by-name](#)

Stops a container with the specified name.

docker/container-start-by-name

(docker/container-start-by-name name)

Starts a container with the specified name.

([docker/container-start-by-name](#) "myapp")

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-running-with-name?](#)

Checks if there is container with the specified name in 'running' state.

[docker/container-stop-by-name](#)

Stops a container with the specified name.

[docker/container-remove-by-name](#)

Removes a container with the specified name.

[docker/container-status-by-name](#)

Returns the status of container with the specified name.

[docker/container-logs](#)

Returns the container logs.

docker/container-status-by-name

(docker/container-status-by-name name)

Returns the status of container with the specified name.

([docker/container-status-by-name](#) "myapp")

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-find-by-name](#)

Find all containers with a specified name

[docker/container-exists-with-name?](#)

Returns true if there is container with the specified name else false.

[docker/container-remove-by-name](#)

Removes a container with the specified name.

docker/container-stop-by-name

```
(docker/container-stop-by-name name)
(docker/container-stop-by-name name time)
```

Stops a container with the specified name.

```
(docker/container-stop-by-name "myapp")
```

SEE ALSO

[docker/run](#)

Create and run a new container from an image.

[docker/container-running-with-name?](#)

Checks if there is container with the specified name in 'running' state.

[docker/container-stop-by-name](#)

Stops a container with the specified name.

[docker/container-remove-by-name](#)

Removes a container with the specified name.

[docker/container-status-by-name](#)

Returns the status of container with the specified name.

[docker/container-logs](#)

Returns the container logs.

[section](#)

[top](#)

docker/cp

```
(docker/cp src-path dst-path & options)
```

Copy files/folders between a container and the local filesystem

Options:

:archive {true, false}	Archive mode (copy all uid/gid information)
:follow-link {true, false}	Always follow symbol link in SRC_PATH
:quiet {true, false}	Suppress progress output during copy. Progress output is automatically suppressed if no terminal is attached

```
;; Copy file from host to docker container
(docker/cp "data.txt" "74789744g489:/data.txt")

;; Copy file from docker container to host
(docker/cp "74789744g489:/data.txt" "data.txt")

;; Copy a folder from host to docker container
(docker/cp "Desktop/images" "74789744g489:/root/img_files/car_photos/images")

;; Copy a folder from docker container to host
(docker/cp "74789744g489:/root/img_files/car_photos/images Desktop/images")
```

SEE ALSO

[docker/diff](#)

Inspect changes to files or directories on a container's filesystem.

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/debug

```
(docker/debug mode)
```

Sets the debugging mode.

Without argument returns the current debug mode.

Mode:

:off	No debug output
:on	Prints the raw docker command line to the current stdout channel ahead of running the command
:on-no-exec	Prints the raw docker command line to the current stdout channel without running the command

```
(docker/debug :on)
```

```
(docker/debug :on-no-exec)
```

```
(docker/debug :off)
```

[section](#)

[top](#)

docker/diff

```
(docker/diff container & options)
```

Inspect changes to files or directories on a container's filesystem.

Options:

:format {string, json} Returns the output either as a string or as JSON data

```
(println (docker/diff "74789744g489"))
```

```
(docker/diff "74789744g4892" :format :json)
```

SEE ALSO

[docker/cp](#)

Copy files/folders between a container and the local filesystem

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

docker/exec

```
(docker/exec container command args)
```

Execute a command in a running container (always in non detached mode).

Returns the captured stdout text if the command succeeds.

Throws `ShellException` if the command fails. The exception carries the exit code and the captured stderr text.

```
(docker/exec "74789744g489" "touch" "/tmp/execWorks")
```

```
(println (docker/exec "74789744g489" "ls" "-la" "/var"))
```

SEE ALSO

[docker/exec&](#)

Execute a command in a running container (always in detached mode).

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

docker/exec&

```
(docker/exec& container command args)
```

Execute a command in a running container (always in detached mode).

Returns always an empty string because the command is run in detached mode. To get the commands captured stdout text use `docker/exec` instead.

Throws `ShellException` if the command fails. The `ShellException` carries the exit code, stdout, and stderr text.

```
(docker/exec& "74789744g489" "touch" "/tmp/execWorks")
```

```
(docker/exec& "74789744g489" "ls" "/var")
```

SEE ALSO

[docker/exec](#)

Execute a command in a running container (always in non detached mode).

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

docker/image-prune

(docker/image-prune & options)

Remove unused images.

If `:all true` is specified, will also remove all images not referenced by any container. This is what you usually expect

Returns the stdout text from the command.

Options:

`:all {true, false}` Remove all unused images, not just dangling ones

```
(println (docker/image-prune))
```

```
(println (docker/image-prune :all true))
```

SEE ALSO

[docker/images](#)

List images.

[docker/image-pull](#)

Download an image from a registry.

[docker/rmi](#)

Remove an image.

[docker/image-rm](#)

Remove an image.

[section](#)

[top](#)

docker/image-pull

(docker/image-pull name & options)

Download an image from a registry.

Images can be pulled by name, name and tag, or digest

Returns the stdout text from the command.

Options:

`:quiet {true, false}` Suppress verbose output

```
(println (docker/image-pull "arangodb/arangodb:3.10.10"))
```

```
(println (docker/image-pull "arangodb/arangodb"))
```

SEE ALSO

[docker/images](#)

List images.

[docker/rmi](#)

Remove an image.

[docker/image-rm](#)

Remove an image.

[docker/image-prune](#)

Remove unused images.

[section](#)

[top](#)

docker/image-ready?

```
(docker/image-ready? repo tag)
```

Returns true if the image exists locally (is pulled) else false.

```
(docker/image-ready? "arangodb/arangodb" "3.10.10")
```

SEE ALSO

[docker/images](#)

List images.

[docker/images-query-by-repo](#)

Returns all pulled local images for a given repo.

[section](#)

[top](#)

docker/image-rm

```
(docker/image-rm image)
```

Remove an image.

```
(println (docker/image-rm "184e47dd1c58"))
```

SEE ALSO

[docker/images](#)

List images.

[docker/image-pull](#)

Download an image from a registry.

[docker/rmi](#)

Remove an image.

[docker/image-prune](#)

Remove unused images.

[section](#)

[top](#)

docker/images

```
(docker/images & options)
```

List images.

Options:

<code>:all {true, false}</code>	Show all images (default hides intermediate images)
<code>:digests {true, false}</code>	Show digests
<code>:quiet {true, false}</code>	If true only display image IDs
<code>:no-trunc {true, false}</code>	Don't truncate output
<code>:format f</code>	Returns the output either as a table string or as JSON data. The format is one of {table, json}

```
(println (docker/images :format :table))
```

```
(docker/images :quiet true :no-trunc true :format :json)
```

```
(println (docker/images :format :json))
```

SEE ALSO

[docker/image-pull](#)

Download an image from a registry.

[docker/rmi](#)

Remove an image.

[docker/image-rm](#)

Remove an image.

[docker/image-prune](#)

Remove unused images.

[docker/run](#)

Create and run a new container from an image.

[docker/images-query-by-repo](#)

Returns all pulled local images for a given repo.

[docker/image-ready?](#)

Returns true if the image exists locally (is pulled) else false.

[section](#)

[top](#)

docker/images-query-by-repo

```
(docker/images-query-by-repo repo)
```

Returns all pulled local images for a given repo.

```
(docker/images-query-by-repo "arangodb/arangodb")
```

```
;; return a list of ids for "arangodb/arangodb" images
(->> (docker/images-query-by-repo "arangodb/arangodb")
      (map #(get % "ID")))
```

SEE ALSO

[docker/images](#)

List images.

[docker/image-ready?](#)

Returns true if the image exists locally (is pulled) else false.

[section](#)

[top](#)

docker/logs

(docker/logs container & options)

Get the logs of a container

Options:

:tail n	Number of lines to show from the end of the logs
:since ts	Show logs since timestamp or relative (e.g. "42m" for 42 minutes)
:until ts	Show logs until timestamp or relative (e.g. "42m" for 42 minutes)
:	Show timestamps
timestamps {true, false}	
:follow {true, false}	Follow log output
:details {true, false}	Show extra details provided to logs
:stream {: out, :err, : out+err}	Return the :out and/or :err stream from the logs. Defaults to :out

([docker/logs](#) "74789744g489")

([docker/logs](#) "74789744g489" :tail 100 :timestamps true :stream :out+err)

([docker/logs](#) "74789744g489" :since "60m" :until "30m")

SEE ALSO

[docker/pause](#)

Pause all processes within a container

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/pause

(docker/pause container)

Pause all processes within a container

([docker/pause](#) "74789744g489")

SEE ALSO

[docker/unpause](#)

Unpause all processes within a container

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/prune

([docker/prune](#))

Remove all stopped containers.

([docker/prune](#))

SEE ALSO

[docker/rm](#)

Remove a container.

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/ps

([docker/ps](#) & options)

List containers.

Options:

<code>:all {true, false}</code>	Show all containers (default shows just running)
<code>:last n</code>	Show n last created containers
<code>:quiet {true, false}</code>	If true only display container IDs
<code>:no-trunc {true, false}</code>	Don't truncate output
<code>:format {table, json}</code>	Returns the output either as a table string or as JSON data

```
(println (docker/ps :all true :format :table))
```

```
(docker/ps :all true :format :json)
```

```
(docker/ps :all true :no-trunc true :format :json)
```

```
(docker/ps :all true :no-trunc true :last 3 :format :json)
```

```
(println (docker/ps :all true :format :json))
```

SEE ALSO

[docker/start](#)

Start a stopped container.

[docker/stop](#)

Stop a container.

[docker/rm](#)

Remove a container.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/rm

(`docker/rm container & options`)

Remove a container.

Options:

<code>:force {true, false}</code>	Force the removal of a running container (uses SIGKILL)
<code>:link link</code>	Remove the specified link
<code>:volumes {true, false}</code>	Remove anonymous volumes associated with the container

(`docker/rm "74789744g489"`)

SEE ALSO

[docker/prune](#)

Remove all stopped containers.

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/rmi

(`docker/rmi image & options`)

Remove an image.

Images can be removed by name, name and tag, or image id

Options:

<code>:force {true, false}</code>	Force removal of the image
<code>:no-prune {true, false}</code>	Do not delete untagged parents

(`println (docker/rmi "arangodb/arangodb:3.10.10" :force true)`)

SEE ALSO

[docker/images](#)

List images.

[docker/image-pull](#)

Download an image from a registry.

[docker/image-rm](#)

Remove an image.

[docker/image-prune](#)

Remove unused images.

[section](#)

[top](#)

docker/run

([docker/run](#) image & options)

Create and run a new container from an image.

Images can be run by name, name and tag, or image id

Options:

<code>:detach {true, false}</code>	Run container in background and return container ID
<code>:attach s</code>	Attach to STDIN, STDOUT or STDERR. Use one of <code>{:stdin, :stdout, :stderr}</code>
<code>:publish port</code>	Publish a container's port to the host. To expose port 8080 inside the container to port 3000 outside the container, pass "3000:8080"
<code>:envs vars</code>	Set environment variable (a sequence of env var defs)
<code>:memory limit</code>	Memory limit
<code>:name name</code>	Assign a name to the container
<code>:quiet {true, false}</code>	Suppress the pull output
<code>:volumes vol</code>	Bind mount a volume (a sequence of volume defs)
<code>:workdir dir</code>	Working directory inside the container
<code>:args args</code>	Arguments passed to container process (a sequence of args or a string)

See also [cargo/start](#) / [cargo/stop](#) for a smarter way to start/stop a container.

```
;; Run an ArangoDB container (use bind mounts, very slow on macOSX)
(docker/run "arangodb/arangodb:3.10.10"
  :name "myapp"
  :publish [ "8529:8529" ]
  :detach true
  :envs [ "ARANGO_ROOT_PASSWORD=xxxxxx"
          "ARANGODB_OVERRIDE_DETECTED_TOTAL_MEMORY=8G"
          "ARANGODB_OVERRIDE_DETECTED_NUMBER_OF_CORES=1" ]
  :volumes [ "/Users/foo/arangodb/db:/var/lib/arangodb3"
             "/Users/foo/arangodb/apps:/var/lib/arangodb3-apps" ])

;; Run an ArangoDB container (use docker volume, faster than bind mount)
(do
  (docker/volume-create "arangodb-db")
  (docker/volume-create "arangodb-apps")
  (docker/run "arangodb/arangodb:3.10.10"
    :name "myapp"
    :publish [ "8529:8529" ]
    :detach true
    :envs [ "ARANGO_ROOT_PASSWORD=xxxxxx"
            "ARANGODB_OVERRIDE_DETECTED_TOTAL_MEMORY=8G"
            "ARANGODB_OVERRIDE_DETECTED_NUMBER_OF_CORES=1" ]
    :volumes [ "arangodb-db:/var/lib/arangodb3"
               "arangodb-apps:/var/lib/arangodb3-apps" ]
    :args [ "--database.auto-upgrade" ]))
```

SEE ALSO

[cargo/start](#)

Starts a container.

[docker/images](#)

List images.

[docker/ps](#)

List containers.

[docker/start](#)

Start a stopped container.

[docker/stop](#)

Stop a container.

[docker/rm](#)

Remove a container.

[docker/prune](#)

Remove all stopped containers.

[docker/exec](#)

Execute a command in a running container (always in non detached mode).

[docker/cp](#)

Copy files/folders between a container and the local filesystem

[docker/diff](#)

Inspect changes to files or directories on a container's filesystem.

[docker/pause](#)

Pause all processes within a container

[docker/unpause](#)

Unpause all processes within a container

[docker/cp](#)

Copy files/folders between a container and the local filesystem

[docker/logs](#)

Get the logs of a container

[docker/container-find-by-name](#)

Find all containers with a specified name

[docker/container-exists-with-name?](#)

Returns true if there is container with the specified name else false.

[docker/container-running-with-name?](#)

Checks if there is container with the specified name in 'running' state.

[docker/container-start-by-name](#)

Starts a container with the specified name.

[docker/container-stop-by-name](#)

Stops a container with the specified name.

[docker/container-remove-by-name](#)

Removes a container with the specified name.

[docker/container-status-by-name](#)

Returns the status of container with the specified name.

[docker/container-exec-by-name](#)

Execute a command in the running container with the specified name (always in non detached mode).

[docker/container-logs](#)

Returns the container logs.

[docker/container-purge-by-name](#)

Removes a container and its image.

[docker/container-image-info-by-name](#)

Returns the image info for a container given by its name.

[section](#)

[top](#)

docker/start

(docker/start container & options)

Start a stopped container.

Options:

:attach {true, false} Attach STDOUT/STDERR and forward signals

See also [cargo/start](#) / [cargo/stop](#) for a smarter way to start/stop a container.

([docker/start](#) "74789744g489")

SEE ALSO

[cargo/start](#)

Starts a container.

[docker/container-start-by-name](#)

Starts a container with the specified name.

[docker/stop](#)

Stop a container.

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/stop

(docker/stop container & options)

Stop a container.

Options:

:signal name Signal to send to the container

:time n Seconds to wait before killing the container

See also [cargo/start](#) / [cargo/stop](#) for a smarter way to start/stop a container.

([docker/stop](#) "74789744g489" :time 30)

SEE ALSO

[cargo/stop](#)

Stops a container

[docker/container-stop-by-name](#)

Stops a container with the specified name.

[docker/start](#)

Start a stopped container.

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/unpause

(`docker/unpause container`)

Unpause all processes within a container

(`docker/unpause "74789744g489"`)

SEE ALSO

[docker/pause](#)

Pause all processes within a container

[docker/ps](#)

List containers.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/version

(`docker/version & options`)

Returns the Docker version.

Options:

`:format f` Returns the output either as a string or as JSON data. The format is one of `{:string, :json}`

`:version v` Returns full (default), server, or client version. The version is one of `{:full, :server, :client}`

(`docker/version`)

(`docker/version :version :client`)

(`docker/version :version :server`)

(`docker/version :format :json`)

(`println (docker/version :format :string)`)

SEE ALSO

[docker/images](#)

List images.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/volume-create

(docker/volume-create vname & options)

Create a volume.

(docker/volume-create "hello")

SEE ALSO

[docker/volume-list](#)

List all the volumes known to Docker.

[docker/volume-inspect](#)

Inspects a volume.

[docker/volume-rm](#)

Remove a volume.

[docker/volume-prune](#)

Remove all unused local volumes. Unused local volumes are those which are not referenced by any containers. Removes both named and ...

[docker/volume-exists?](#)

Returns true if the volume with the specified name exists.

[section](#)

[top](#)

docker/volume-exists?

(docker/volume-exists? name)

Returns true if the volume with the specified name exists.

(docker/volume-exists? "hello")

SEE ALSO

[docker/volume-list](#)

List all the volumes known to Docker.

[section](#)

[top](#)

docker/volume-inspect

(docker/volume-inspect vname & options)

Inspects a volume.

Options:

`:format {string :json}` Returns the output either as a ascii or as JSON data

```
(docker/volume-inspect "hello")
```

SEE ALSO

[docker/volume-list](#)

List all the volumes known to Docker.

[docker/volume-create](#)

Create a volume.

[docker/volume-inspect](#)

Inspects a volume.

[docker/volume-prune](#)

Remove all unused local volumes. Unused local volumes are those which are not referenced by any containers. Removes both named and ...

[docker/volume-exists?](#)

Returns true if the volume with the specified name exists.

[section](#)

[top](#)

docker/volume-list

```
(docker/volume-list & options)
```

List all the volumes known to Docker.

Options:

`:quiet {true, false}` Only display volume names

`:format {table, json}` Returns the output either as a ascii table or as JSON data

```
(docker/volume-list)
```

SEE ALSO

[docker/volume-create](#)

Create a volume.

[docker/volume-inspect](#)

Inspects a volume.

[docker/volume-rm](#)

Remove a volume.

[docker/volume-prune](#)

Remove all unused local volumes. Unused local volumes are those which are not referenced by any containers. Removes both named and ...

[docker/volume-exists?](#)

Returns true if the volume with the specified name exists.

[docker/images](#)

List images.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docker/volume-prune

```
(docker/volume-prune )
```

Remove all unused local volumes. Unused local volumes are those which are not referenced by any containers. Removes both named and anonymous volumes!

```
(docker/volume-prune)
```

SEE ALSO

[docker/volume-list](#)

List all the volumes known to Docker.

[docker/volume-create](#)

Create a volume.

[docker/volume-inspect](#)

Inspects a volume.

[docker/volume-rm](#)

Remove a volume.

[docker/volume-exists?](#)

Returns true if the volume with the specified name exists.

[section](#)[top](#)

docker/volume-rm

```
(docker/volume-rm name)
```

Remove a volume.

```
(docker/volume-remove "hello")
```

SEE ALSO

[docker/volume-list](#)

List all the volumes known to Docker.

[docker/volume-create](#)

Create a volume.

[docker/volume-inspect](#)

Inspects a volume.

[docker/volume-prune](#)

Remove all unused local volumes. Unused local volumes are those which are not referenced by any containers. Removes both named and ...

[docker/volume-exists?](#)

Returns true if the volume with the specified name exists.

[section](#)[top](#)

docker/wait

```
(docker/wait & containers)
```

Block until one or more containers stop, then return their exit codes

```
(docker/wait "74789744g4892" "2341428e53535")
```

SEE ALSO

[docker/ps](#)

List containers.

[docker/rm](#)

Remove a container.

[docker/run](#)

Create and run a new container from an image.

[section](#)

[top](#)

docoll

```
(docoll f coll)
```

Applies `f` to the items of the collection presumably for side effects. Returns `nil`.

If `coll` is a lazy sequence, `docoll` iterates over the lazy sequence and realizes value by value while calling function `f` on the realized values.

```
(docoll #(println %) [1 2 3 4])
```

```
1
2
3
4
=> nil
```

```
(docoll (fn [[k v]] (println (pr-str k v)))
        {:a 1 :b 2 :c 3 :d 4}))
```

```
:a 1
:b 2
:c 3
:d 4
=> nil
```

```
;; docoll all elements of a queue. calls (take! queue) to get the
;; elements of the queue.
;; note: use nil to mark the end of the queue otherwise docoll will
;;       block forever!
(let [q (conj! (queue) 1 2 3 nil)]
  (docoll println q))
```

```
1
2
3
=> nil
```

```
;; lazy sequence
(let [q (conj! (queue) 1 2 3 nil)]
  (defn f []
    (let [v (poll! q)]
      (println "Producing " v)
      v))
  (docoll #(println "Collecting" %))
```

```
      (lazy-seq f)))
Producing 1
Collecting 1
Producing 2
Collecting 2
Producing 3
Collecting 3
Producing nil
=> nil
```

SEE ALSO

[mapv](#)

Returns a vector consisting of the result of applying f to the set of first items of each coll, followed by applying f to the set of ...

[section](#)

[top](#)

done?

```
(done? f)
```

Returns true if the future or promise is done otherwise false

```
(do
  (def wait (fn [] (sleep 200) 100))
  (let [f (future wait)]
    (sleep 50)
    (printf "After 50ms: done=%b\n" (done? f))
    (sleep 300)
    (printf "After 300ms: done=%b\n" (done? f)))))
After 50ms: done=false
After 300ms: done=true
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[promise](#)

Returns a promise object that can be read with deref, and set, once only, with deliver. Calls to deref prior to delivery will block, ...

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[cancel](#)

Cancels a future or a promise

[cancelled?](#)

Returns true if the future or promise is cancelled otherwise false

[section](#)

[top](#)

dorun

```
(dorun count expr)
```

Runs the `expr` count times in the most effective way. It's main purpose is supporting benchmark tests. Returns the expression result of the last invocation.

Note:

For best performance enable `macroexpand-on-load` ! The expression is evaluated for every run. Alternatively a zero or one arg function referenced by a symbol can be passed:

```
(let [f (fn [] (+ 1 1))]  
      (dorun 10 f))
```

When passing a one arg function `dorun` passes the incrementing counter value (0..N) to the function:

```
(let [f (fn [x] (+ x 1))]  
      (dorun 10 f))
```

```
(dorun 10 (+ 1 1))  
=> 2
```

[section](#)

[top](#)

doseq

```
(doseq seq-exprs & body)
```

Repeatedly executes `body` (presumably for side-effects) with bindings and filtering as provided by `list-comp`. Does not retain the head of the sequence. Returns `nil`.

Supported modifiers are: `:when` predicate

```
(doseq [x (range 10)] (print x))  
0123456789  
=> nil
```

```
(doseq [x (range 10)] (print x) (print "-"))  
0-1-2-3-4-5-6-7-8-9-  
=> nil
```

```
(doseq [x (range 5)] (print (* x 2)))  
02468  
=> nil
```

```
(doseq [x (range 10) :when (odd? x)] (print x))  
13579  
=> nil
```

```
(doseq [x (range 10) :when (odd? x)] (print (* x 2)))  
26101418  
=> nil
```

```
(doseq [x [1 2 3] y [1 2 3]] (println [x y]))  
[1 1]  
[1 2]  
[1 3]  
[2 1]  
[2 2]  
[2 3]  
[3 1]  
[3 2]  
[3 3]  
=> nil
```

```
(doseq [[x y] [[0 1] [1 2]]] (println [x y]))
[0 1]
[1 2]
=> nil

(doseq [[k v] {:a 1 :b 2}] (println [k v]))
[:a 1]
[:b 2]
=> nil

(doseq [[c vals] (group-by count ["a" "as" "asd" "aa" "asdf" "qwer"])]
  (println c vals))
1 [a]
2 [as aa]
3 [asd]
4 [asdf qwer]
=> nil
```

SEE ALSO

[for](#)

List comprehension. Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and ...

[list-comp](#)

List comprehension. Takes a vector of one or more binding-form or collection-expr pairs, each followed by zero or more modifiers, and ...

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[section](#)

[top](#)

dotimes

```
(dotimes bindings & body)
```

Repeatedly executes body with name bound to integers from 0 through n-1.

```
(dotimes [n 3] (println (str "n is " n)))
n is 0
n is 1
n is 2
=> nil
```

SEE ALSO

[repeat](#)

Returns a lazy sequence of x values or a collection with the value x repeated n times.

[repeatedly](#)

Takes a function of no args, presumably with side effects, and returns a collection of n calls to it

[doseq](#)

Repeatedly executes body (presumably for side-effects) with bindings and filtering as provided by list-comp. Does not retain the head ...

[list-comp](#)

List comprehension. Takes a vector of one or more binding-form or collection-expr pairs, each followed by zero or more modifiers, and ...

[section](#)

[top](#)

doto

```
(doto x & forms)
```

Evaluates x then calls all of the methods and functions with the value of x supplied at the front of the given arguments. The forms are evaluated in order. Returns x.

```
(doto (. :java.util.HashMap :new)
      (. :put :a 1)
      (. :put :b 2))
=> {"a" 1 "b" 2}
```

[section](#)[top](#)

double

```
(double x)
```

Converts to double

```
(double 1)
=> 1.0

(double nil)
=> 0.0

(double false)
=> 0.0

(double true)
=> 1.0

(double 1.2)
=> 1.2

(double 1.2F)
=> 1.2000000476837158

(double 1.2M)
=> 1.2

(double "1.2")
=> 1.2
```

[section](#)[top](#)

double-array

```
(double-array coll)
(double-array len)
(double-array len init-val)
```

Returns an array of Java primitive doubles containing the contents of coll or returns an array with the given length and optional init value.

To create an array of :java.lang.Double use:

```
(make-array :java.lang.Long 3)
```

```
(double-array '(1.0 2.0 3.0))  
=> [1.0, 2.0, 3.0]
```

```
(double-array '(1I 2 3.2 3.56M))  
=> [1.0, 2.0, 3.2, 3.56]
```

```
(double-array 10)  
=> [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
(double-array 10 42.0)  
=> [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0]
```

SEE ALSO

[java-double-list](#)

Converts a Venice list/vector to a Java Double list

[section](#)

[top](#)

double?

```
(double? n)
```

Returns true if n is a double

```
(double? 4.0)  
=> true
```

```
(double? 4.0F)  
=> false
```

```
(double? 3)  
=> false
```

```
(double? 3I)  
=> false
```

```
(double? 3.0M)  
=> false
```

```
(double? true)  
=> false
```

```
(double? nil)  
=> false
```

```
(double? {})  
=> false
```

[section](#)

[top](#)

drop

```
(drop n coll)
```

Returns a collection of all but the first n items in coll.
Returns a stateful transducer when no collection is provided.

```
(drop 3 [1 2 3 4 5])  
=> [4 5]
```

```
(drop 10 [1 2 3 4 5])  
=> []
```

[section](#)[top](#)

drop-last

```
(drop-last n coll)
```

Return a sequence of all but the last n items in coll.
Returns a stateful transducer when no collection is provided.

```
(drop-last 3 [1 2 3 4 5])  
=> [1 2]
```

```
(drop-last 10 [1 2 3 4 5])  
=> []
```

[section](#)[top](#)

drop-while

```
(drop-while predicate coll)
```

Returns a list of the items in coll starting from the first item for which `(predicate item)` returns logical false.
Returns a stateful transducer when no collection is provided.

```
(drop-while neg? [-2 -1 0 1 2 3])  
=> [0 1 2 3]
```

[section](#)[top](#)

empty

```
(empty coll)
```

Returns an empty collection of the same category as coll, or nil if coll is nil. If the collection is mutable clears the collection and returns the emptied collection.


```
(empty {:a 1})  
=> {}
```

```
(empty [1 2])  
=> []
```

```
(empty '(1 2))  
=> ()
```

[section](#)[top](#)

empty-to-nil

```
(empty-to-nil x)
```

Returns nil if x is empty

```
(empty-to-nil "")  
=> nil
```

```
(empty-to-nil [])  
=> nil
```

```
(empty-to-nil '())  
=> nil
```

```
(empty-to-nil {})  
=> nil
```

[section](#)[top](#)

empty?

```
(empty? x)
```

Returns true if x is empty. Accepts strings, collections and bytebufs.

```
(empty? {})  
=> true
```

```
(empty? [])  
=> true
```

```
(empty? '())  
=> true
```

```
(empty? nil)  
=> true
```

```
(empty? "")  
=> true
```

SEE ALSO

not-empty?

Returns true if x is not empty. Accepts strings, collections and bytebufs.

[section](#)[top](#)

entries

```
(entries m)
```

Returns a collection of the map's entries.

```
(entries {:a 1 :b 2 :c 3})
=> ([:a 1] [:b 2] [:c 3])

(let [e (entries {:a 1 :b 2 :c 3})]
  (println (map key e))
  (println (map val e)))
(:a :b :c)
(1 2 3)
=> nil

;; compare to 'into'
(let [e (into [] {:a 1 :b 2 :c 3})]
  (println (map first e))
  (println (map second e)))
(:a :b :c)
(1 2 3)
=> nil
```

SEE ALSO

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[key](#)

Returns the key of the map entry.

[val](#)

Returns the val of the map entry.

[keys](#)

Returns a collection of the map's keys.

[vals](#)

Returns a collection of the map's values.

[map-entry](#)

Creates a new map entry

[section](#)[top](#)

enum?

```
(enum? class)
```

Returns true if class is a Java *enum*.

Get all values of a Java *enum*:

```
(. :java.time.Month :values)
```

Get a Java *enum* value:

```
(let [jan (. :java.time.Month :JANUARY)]  
  (. :java.time.LocalDate :of 1994 jan 21))
```

This can be simplified to:

```
(. :java.time.LocalDate :of 1994 :JANUARY 21)
```

Equality testing *enums* `(let [date (. :java.time.LocalDate :of 2026 1 1) month (. date :getMonth)] (enum? (class-of month)) ;;
true (month is a java enum) (== month :JANUARY) ;; true (value equality) (== :JANUARY month) ;; true (value equality)`

```
(enum? :java.time.Month)  
=> true
```

[section](#)

[top](#)

eval

```
(eval form)
```

Evaluates the form data structure (not text!) and returns the result.

```
(eval '(let [a 10] (+ 3 4 a)))  
=> 17
```

```
(eval (list + 1 2 3))  
=> 6
```

```
(eval (read-string "(+ 1 2)"))  
=> 3
```

```
(let [s "(+ 2 x)" x 10]  
  (eval (read-string s)))  
=> 12
```

SEE ALSO

[read-string](#)

Reads Venice source from a string and transforms its content into a Venice data structure, following the rules of the Venice syntax.

[section](#)

[top](#)

even?

```
(even? n)
```

Returns true if n is even, throws an exception if n is not an integer

```
(even? 4)  
=> true
```

```
(even? 3)  
=> false
```

```
(even? 3I)
=> false
```

SEE ALSO

[odd?](#)

Returns true if n is odd, throws an exception if n is not an integer

[section](#)

[top](#)

every-pred

```
(every-pred p1 & p)
```

Takes a set of predicates and returns a function f that returns true if all of its composing predicates return a logical true value against all of its arguments, else it returns false. Note that f is short-circuiting in that it will stop execution on the first argument that triggers a logical false result against the original predicates.

```
((every-pred number?) 1)
=> true
```

```
((every-pred number?) 1 2)
=> true
```

```
((every-pred number? even?) 2 4 6)
=> true
```

[section](#)

[top](#)

every?

```
(every? pred coll)
```

Returns true if coll is a collection and the predicate is true for all collection items, false otherwise.

```
(every? number? nil)
=> false
```

```
(every? number? [])
=> true
```

```
(every? number? [1 2 3 4])
=> true
```

```
(every? number? [1 2 3 :a])
=> false
```

```
(every? #(>= % 10) [10 11 12])
=> true
```

SEE ALSO

[any?](#)

Returns true if the predicate is true for at least one collection item, false otherwise.

not-any?

Returns false if the predicate is true for at least one collection item, true otherwise

not-every?

Returns true if coll is a collection and the predicate is not true for all collection items, false otherwise.

[section](#)[top](#)

ex

```
(ex class)
(ex class args*)
```

Creates an exception of type *class* with optional *args*. The *class* must be a subclass of :java.lang.Exception

The exception types:

- :java.lang.Exception
- :java.lang.RuntimeException
- :com.github.jlangch.venice.VncException
- :com.github.jlangch.venice.ValueException

are imported implicitly so its alias :Exception, :RuntimeException, :VncException, and :ValueException can be used.

Checked vs unchecked exceptions

All exceptions in Venice are *unchecked*.

If *checked* exceptions are thrown in Venice they are immediately wrapped in a :RuntimeException before being thrown!

If Venice catches a *checked* exception from a Java Interop call it wraps it in a :RuntimeException before handling it by the catch block selectors.

```
(try
  (throw (ex :VncException)))
(catch :VncException e "caught :VncException"))
=> "caught :VncException"

(try
  (throw (ex :RuntimeException "#test")))
(catch :Exception e
  "msg: ~(ex-message e)")
=> "msg: #test"

(try
  (throw (ex :ValueException 100)))
(catch :ValueException e
  "value: ~(ex-value e)")
=> "value: 100"

(do
  (defn throw-ex-with-cause []
    (try
      (throw (ex :java.io.IOException "I/O failure"))
      (catch :Exception e
        (throw (ex :VncException "failure" (ex-cause e))))))
    (try
      (throw-ex-with-cause)
      (catch :Exception e
        "msg: ~(ex-message e), cause: ~(ex-message (ex-cause e)))))
=> "msg: failure, cause: I/O failure"
```

SEE ALSO

throw

Throws an exception.

try

Exception handling: try - catch - finally

try-with

try-with-resources allows the declaration of resources to be used in a try block with the assurance that the resources will be closed ...

ex?

Returns true if x is a an instance of :java.lang.Throwable

ex-venice?

Returns true if x is a an instance of :VncException

[section](#)

[top](#)

ex-cause

```
(ex-cause x)
```

Returns the exception cause or nil

```
(ex-cause (ex :VncException "a message" (ex :RuntimeException "..cause..")))  
=> java.lang.RuntimeException: ..cause..
```

```
(ex-cause (ex :VncException "a message"))  
=> nil
```

SEE ALSO

ex

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

ex-message

Returns the message of the exception

ex-value

Returns the value associated with a :ValueException or nil if the exception is not a :ValueException

[section](#)

[top](#)

ex-java-stacktrace

```
(ex-java-stacktrace x)  
(ex-java-stacktrace x format)
```

Returns the Java stacktrace for an exception.

The optional format (:string or :list) controls the format of the returned stacktrace. The default format is :string.

```
(println (ex-java-stacktrace (ex :RuntimeException "message")))
```

```
(println (ex-java-stacktrace (ex :VncException "message") :list))
```

SEE ALSO

ex

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[ex-venice-stacktrace](#)

Returns the Venice stacktrace for an exception or nil if the exception is not a venice exception.

[section](#)

[top](#)

ex-message

```
(ex-message x)
```

Returns the message of the exception

```
(ex-message (ex :VncException "a message"))  
=> "a message"
```

```
(ex-message (ex :RuntimeException))  
=> nil
```

SEE ALSO

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[ex-cause](#)

Returns the exception cause or nil

[ex-value](#)

Returns the value associated with a :ValueException or nil if the exception is not a :ValueException

[section](#)

[top](#)

ex-value

```
(ex-value x)
```

Returns the value associated with a :ValueException or nil if the exception is not a :ValueException

```
(ex-value (ex :ValueException [10 20]))  
=> (10 20)
```

```
(ex-value (ex :RuntimeException))  
=> nil
```

SEE ALSO

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[ex-message](#)

Returns the message of the exception

[ex-cause](#)

Returns the exception cause or nil

[section](#)

[top](#)

ex-venice-stacktrace

```
(ex-venice-stacktrace x)
(ex-venice-stacktrace x format)
```

Returns the Venice stacktrace for an exception or nil if the exception is not a venice exception.

The optional format (:string or :list) controls the format of the returned stacktrace. The default format is :string.

```
(println (ex-venice-stacktrace (ex :ValueException [10 20])))
Exception in thread "main" ValueException:

[Callstack]
  at: ex (example: line 70, col 43)
=> nil

(println (ex-venice-stacktrace (ex :RuntimeException "message")))
nil
=> nil

(println (ex-venice-stacktrace (ex :ValueException [10 20]) :list))
({:fn ex :file example :line 70 :col 43})
=> nil
```

SEE ALSO

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[ex-java-stacktrace](#)

Returns the Java stacktrace for an exception.

[section](#)

[top](#)

ex-venice?

```
(ex-venice? x)
```

Returns true if x is a an instance of :VncException

```
(ex-venice? (ex :VncException))
=> true

(ex-venice? (ex :RuntimeException))
=> false
```

SEE ALSO

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[ex?](#)

Returns true if x is a an instance of :java.lang.Throwable

[section](#)

[top](#)

ex?

```
(ex? x)
```

Returns true if x is a an instance of :java.lang.Throwable

```
(ex? (ex :RuntimeException))  
=> true
```

SEE ALSO

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[ex-venice?](#)

Returns true if x is a an instance of :VncException

[section](#)

[top](#)

excel/add-area-chart

```
(add-area-chart sheet chart-title  
                  chart-addr-range  
                  legend-position  
                  category-axis-title  
                  category-axis-position  
                  value-axis-title  
                  value-axis-position  
                  three-dimensional?  
                  categories-addr-range  
                  series)
```

Adds an area chart.

Arguments:

chart-title	The chart title
chart-addr-range	The chart position in the Excel
legend-position	The legend position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
category-axis-title	The category axis title
category-axis-position	The category axis position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
value-axis-title	The value axis title
value-axis-position	The value axis position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
three-dimensional?	Render in 3D: <code>true</code> or <code>false</code>
categories-addr-range	The category names in the Excel
series	The value series data. 1 to N series

```
(do  
  (load-module :excel)  
  (let [wbook (excel/create :xlsx)  
        sheet (excel/add-sheet wbook "Sheet 1")  
        data [ ["Year" "Bears" "Dolphins" "Whales"]  
                ["2017" 8 150 80 ]  
                ["2018" 54 77 54 ]  
                ["2019" 93 32 100 ]  
                ["2020" 116 11 76 ]
```

```

["2021"      137      6      93 ]
["2022"      184      1      72 ]]]
(excel/write-data sheet data)

(excel/add-area-chart sheet
  "Bears Population"
  (excel/cell-address-range 10 25 1 7)
  :RIGHT
  "Year"
  :BOTTOM
  "Population"
  :LEFT
  false
  (excel/cell-address-range 2 7 1 1)
  [ (excel/area-data-series
    "Bears"
    (excel/cell-address-range 2 7 2 2)) ])

(excel/write->file wbook "sample.xlsx"))

```

SEE ALSO

[excel/add-line-chart](#)

Adds a line chart.

[excel/add-bar-chart](#)

Adds a bar chart.

[excel/add-pie-chart](#)

Adds a pie chart.

[excel/area-data-series](#)

Build an area chart data series

[excel/cell-address-range](#)

Build a cell address range

[section](#) [top](#)

excel/add-bar-chart

```

(add-bar-chart sheet chart-title
  chart-addr-range
  legend-position
  category-axis-title
  category-axis-position
  value-axis-title
  value-axis-position
  three-dimensional?
  direction-bar?
  grouping
  vary-colors?
  categories-addr-range
  series)

```

Adds a bar chart.

Arguments:

chart-title	The chart title
chart-addr-range	The chart position in the Excel
legend-position	The legend position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>

category-axis-title	The category axis title
category-axis-position	The category axis position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
value-axis-title	The value axis title
value-axis-position	The value axis position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
three-dimensional?	Render in 3D: <code>true</code> or <code>false</code>
direction-bar?	Render as horizontal bars or vertical columns: <code>true</code> or <code>false</code>
grouping	Bar grouping: <code>:STANDARD</code> , <code>:CLUSTERED</code> , <code>:STACKED</code> , <code>:PERCENT_STACKED</code>
vary-colors?	Vary the colors: <code>true</code> or <code>false</code>
categories-addr-range	The category names in the Excel
series	The value series data. 1 to N series

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")
        data [
          ["Year" "Bears" "Dolphins" "Whales"]
          ["2017" 8 150 80 ]
          ["2018" 54 77 54 ]
          ["2019" 93 32 100 ]
          ["2020" 116 11 76 ]
          ["2021" 137 6 93 ]
          ["2022" 184 1 72 ]]]
    (excel/write-data sheet data)

    (excel/add-bar-chart sheet
      "Bears Population"
      (excel/cell-address-range 10 25 1 7)
      :RIGHT
      "Year"
      :BOTTOM
      "Population"
      :LEFT
      false
      false
      :STANDARD
      false
      (excel/cell-address-range 2 7 1 1)
      [ (excel/bar-data-series
          "Bears"
          (excel/cell-address-range 2 7 2 2))
        (excel/bar-data-series
          "Dolphins"
          (excel/cell-address-range 2 7 3 3))
        (excel/bar-data-series
          "Whales"
          (excel/cell-address-range 2 7 4 4)) ])

    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/add-line-chart](#)
Adds a line chart.

[excel/add-area-chart](#)
Adds an area chart.

[excel/add-pie-chart](#)
Adds a pie chart.

[excel/bar-data-series](#)

Build a bar chart data series

[excel/cell-address-range](#)

Build a cell address range

[section](#)

[top](#)

excel/add-column

(add-column sheet title)

(add-column sheet title options)

Defines a column with optional attributes on the sheet.

Note: The column cell value is just read from the passed tabular dataset. If there is any mapping or conversion needed it has to be applied to the dataset before writing it to the sheet!

Options:

:id id	a column id
:field f	a field, e.g. :first-name
:width n	width in points, e.g. 100
:skip s	skip column, e.g. true, false
:header-style r	style name for header row, e.g. :header
:body-style r	style name for body rows, e.g. :body
:footer-style r	style name for footer row, e.g. :footer
:footer-value v	explicit text or numeric value for the column's footer cell, e.g. "done", 10000.00M, nil
:footer-aggregate e	aggregation mode for the column's footer cell value, e.g. {:min, :max, :avg, :sum, :none}

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :weight 70.5 }
                {:first "Sue" :last "Ford" :weight 54.2 } ]
        wbook (excel/create :xlsx)]
    (excel/add-font wbook :header { :bold true })
    (excel/add-style wbook :header { :font :header
                                     :bg-color :GREY_25_PERCENT
                                     :h-align :center })
    (excel/add-style wbook :weight { :format, "#,##0.0"
                                     :h-align :right })

    (let [sheet (excel/add-sheet wbook "Sheet 1"
                                { :no-header-row false
                                  :default-header-style :header })]
      (excel/add-column sheet "First Name" { :field :first })
      (excel/add-column sheet "Last Name" { :field :last })
      (excel/add-column sheet "Weight" { :field :weight
                                         :body-style :weight })

      (excel/write-items sheet data)
      (excel/auto-size-columns sheet)
      (excel/write->file wbook "sample.xlsx"))))
```

SEE ALSO

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[section](#)[top](#)

excel/add-conditional-bg-color

```
(add-conditional-bg-color sheet
  rule
  color-html
  region-first-row
  region-last-row
  region-first-col
  region-last-col)
```

Add a conditional background color

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/add-conditional-bg-color sheet "ISBLANK(A1)" "#CC636A" 1 2 2 2)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)[top](#)

excel/add-conditional-border

```
(add-conditional-border sheet
  rule
  border-top-style
  border-right-style
  border-bottom-style
  border-left-style
  border-top-color-html
  border-right-color-html
  border-bottom-color-html
  border-left-color-html
  region-first-row
  region-last-row
  region-first-col
  region-last-col)
```

Add a conditional border

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 45)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/add-conditional-border sheet "C1 > 30"
      :thin :thin :thin :thin
      nil nil nil nil
      1 1 3 3)
    (excel/add-conditional-border sheet "C2 > 30"
      :thin :thin :thin :thin
      nil nil nil nil
      2 2 3 3)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/add-conditional-font-color

```
(add-conditional-font-color sheet
                             rule
                             color-html
                             region-first-row
                             region-last-row
                             region-first-col
                             region-last-col)
```

Add a conditional font color

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 45)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/add-conditional-font-color sheet "C1 > 30" "#CC636A" 1 1 3 3)
    (excel/add-conditional-font-color sheet "C2 > 30" "#CC636A" 2 2 3 3)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/add-email-hyperlink

```
(add-email-hyperlink sheet row col text url)
```

Adds an email hyperlink to a cell

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-font wbook :hyperlink { :underline true
                                        :color :BLUE })
    (excel/add-style wbook :hyperlink { :font :hyperlink })
    (excel/write-values sheet 1 1 "John" "Doe")
    (excel/write-values sheet 2 1 "Sue" "Ford")
    (excel/add-email-hyperlink sheet 1 3 "john.doe@foo.org" "john.doe@foo.org")
    (excel/add-email-hyperlink sheet 2 3 "sue.ford@foo.org" "sue.ford@foo.org")
    (excel/cell-style sheet 1 3 :hyperlink)
    (excel/cell-style sheet 2 3 :hyperlink)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/remove-hyperlink](#)

Remove a cell comment

[excel/add-url-hyperlink](#)

Adds an URL hyperlink to a cell

[section](#)

[top](#)

excel/add-font

```
(add-font wbook font-id)
```

```
(add-font wbook font-id options)
```

Add font with optional attributes to an Excel.

Options:

:name s	font name, e.g. 'Arial'
:height n	height in points, e.g. 12
:bold b	bold, e.g. true, false
:italic b	italic, e.g. true, false
:underline b	underline, e.g. true, false
:color c	color, either an Excel indexed color or a HTML color, e.g. :BLUE, "#00FF00" note: only XLSX supports 24 bit colors

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
               {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)]
    (excel/add-font wbook :header { :height 12
                                    :bold true
                                    :italic false
                                    :underline false
```



```

                                :color :BLUE })
(excel/add-style wbook :header { :font :header })

(let [sheet (excel/add-sheet wbook "Sheet 1"
                             { :no-header-row false
                               :default-header-style :header })]
  (excel/add-column sheet "First Name" { :field :first })
  (excel/add-column sheet "Last Name" { :field :last })
  (excel/add-column sheet "Age" { :field :age })
  (excel/write-items sheet data)
  (excel/auto-size-columns sheet)
  (excel/write->file wbook "sample.xlsx"))))

```

SEE ALSO

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[section](#)
[top](#)

excel/add-image

```

(add-image sheet row col data type)
(add-image sheet row col data type scale-X scale-Y)

```

Adds an image given by its binary data (a `bytebuf`) to a specific anchor cell given by its row and col. Optionally the image can be scaled by an X and Y axis factor. The image types `:PNG` and `:JPEG` are supported.

```

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")
        image "com/github/jlangch/venice/images/venice.png"
        data (io/load-classpath-resource image)]
    (excel/add-image sheet 2 2 data :PNG)
    (excel/write->file wbook "sample.xlsx")))

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")
        image "com/github/jlangch/venice/images/venice.png"
        data (io/load-classpath-resource image)]
    (excel/add-image sheet 2 2 data :PNG 0.5 0.5)
    (excel/write->file wbook "sample.xlsx")))

```

SEE ALSO

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/add-line-chart

```
(add-line-chart sheet chart-title
                  chart-addr-range
                  legend-position
                  category-axis-title
                  category-axis-position
                  value-axis-title
                  value-axis-position
                  three-dimensional?
                  vary-colors?
                  categories-addr-range
                  series)
```

Adds a line chart.

Arguments:

chart-title	The chart title
chart-addr-range	The chart position in the Excel
legend-position	The legend position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
category-axis-title	The category axis title
category-axis-position	The category axis position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
value-axis-title	The value axis title
value-axis-position	The value axis position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
three-dimensional?	Render in 3D: <code>true</code> or <code>false</code>
vary-colors?	Vary the colors: <code>true</code> or <code>false</code>
categories-addr-range	The category names in the Excel
series	The value series data. 1 to N series

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")
        data  [ ["Year"   "Bears" "Dolphins" "Whales"]
                  ["2017"   8       150       80 ]
                  ["2018"  54       77       54 ]
                  ["2019"  93       32      100 ]
                  ["2020" 116       11       76 ]
                  ["2021" 137        6       93 ]
                  ["2022" 184        1       72 ]]]
    (excel/write-data sheet data)

    (excel/add-line-chart sheet
                          "Wildlife Population"
                          (excel/cell-address-range 10 25 1 10)
                          :RIGHT
                          "Year"
                          :BOTTOM)
```

```

    "Population"
    :LEFT
    false
    true
    (excel/cell-address-range 2 7 1 1)
    [ (excel/line-data-series
      "Bears"
      true
      :CIRCLE
      (excel/cell-address-range 2 7 2 2))
      (excel/line-data-series
      "Dolphins"
      true
      :CIRCLE
      (excel/cell-address-range 2 7 3 3))
      (excel/line-data-series
      "Whales"
      true
      :CIRCLE
      (excel/cell-address-range 2 7 4 4)) ] )

    (excel/write->file wbook "sample.xlsx"))))

```

SEE ALSO

[excel/add-bar-chart](#)

Adds a bar chart.

[excel/add-area-chart](#)

Adds an area chart.

[excel/add-pie-chart](#)

Adds a pie chart.

[excel/line-data-series](#)

Build a line chart data series

[excel/cell-address-range](#)

Build a cell address range

[section](#)

[top](#)

excel/add-merge-region

```
(add-merge-region sheet row-from row-to col-from col-to)
```

Add a merge region to the sheet.

```

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Population")]
    (excel/col-width sheet 2 70)
    (excel/col-width sheet 3 70)
    (excel/add-merge-region sheet 2 2 2 3)
    (excel/write-value sheet 2 2 "Contry Population")
    (excel/write-value sheet 3 2 "Country")
    (excel/write-value sheet 3 3 "Population")
    (excel/write-value sheet 4 2 "Germany")
    (excel/write-value sheet 4 3 83_783_942)
    (excel/write-value sheet 5 2 "Italy")
    (excel/write-value sheet 5 3 60_461_826)

```

```
(excel/write-value sheet 6 2 "Austria")
(excel/write-value sheet 6 3 9_006_398)
(excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[section](#)

[top](#)

excel/add-pie-chart

```
(add-pie-chart sheet chart-title
                chart-addr-range
                legend-position
                three-dimensional?
                vary-colors?
                categories-addr-range
                series)
```

Adds a pie chart.

Arguments:

chart-title	The chart title
chart-addr-range	The chart position in the Excel
legend-position	The legend position: <code>:TOP</code> , <code>:TOP_RIGHT</code> , <code>:RIGHT</code> , <code>:BOTTOM</code> , <code>:LEFT</code>
three-dimensional?	Render in 3D: <code>true</code> or <code>false</code>
vary-colors?	Vary the colors: <code>true</code> or <code>false</code>
categories-addr-range	The category names in the Excel
series	The value series data. 1 series required

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")
        data  [ ["Year"   "Bears" "Dolphins" "Whales"]
                ["2017"    8       150       80 ]
                ["2018"   54       77       54 ]
                ["2019"   93       32      100 ]
                ["2020"  116       11       76 ]
                ["2021"  137        6       93 ]
                ["2022"  184        1       72 ]]]
    (excel/write-data sheet data)

    (excel/add-pie-chart sheet
                        "Wildlife Population 2017"
                        (excel/cell-address-range 10 25 1 7)
                        :RIGHT
                        false
                        true
                        (excel/cell-address-range 1 1 2 4)
                        [ (excel/pie-data-series
                          (excel/cell-address-range 2 2 2 4)) ])

    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/add-line-chart](#)

Adds a line chart.

[excel/add-bar-chart](#)

Adds a bar chart.

[excel/add-area-chart](#)

Adds an area chart.

[excel/pie-data-series](#)

Build a pie chart data series

[excel/cell-address-range](#)

Build a cell address range

[section](#)

[top](#)

excel/add-sheet

(add-sheet wbook title)

(add-sheet wbook title options)

Adds a sheet with optional attributes to an Excel.

Options:

:no-header-row b	without header row, e.g. true, false
:default-column-width n	default column width in points, e.g. 100
:default-header-style s	default header style, e.g. :header
:default-body-style s	default body style, e.g. :body
:default-footer-style s	default footer style, e.g. :footer
:merged-region r	merged region [row-from row-to col-from col-to], e.g. [1 1 4 10]
:display-zeros b	display zeros, e.g. true, false. Defines if a cell should show 0 (zero) when containing zero value. When false, cells with zero value appear blank instead of showing the number zero.

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)]
    (excel/add-font wbook :bold { :bold true })
    (excel/add-font wbook :italic { :italic true })
    (excel/add-style wbook :header { :font :bold })
    (excel/add-style wbook :body { :font :italic })
    (excel/add-style wbook :footer { :font :bold })

    (let [sheet (excel/add-sheet wbook "Sheet 1"]
```

```

        { :no-header-row false
          :default-column-width 100
          :default-header-style :header
          :default-body-style :body
          :default-footer-style :footer
          :display-zeros true}}]
(excel/add-column sheet "First Name" { :field :first })
(excel/add-column sheet "Last Name" { :field :last })
(excel/add-column sheet "Age" { :field :age })
(excel/write-items sheet data)
(excel/auto-size-column sheet 1)
(excel/auto-size-column sheet 2)
(excel/auto-size-column sheet 3)
(excel/write->file wbook "sample.xlsx"))))

```

SEE ALSO

[excel/add-column](#)

Defines a column with optional attributes on the sheet.

[excel/protect-sheet](#)

Protect the sheet.

[excel/add-merge-region](#)

Add a merge region to the sheet.

[section](#)

[top](#)

excel/add-style

```

(add-style wbook style-id)
(add-style wbook style-id options)

```

Add a style with optional attributes to an Excel.

Options:

:format s	cell format, e.g. "#0" Default formats: - long: "###0" - integer: "###0" - float: "###0.00" - double: "###0.00" - date: "d.m.yyyy" - datetime: "d.m.yyyy hh:mm:ss"
:font r	font name, e.g. :header
:bg-color c	background color, either an Excel indexed color or a HTML color, e.g. :PLUM, "#00FF00" Note: only XLSX supports 24 bit colors
:wrap-text b	wrap text, e.g. true, false
:h-align e	horizontal alignment {:left, :center, :right}
:v-align e	vertical alignment {:top, :middle, :bottom}
:rotation r	rotation angle [degree], e.g. 45
:border-top s	border top style, e.g. :thin
:border-right s	border right style, e.g. :none
:border-bottom s	border bottom style, e.g. :thin
:border-left s	border left style, e.g. :none

Available border styles:

:none :dotted :medium-dashed :medium-dash-dot-dot

:thin	:thick	:dash-dot	:slanted-dash-dot
:medium	:double	:medium-dash-dot	
:dashed	:hair	:dash-dot-dot	

```
(do
  (load-module :excel)
  (let [data [ { :first "John" :last "Doe" :weight 70.5 }
                { :first "Sue" :last "Ford" :weight 54.2 } ]
        wbook (excel/create :xlsx)]
    (excel/add-font wbook :header { :bold true })
    (excel/add-style wbook :header { :font :header
                                      :bg-color :GREY_25_PERCENT
                                      :h-align :center
                                      :rotation 0
                                      :border-top :thin
                                      :border-bottom :thin })
    (excel/add-style wbook :weight { :format "#,##0.0"
                                      :h-align :right })

    (let [sheet (excel/add-sheet wbook "Sheet 1"
                                  { :no-header-row false
                                    :default-header-style :header })]
      (excel/add-column sheet "First Name" { :field :first })
      (excel/add-column sheet "Last Name" { :field :last })
      (excel/add-column sheet "Weight" { :field :weight
                                          :body-style :weight })

      (excel/write-items sheet data)
      (excel/auto-size-columns sheet)
      (excel/write->file wbook "sample.xlsx"))))
```

SEE ALSO

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[section](#)

[top](#)

excel/add-text-data-validation

```
(add-text-data-validation sheet
  strings
  empty-cell-allowed?
  err-title
  err-text
  region-first-row
  region-last-row
  region-first-col
  region-last-col)
```

Adds a text enumeration validation to a cell region in a sheet

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" "male" 28))
```

```
(excel/write-values sheet 2 1 "Sue" "Ford" "female" 26)
(excel/add-text-data-validation sheet
  ["male" "female" "unknown"]
  true ;; allow empty cell
  "Invalid gender"
  "Use one of: 'male', 'female', 'unknown'"
  1 2 3 3)
(excel/auto-size-columns sheet)
(excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/add-url-hyperlink

```
(add-url-hyperlink sheet row col text url)
```

Adds an URL hyperlink to a cell

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-font wbook :hyperlink { :underline true
                                         :color :BLUE })
    (excel/add-style wbook :hyperlink { :font :hyperlink })
    (excel/write-values sheet 1 1 "John" "Doe")
    (excel/write-values sheet 2 1 "Sue" "Ford")
    (excel/add-url-hyperlink sheet 1 3 "https://john.doe.org/" "https://john.doe.org/")
    (excel/add-url-hyperlink sheet 2 3 "https://sue.ford.org/" "https://sue.ford.org/")
    (excel/cell-style sheet 1 3 :hyperlink)
    (excel/cell-style sheet 2 3 :hyperlink)))
```



```
(excel/auto-size-columns sheet)
(excel/write->file wbook "sample.xlsx"))
```

SEE ALSO

[excel/remove-hyperlink](#)

Remove a cell comment

[excel/add-email-hyperlink](#)

Adds an email hyperlink to a cell

[section](#)

[top](#)

excel/addr->string

```
(addr->string row col)
```

Returns an Excel A1-style cell address string representation for a row and column address

```
(excel/addr->string 1 3)
```

```
(excel/addr->string 30 56)
```

```
(do
  (load-module :excel)
  (let [data [ {:a 100 :b 200 }
                {:a 101 :b 201 }
                {:a 102 :b 202 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row true })
        addr #(excel/addr->string %1 %2)
        sum #(str "SUM(" %1 "," %2 ")")]
    (excel/add-column sheet "A" { :field :a })
    (excel/add-column sheet "B" { :field :b })
    (excel/add-column sheet "C" { :field :c })
    (excel/write-items sheet data)
    (excel/cell-formula sheet 1 3 (sum (addr 1 1) (addr 1 2)))
    (excel/cell-formula sheet 2 3 (sum (addr 2 1) (addr 2 2)))
    (excel/cell-formula sheet 3 3 (sum (addr 3 1) (addr 3 2)))
    (excel/evaluate-formulas wbook)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))
```

SEE ALSO

[excel/col->string](#)

Returns an Excel A-style column number string representation for a column number

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[section](#)

[top](#)

excel/area-data-series

```
(area-data-series title data-address-range)
```

Build an area chart data series

Arguments:

title	The series title
data-address-range	The series data in the Excel

```
(excel/area-data-series "Countries" (excel/cell-address-range 2 2 1 5))
```

SEE ALSO

[excel/cell-address-range](#)
Build a cell address range

[section](#)

[top](#)

excel/auto-size-column

```
(auto-size-column sheet col)
```

Auto size the width of column col (1..n) in the sheet.

```
(do
  (load-module :excel)
  (let [data [ { :first "John" :last "Doe" :age 28 }
               { :first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-column sheet 1)
    (excel/auto-size-column sheet 2)
    (excel/auto-size-column sheet 3)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/auto-size-columns](#)
Auto size the width of all columns in the sheet.

[excel/write-items](#)
Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)
Render a single data item to the sheet

[excel/write-value](#)
Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)
Set a formula for a specific cell given by its row and col.

[excel/row-height](#)
Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/auto-size-columns

`(auto-size-columns sheet)`

Auto size the width of all columns in the sheet.

```
(do
  (load-module :excel)
  (let [data [ { :first "John" :last "Doe"   :age 28 }
                { :first "Sue"  :last "Ford"  :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name"  { :field :last })
    (excel/add-column sheet "Age"        { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#) [top](#)

excel/bar-data-series

`(bar-data-series title data-address-range)`

Build a bar chart data series

Arguments:

title	The series title
data-address-range	The series data in the Excel

```
(excel/bar-data-series "Countries" (excel/cell-address-range 2 2 1 5))
```

SEE ALSO

[excel/cell-address-range](#)

Build a cell address range

[section](#) [top](#)

excel/bg-color

```
(bg-color sheet row col color)
(bg-color sheet row col-start col-end color)
(bg-color sheet row-start row-end col-start col-end color & colors)
```

Sets a background color for a single cell, a range of columns within a row, or region of cells.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]

    ;; single cells
    (excel/bg-color sheet 1 1 "#27ae60")
    (excel/bg-color sheet 1 2 "#52be80")
    (excel/bg-color sheet 1 3 "#7dcea0")

    ;; range of cells in row
    (excel/bg-color sheet 1 4 6 "#3498db")

    ;; area of cells
    (excel/bg-color sheet 1 6 7 9 "#aed6f1")
    (excel/bg-color sheet 1 6 10 12 "#bb8fce" "#d2b4de")
    (excel/bg-color sheet 1 6 13 15 "#f1c40f" "#f4d03f" "#f7dc6f")
    (excel/write->file wbook "sample.xlsx")))

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-data sheet [[100 101 102]
                              [200 201 203]
                              [300 301 303]
                              [400 401 403]
                              [500 501 503]
                              [600 601 603]])
    (excel/bg-color sheet 1 6 1 3 "#a9cafc" "#d9e7fc")
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/cell-style](#)

Apply a defined cell style to a cell

[section](#)

[top](#)

excel/cell-address-range

```
(cell-address-range row-first row-last col-first col-last)
```

Build a cell address range

```
(excel/cell-address-range 1 2 1 10)
```

SEE ALSO

[excel/cell-address](#)

Returns the cell address in A1 style for a cell at row/col in a sheet

[section](#)

[top](#)

excel/cell-data-format-string

```
(cell-data-format-string sheet row col)
```

Returns the sheet cell data format string

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    (excel/cell-data-format-string sheet 1 3)))
```

SEE ALSO

[excel/cell-formula-result-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown } after formula ...

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-hidden?](#)

Returns true if the sheet cell is hidden else false.

[excel/cell-locked?](#)

Returns true if the sheet cell is locked else false.

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)

[top](#)

excel/cell-empty?

```
(cell-empty? sheet row col)
```

Returns true if the sheet cell given by row/col is empty.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    [(excel/cell-empty? sheet 1 1)
     (excel/cell-empty? sheet 2 1)
     (excel/cell-empty? sheet 3 1)]))
```

SEE ALSO

[excel/cell-hidden?](#)

Returns true if the sheet cell is hidden else false.

[excel/cell-locked?](#)

Returns true if the sheet cell is locked else false.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)

[top](#)

excel/cell-formula

```
(cell-formula sheet row col formula)
```

Set a formula for a specific cell given by its row and col.

```
(do
  (load-module :excel))
```

```

(let [data [ {:a 100 :b 200 }
              {:a 101 :b 201 }
              {:a 102 :b 202 } ]
      wbook (excel/create :xlsx)
      sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row true })]
  (excel/add-column sheet "A" { :field :a })
  (excel/add-column sheet "B" { :field :b })
  (excel/add-column sheet "C" { :field :c })
  (excel/write-items sheet data)
  (excel/cell-formula sheet 1 3 "SUM(A1,B1)")
  (excel/cell-formula sheet 2 3 "SUM(A2,B2)")
  (excel/cell-formula sheet 3 3 "SUM(A3,B3)")
  (excel/evaluate-formulas wbook)
  (excel/auto-size-columns sheet)
  (excel/write->file wbook "sample.xlsx"))

(do
  (load-module :excel)
  (let [data [ {:a 100 :b 200 }
                {:a 101 :b 201 }
                {:a 102 :b 202 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row true })]
    (excel/add-font wbook :bold { :bold true })
    (excel/add-style wbook :bold { :font :bold })
    (excel/add-column sheet "A" { :field :a })
    (excel/add-column sheet "B" { :field :b })
    (excel/add-column sheet "C" { :field :c })
    (excel/write-items sheet data)
    (excel/cell-formula sheet 1 3 "SUM(A1,B1)" :bold)
    (excel/cell-formula sheet 2 3 "SUM(A2,B2)" :bold)
    (excel/cell-formula sheet 3 3 "SUM(A3,B3)" :bold)
    (excel/evaluate-formulas wbook)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

```

SEE ALSO

[excel/addr->string](#)

Returns an Excel A1-style cell address string representation for a row and column address

[excel/sum-formula](#)

Returns a sum formula for the given cell area

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

excel/cell-formula-result-type

(cell-formula-result-type sheet row col)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown } after formula cell evaluation. For non formula cells this function is the same as the `cell-type` function.

```
(do
  (load-module :excel)

  (defn create-excel []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[10 20.123 {:formula "SUM(A1,B1)"}]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (create-excel))
        sheet (excel/sheet wbook "Data")]
      (excel/evaluate-formulas wbook)
      (printf "Cell (1,3): %s\n" (excel/cell-type sheet 1 3))
      (printf "Cell (1,3): %s (formula result type)\n"
               (excel/cell-formula-result-type sheet 1 3))))
```

SEE ALSO

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)

[top](#)

excel/cell-hidden?

(cell-hidden? sheet row col)

Returns true if the sheet cell is hidden else false.

```
(do
  (load-module :excel)
```



```
(let [wbook (excel/create :xlsx)
      sheet (excel/add-sheet wbook "Sheet 1")]
  (excel/write-values sheet 1 1 "John" "Doe" 28)
  (excel/auto-size-columns sheet)
  (excel/write->file wbook "sample.xlsx"))

(let [wbook (excel/open "sample.xlsx")
      sheet (excel/sheet wbook "Sheet 1")]
  (excel/cell-hidden? sheet 1 1)))
```

SEE ALSO

[excel/cell-locked?](#)

Returns true if the sheet cell is locked else false.

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[section](#)

[top](#)

excel/cell-lock

```
(cell-lock sheet row col locked?)
```

Locks/unlocks a cell.

Note: Excel locks new cells by default.

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/cell-lock sheet 1 1 false)
    (excel/cell-lock sheet 1 2 false)
    (excel/cell-lock sheet 1 3 true)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    [(excel/cell-locked? sheet 1 1)
     (excel/cell-locked? sheet 1 2)
     (excel/cell-locked? sheet 1 3)]))
```

SEE ALSO

[excel/cell-locked?](#)

Returns true if the sheet cell is locked else false.

[excel/cell-hidden?](#)

Returns true if the sheet cell is hidden else false.

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

excel/cell-locked?

```
(cell-locked? sheet row col)
```

Returns true if the sheet cell is locked else false.

Note: Excel locks new cells by default.

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/cell-lock sheet 1 1 false)
    (excel/cell-lock sheet 1 2 false)
    (excel/cell-lock sheet 1 3 true)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    [(excel/cell-locked? sheet 1 1)
     (excel/cell-locked? sheet 1 2)
     (excel/cell-locked? sheet 1 3)]))
```

SEE ALSO

[excel/cell-lock](#)

Locks/unlocks a cell.

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

excel/cell-style

```
(cell-style sheet row col style-id)
```

```
(cell-style sheet row-from row-to col-from col-to style-id)
```

Apply a defined cell style to a cell

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row false })]
    (excel/add-font wbook :bold { :bold true
                                   :color "#54039c" })
    (excel/add-style wbook :style-1 { :font :bold
                                       :h-align :left
                                       :rotation 0 })
    (excel/add-style wbook :style-2 { :bg-color "#cae1fa"
```

```

                                :h-align :center
                                :rotation 0
                                :border-top :thin
                                :border-left :thin
                                :border-bottom :thin
                                :border-right :thin})
(excel/add-style wbook :style-3 { :h-align :right
                                :format "#,##0.00" })

(excel/write-value sheet 2 1 100)
(excel/write-value sheet 2 2 200)
(excel/write-value sheet 2 3 300)

(excel/cell-style sheet 2 1 :style-1)
(excel/cell-style sheet 2 2 :style-2)
(excel/cell-style sheet 2 3 :style-3)

(excel/write->file wbook "sample.xlsx"))

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row false })]
    (excel/add-style wbook :style { :bg-color "#cae1fa"
                                    :h-align :center
                                    :format "#,##0.00" })

    (excel/write-value sheet 2 2 100)
    (excel/write-value sheet 2 3 200)
    (excel/write-value sheet 2 4 300)
    (excel/write-value sheet 3 2 101)
    (excel/write-value sheet 3 3 201)
    (excel/write-value sheet 3 4 301)

    (excel/cell-style sheet 2 3 2 4 :style)

    (excel/write->file wbook "sample.xlsx")))

```

SEE ALSO

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[section](#)
[top](#)

excel/cell-style-info

```
(cell-style-info sheet row col)
```

Returns a map with the cell's styles.

```

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]

```

```
(excel/write-values sheet 1 1 "John" "Doe" 28)
(excel/cell-style-info sheet 1 1)))
```

SEE ALSO

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/cell-style](#)

Apply a defined cell style to a cell

[section](#)

[top](#)

excel/cell-type

```
(cell-type sheet row col)
```

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

Note:

1. Excel returns cells containing long, double, date or datetime values as `:numeric` . The reader decides how to read a numeric cell using either of `excel/read-long-val` , `excel/read-double-val` , or `excel/read-date-val` .
2. To evaluate formulas to values call `excel/evaluate-formulas` on the workbook the right after opening the excel document.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 "101" 102.0]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    [(excel/cell-type sheet 1 1)
     (excel/cell-type sheet 1 2)
     (excel/cell-type sheet 1 3)
     (excel/cell-type sheet 1 4)]))
```

SEE ALSO

[excel/cell-formula-result-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown } after formula ...

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-hidden?](#)

Returns true if the sheet cell is hidden else false.

[excel/cell-locked?](#)

Returns true if the sheet cell is locked else false.

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)

[top](#)

excel/clear-row

```
(clear-row sheet row)
(clear-row sheet row clear-value clear-style)
```

Clears the values and/or styles in a specific row in a sheet.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/clear-row sheet 2)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/insert-empty-row](#)

Inserts an empty row or multiple empty rows to a sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

excel/col->string

```
(col->string col)
```

Returns an Excel A-style column number string representation for a column number

```
(excel/col->string 1)
```

```
(excel/col->string 56)
```

SEE ALSO

[excel/addr->string](#)

Returns an Excel A1-style cell address string representation for a row and column address

excel/col-hidden?

```
(col-hidden? sheet col)
```

Returns true if the sheet column is hidden else false.

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    (excel/col-hidden? sheet 1)))
```

SEE ALSO

[excel/cell-locked?](#)

Returns true if the sheet cell is locked else false.

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

excel/col-width

```
(col-width sheet col width)
```

Set the width of a column (1..n) in the sheet.

```
(do
  (load-module :excel)
  (let [os      (io/file-out-stream "sample.xlsx")
        data    [ {:first "John" :last "Doe"   :age 28 }
                   {:first "Sue"  :last "Ford"  :age 26 } ]
        wbook    (excel/create :xlsx)
        sheet    (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name"  { :field :last })
    (excel/add-column sheet "Age"       { :field :age })
    (excel/write-items sheet data)
    (excel/col-width sheet 1 80)
    (excel/col-width sheet 2 80)
    (excel/col-width sheet 3 60)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[section](#)

[top](#)

excel/copy-cell-style

```
(copy-cell-style sheet cell-from-row cell-from-col cell-to-row cell-to-col)
```

Copies the style from cell-from to cell-to

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/copy-cell-style sheet 1 1 2 1)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/copy-row

```
(copy-row sheet row-from row-to)
```

```
(copy-row sheet row-from row-to copy-value copy-style)
```

Copies a specific row in a sheet.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/copy-row sheet 1 2)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/insert-empty-row](#)

Inserts an empty row or multiple empty rows to a sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/copy-row-to-end

(copy-row-to-end sheet row)

(copy-row-to-end sheet row copy-value copy-style)

Copies a specific row from a sheet to end of the sheet.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/copy-row-to-end sheet 1)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/insert-empty-row](#)

Inserts an empty row or multiple empty rows to a sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/create

(create type)

Creates a new Excel for the given type :xls or :xlsx.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/open](#)

Opens an existing Excel for reading or modifying.

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[excel/write->file](#)

Writes the excel to a file.

[excel/write->stream](#)

Writes the excel to a Java :OutputStream.

[excel/write->bytebuf](#)

Writes the excel to a bytebuf. Returns the bytebuf.

[excel/evaluate-formulas](#)

Evaluate all formulas in a workbook or a sheet.

[section](#)

[top](#)

excel/delete-row

(delete-row sheet row)

Deletes a specific row from a sheet.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/delete-row sheet 1))
```

```
(excel/auto-size-columns sheet)
(excel/write->file wbook "sample.xlsx"))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/insert-empty-row](#)

Inserts an empty row or multiple empty rows to a sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/display-grid-lines

```
(display-grid-lines sheet display?)
```

If `true` displays grid lines in the Excel else omits them.

To control grid lines for printed Excls see the function `excel/print-layout`.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/display-grid-lines sheet false)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/print-layout](#)

Sets the sheet's print layout

[excel/page-margins](#)

Sets the sheet's page margins (in inches)

[excel/header-margin](#)

Sets the sheet's header margin (in inches)

[excel/footer-margin](#)

Sets the sheet's footer margin (in inches)

[excel/header](#)

Sets the sheet's header text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/footer](#)

Sets the sheet's footer text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[section](#) [top](#)

excel/evaluate-formula

(evaluate-formula sheet row col)

Evaluate the formula a sheet cell.

```
(do
  (load-module :excel)
  (let [data [ {:a 100 :b 200 }
                {:a 101 :b 201 }
                {:a 102 :b 202 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row true })]
    (excel/add-column sheet "A" { :field :a })
    (excel/add-column sheet "B" { :field :b })
    (excel/add-column sheet "C" { :field :c })
    (excel/write-items sheet data)
    (excel/cell-formula sheet 1 3 "SUM(A1,B1)")
    (excel/cell-formula sheet 2 3 "SUM(A2,B2)")
    (excel/cell-formula sheet 3 3 "SUM(A3,B3)")
    (excel/evaluate-formula sheet 1 3)
    (excel/evaluate-formula sheet 2 3)
    (excel/evaluate-formula sheet 3 3)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/evaluate-formulas](#)

Evaluate all formulas in a workbook or a sheet.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/remove-formula](#)

Remove a cell formula

[section](#) [top](#)

excel/evaluate-formulas

(evaluate-formulas wbook-or-sheet)

Evaluate all formulas in a workbook or a sheet.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))]
    (excel/evaluate-formulas wbook)))
```

SEE ALSO

[excel/evaluate-formula](#)

Evaluate the formula a sheet cell.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/remove-formula](#)

Remove a cell formula

[section](#)

[top](#)

excel/footer

```
(footer sheet text position font-size bold?)
```

Sets the sheet's footer text at position (`:LEFT` , `:CENTER` , `:RIGHT`) with a given font-size (in points) and bold style

Supported place holders in the text string

<code>{page}</code>	The current page number
<code>{num-pages}</code>	The number of pages
<code>{date}</code>	The current date
<code>{time}</code>	The current time

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/footer sheet "Customer Report" :LEFT 12 false)
    (excel/footer sheet "{page} / {num-pages}" :RIGHT 12 false)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/print-layout](#)

Sets the sheet's print layout

[excel/page-margins](#)

Sets the sheet's page margins (in inches)

[excel/header-margin](#)

Sets the sheet's header margin (in inches)

[excel/footer-margin](#)

Sets the sheet's footer margin (in inches)

[excel/header](#)

Sets the sheet's header text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/display-grid-lines](#)

If true displays grid lines in the Excel else omits them.

[section](#)

[top](#)

excel/footer-margin

```
(footer-margin sheet margin)
```

Sets the sheet's footer margin (in inches)

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/footer-margin sheet 0.25)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/print-layout](#)

Sets the sheet's print layout

[excel/page-margins](#)

Sets the sheet's page margins (in inches)

[excel/header-margin](#)

Sets the sheet's header margin (in inches)

[excel/header](#)

Sets the sheet's header text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/footer](#)

Sets the sheet's footer text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/display-grid-lines](#)

If true displays grid lines in the Excel else omits them.

[section](#)

[top](#)

excel/freeze-pane

```
(freeze-pane sheet rows cols)
```

Creates a split (freezepane). Any existing freezepane or split pane is overwritten.

If both rows and cols are 0 then the existing freeze pane is removed.

rows: the number of rows to freeze (starting from the first row) cols: the number of columns to freeze (starting from the first column)

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
```

```
sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row false })]
(excel/write-data sheet [(map #(str "Col " %) (range 1 11))])
(excel/write-data sheet (partition 10 (range 100 500)) 2 1)
(excel/freeze-pane sheet 1 0)
(excel/auto-size-columns sheet)
(excel/write->file wbook "sample.xlsx"))
```

SEE ALSO

[excel/add-merge-region](#)

Add a merge region to the sheet.

[section](#)

[top](#)

excel/header

(header sheet text position font-size bold?)

Sets the sheet's header text at position (:LEFT , :CENTER , :RIGHT) with a given font-size (in points) and bold style

Supported place holders in the text string

{page}	The current page number
{num-pages}	The number of pages
{date}	The current date
{time}	The current time

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/header sheet "Sheet Title" :CENTER 24 true)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/print-layout](#)

Sets the sheet's print layout

[excel/page-margins](#)

Sets the sheet's page margins (in inches)

[excel/header-margin](#)

Sets the sheet's header margin (in inches)

[excel/footer-margin](#)

Sets the sheet's footer margin (in inches)

[excel/footer](#)

Sets the sheet's footer text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/display-grid-lines](#)

If true displays grid lines in the Excel else omits them.

[section](#)

[top](#)

excel/header-margin

```
(header-margin sheet margin)
```

Sets the sheet's header margin (in inches)

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/header-margin sheet 0.25)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/print-layout](#)

Sets the sheet's print layout

[excel/page-margins](#)

Sets the sheet's page margins (in inches)

[excel/footer-margin](#)

Sets the sheet's footer margin (in inches)

[excel/header](#)

Sets the sheet's header text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/footer](#)

Sets the sheet's footer text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/display-grid-lines](#)

If true displays grid lines in the Excel else omits them.

[section](#)

[top](#)

excel/hide-columns

```
(hide-columns sheet & columns)
```

Hide columns in the sheet.

```
;; hide column by column index (1..n)
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-columns sheet)
    (excel/hide-columns sheet 2) ;; hide column #2
    (excel/write->file wbook "sample.xlsx")))
```

```
;; hide column by column id
```

```
(do
```



```
(load-module :excel)
(let [data [ {:first "John" :last "Doe" :age 28 }
              {:first "Sue" :last "Ford" :age 26 } ]
      wbook (excel/create :xlsx)
      sheet (excel/add-sheet wbook "Sheet 1")]
  (excel/add-column sheet "Last Name" { :field :last, :id "lastname"})
  (excel/add-column sheet "First Name" { :field :first, :id "firstname"})
  (excel/add-column sheet "Age" { :field :age, :id "age" })
  (excel/write-items sheet data)
  (excel/auto-size-columns sheet)
  (excel/hide-columns sheet "firstname") ;; hide column "firstname"
  (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/insert-empty-row

```
(insert-empty-row sheet row)
(insert-empty-row sheet row count)
```

Inserts an empty row or multiple empty rows to a sheet.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/insert-empty-row sheet 2)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/clear-row](#)

Clears the values and/or styles in a specific row in a sheet.

[excel/delete-row](#)

Deletes a specific row from a sheet.

[excel/copy-row](#)

Copies a specific row in a sheet.

[excel/copy-row-to-end](#)

Copies a specific row from a sheet to end of the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/line-data-series

(line-data-series title smooth? marker data-address-range)

Build a line chart data series

Arguments:

title	The series title
smooth?	Smooth rendering (splines): <code>true</code> or <code>false</code>
marker	The marker type: <code>:CIRCLE</code> , <code>:DASH</code> , <code>:DIAMOND</code> , <code>:DOT</code> , <code>:NONE</code> , <code>:PLUS</code> , <code>:SQUARE</code> , <code>:STAR</code> , <code>:TRIANGLE</code>
data-address-range	The series data in the Excel

```
(excel/line-data-series "Countries"
  false
  :SQUARE
  (excel/cell-address-range 2 2 1 5))
```

SEE ALSO

[excel/cell-address-range](#)

Build a cell address range

[section](#)

[top](#)

excel/open

(open source)

Opens an existing Excel for reading or modifying.

Supported sources are *string file path*, `bytebuf`, `:java.io.File`, or `:java.io.InputStream`.

- *string file path* -> (excel/open "/Users/foo/data/test.xlsx")
- *classpath* -> (excel/open (io/load-classpath-resource "org/foo/data/test.xlsx"))
- *:java.io.File* -> (excel/open (io/file "/Users/foo/data/test.xlsx"))

- `:java.io.InputStream` -> `(excel/open (io/file-in-stream "/Users/foo/data/test.xlsx"))`
- `bytebuf` -> `(excel/open (io/slurp "/Users/foo/data/test.xlsx" :binary true))`

```
(do
  (load-module :excel)

  ;; create an excel
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  ;; open the excel and add another row
  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample1.xlsx")))
```

SEE ALSO

[excel/create](#)

Creates a new Excel for the given type :xls or :xlsx.

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[excel/add-style](#)

Add a style with optional attributes to an Excel.

[excel/write->file](#)

Writes the excel to a file.

[excel/write->stream](#)

Writes the excel to a Java :OutputStream.

[excel/write->bytebuf](#)

Writes the excel to a bytebuf. Returns the bytebuf.

[excel/evaluate-formulas](#)

Evaluate all formulas in a workbook or a sheet.

[section](#)

[top](#)

excel/page-margins

`(page-margins sheet left right top bottom)`

Sets the sheet's page margins (in inches)

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet))
```

```
(excel/page-margins sheet 0.25 0.25 0.75 0.75)
(excel/write->file wbook "sample.xlsx"))
```

SEE ALSO

- [excel/print-layout](#)
Sets the sheet's print layout
- [excel/header-margin](#)
Sets the sheet's header margin (in inches)
- [excel/footer-margin](#)
Sets the sheet's footer margin (in inches)
- [excel/header](#)
Sets the sheet's header text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style
- [excel/footer](#)
Sets the sheet's footer text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style
- [excel/display-grid-lines](#)
If true displays grid lines in the Excel else omits them.

excel/pie-data-series

```
(pie-data-series data-address-range)
```

Build a pie chart data series

Arguments:

data-address-range The series data in the Excel

```
(excel/pie-data-series (excel/cell-address-range 2 2 1 5))
```

SEE ALSO

- [excel/cell-address-range](#)
Build a cell address range

excel/print-layout

```
(print-layout sheet paper-size orientation
               header-margin footer-margin,
               grid-lines?
               fit-to-width? fit-to-page?)
```

Sets the sheet's print layout

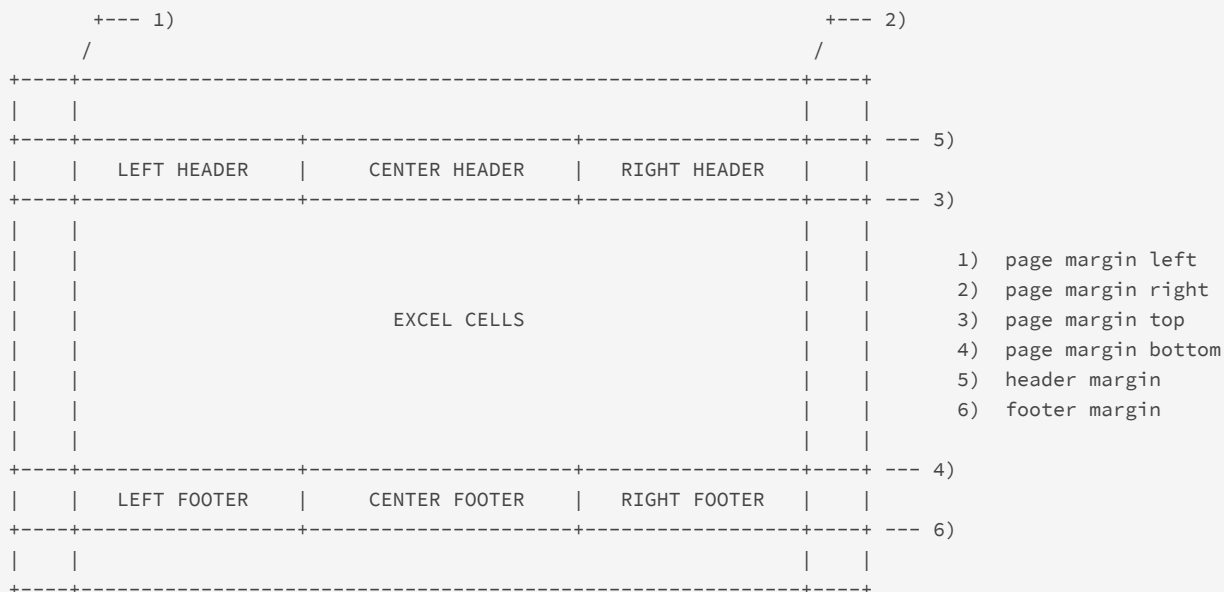
Arguments:

paper-size	:A4 , :A5 , :LETTER , :LETTER_SMALL , ...
orientation	:PORTARIT or :LANDSACPE
header-margin	header margin (in inches)
footer-margin	footer margin (in inches)

grid-lines? show grid lines { true or false }

fit-to-width? fit to page width { true or false }

fit-to-page? fit to page { true or false }



note: all margins in inches

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/print-layout sheet :A4 :LANDSCAPE 1.25 1.25 false true true)
    (excel/page-margins sheet 1.25 1.25 2.5 2.5)
    (excel/header sheet "H/LEFT" :LEFT 24 true)
    (excel/header sheet "H/CENTER" :CENTER 24 true)
    (excel/header sheet "H/RIGHT" :RIGHT 24 true)
    (excel/footer sheet "F/LEFT" :LEFT 24 true)
    (excel/footer sheet "F/CENTER" :CENTER 24 true)
    (excel/footer sheet "F/RIGHT" :RIGHT 24 true)
    (excel/write->file wbook "sample.xlsx")))
```

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/print-layout sheet :A4 :LANDSCAPE 1.25 1.25 false true true)
    (excel/page-margins sheet 1.25 1.25 2.5 1.25)
    (excel/header sheet "Example Report" :CENTER 24 true)
    (excel/footer sheet "{date} {time}" :LEFT 11 false)
    (excel/footer sheet "[page] / {num-pages}" :RIGHT 11 false)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/page-margins](#)

Sets the sheet's page margins (in inches)

[excel/header-margin](#)

Sets the sheet's header margin (in inches)

[excel/footer-margin](#)

Sets the sheet's footer margin (in inches)

[excel/header](#)

Sets the sheet's header text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/footer](#)

Sets the sheet's footer text at position (:LEFT, :CENTER, :RIGHT) with a given font-size (in points) and bold style

[excel/display-grid-lines](#)

If true displays grid lines in the Excel else omits them.

[section](#)

[top](#)

excel/protect-sheet

```
(protect-sheet sheet password)
```

Protect the sheet.

This will ensure that locked cells remain locked and unlocked cells are editable.

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/auto-size-columns sheet)
    (excel/protect-sheet sheet "password")
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/add-sheet](#)

Adds a sheet with optional attributes to an Excel.

[section](#)

[top](#)

excel/read-boolean-val

```
(read-boolean-val sheet row col)
```

Returns the sheet cell value as boolean.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 true 102]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
```

```
sheet (excel/sheet wbook "Data")]
(excel/read-boolean-val sheet 1 2)))
```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[excel/read-val](#)

Returns the sheet cell value.

[section](#)[top](#)

excel/read-date-val

```
(read-date-val sheet row col)
```

Returns the sheet cell value as a date (:java.time.LocalDate).

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")
          dt1   (time/local-date 2021 1 1)
          dt2   (time/local-date 2022 4 15)]
      (excel/write-data sheet [[100 dt1 dt2 102]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    [(excel/read-date-val sheet 1 2)
     (excel/read-date-val sheet 1 3)]))
```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[excel/read-val](#)

Returns the sheet cell value.

[section](#)

[top](#)

excel/read-datetime-val

```
(read-datetime-val sheet row col)
```

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")
          ts1   (time/local-date-time 2021 1 1 15 30 45)
          ts2   (time/local-date-time 2021 1 31 08 00 00)]
      (excel/write-data sheet [[100 ts1 ts2 102]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    [(excel/read-datetime-val sheet 1 2)
     (excel/read-datetime-val sheet 1 3)])))
```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-val](#)

Returns the sheet cell value.

excel/read-double-val

```
(read-double-val sheet row col)
```

Returns the sheet cell value as double.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101.23 102]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/read-double-val sheet 1 2)))
```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-val](#)

Returns the sheet cell value.

excel/read-error-code

```
(read-error-code sheet row col)
```

Reads the error code from a cell. Returns a string indicating the error or nil if the cell is nozt in error state.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
```

```
(excel/write-data sheet [[100 200 {:formula "1 / 0"}]])
(excel/write->bytebuf wbook))

(let [wbook (excel/open (test-xls))
      sheet (excel/sheet wbook "Data")]
  (excel/evaluate-formulas wbook)
  (excel/read-error-code sheet 1 3))) ;; #DIV/0!
```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)
[top](#)

excel/read-long-val

```
(read-long-val sheet row col)
```

Returns the sheet cell value as long.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101 102]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/read-long-val sheet 1 2)))

(do
  (load-module :excel)

  (defn test-xls []
    (let [data [ {:a 100 :b 200 } ]
          wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
```

```

        { :no-header-row true }]]
(excel/add-column sheet "A" { :field :a })
(excel/add-column sheet "B" { :field :b })
(excel/write-items sheet data)
(excel/cell-formula sheet 1 3 "SUM(A1,B1)")
(excel/write->bytebuf wbook))

(let [wbook (excel/open (test-xls))
      sheet (excel/sheet wbook "Data")]
  (excel/read-long-val sheet 1 3)))

```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[excel/read-val](#)

Returns the sheet cell value.

[section](#)
[top](#)

excel/read-string-val

```
(read-string-val sheet row col)
```

Returns the sheet cell value as string.

```

(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 "101" 102.0]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/read-string-val sheet 1 2)))

```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[excel/read-val](#)

Returns the sheet cell value.

[section](#)[top](#)

excel/read-val

```
(read-val sheet row col)
```

Returns the sheet cell value.

Returns a *nil*, *string*, *boolean*, or *double* value depending on the cell's excel type :*blank*, :*string*, :*boolean*, or :*numeri*.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 "101" 102.0]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/read-val sheet 1 2)))
```

SEE ALSO

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)

[top](#)

excel/remove-comment

(remove-comment sheet row col)

Remove a cell comment

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 45)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/remove-comment sheet 1 1)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/remove-formula](#)

Remove a cell formula

[excel/remove-hyperlink](#)

Remove a cell comment

[section](#)

[top](#)

excel/remove-formula

(remove-formula sheet row col)

Remove a cell formula

```
(do
  (load-module :excel)
  (let [data [ { :a 100 :b 200 }
               { :a 101 :b 201 }
               { :a 102 :b 202 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row true })]
    (excel/add-column sheet "A" { :field :a })
    (excel/add-column sheet "B" { :field :b })
    (excel/add-column sheet "C" { :field :c })
    (excel/write-items sheet data)
    (excel/cell-formula sheet 1 3 (excel/sum-formula sheet 1 1 1 2))
    (excel/cell-formula sheet 2 3 (excel/sum-formula sheet 2 2 1 2))
    (excel/cell-formula sheet 3 3 (excel/sum-formula sheet 3 3 1 2))
    (excel/remove-formula sheet 1 3)
    (excel/evaluate-formulas wbook)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/remove-comment](#)

Remove a cell comment

[excel/remove-hyperlink](#)

Remove a cell comment

[section](#)

[top](#)

excel/remove-hyperlink

```
(remove-hyperlink sheet row col)
```

Remove a cell comment

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 45)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/remove-hyperlink sheet 1 1)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/add-url-hyperlink](#)

Adds an URL hyperlink to a cell

[excel/add-email-hyperlink](#)

Adds an email hyperlink to a cell

[excel/remove-comment](#)

Remove a cell comment

[excel/remove-formula](#)

Remove a cell formula

[section](#)

[top](#)

excel/row-height

```
(row-height sheet row height)
```

Set the height of a row (1..n) in the sheet.

```
(do
  (load-module :excel)
  (let [os (io/file-out-stream "sample.xlsx")
        data [ { :first "John" :last "Doe" :age 28 }
                 { :first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
```

```
(excel/write-items sheet data)
(excel/auto-size-columns sheet)
(excel/row-height sheet 2 50)
(excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/col-width](#)

Set the width of a column (1..n) in the sheet.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[section](#)[top](#)

excel/sheet

```
(sheet wbook ref)
```

Returns a sheet from the Excel referenced by its name or sheet index.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet1 (excel/add-sheet wbook "Data1")
          sheet2 (excel/add-sheet wbook "Data2")]
      (excel/write-data sheet1 [[100 101 102] [200 201 202]])
      (excel/write-data sheet2 [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet1 (excel/sheet wbook "Data1")
        sheet2 (excel/sheet wbook 2)]
    ))
```

SEE ALSO

[excel/sheet-count](#)

Returns the number of sheets in the Excel.

[excel/evaluate-formulas](#)

Evaluate all formulas in a workbook or a sheet.

[excel/sheet-name](#)

Returns the name of a sheet.

[excel/sheet-row-range](#)

Returns the first and the last row with data in a sheet as vector. Returns -1 values if no row exists.

[excel/sheet-col-range](#)

Returns the first and the last col with data in a sheet row as vector. Returns -1 values if the row does not exist or the row does ...

[excel/cell-empty?](#)

Returns true if the sheet cell given by row/col is empty.

[excel/cell-type](#)

Returns the sheet cell type as one of { :notfound, :blank, :string, :boolean, :numeric, :formula, :error, or :unknown }

[excel/read-string-val](#)

Returns the sheet cell value as string.

[excel/read-boolean-val](#)

Returns the sheet cell value as boolean.

[excel/read-long-val](#)

Returns the sheet cell value as long.

[excel/read-double-val](#)

Returns the sheet cell value as double.

[excel/read-date-val](#)

Returns the sheet cell value as a date (:java.time.LocalDate).

[excel/read-datetime-val](#)

Returns the sheet cell value as a datetime (:java.time.LocalDateTime).

[section](#)[top](#)

excel/sheet-col-range

(sheet-col-range sheet)

Returns the first and the last col with data in a sheet row as vector. Returns -1 values if the row does not exist or the row does not have any columns.

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Mary" "Smith" 28)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    (excel/sheet-col-range sheet 1)))

(do
  (load-module :excel)

  (defn print-cell-meta [sheet row col]
    (println (str (excel/addr->string row col) "> "
                  "type: " (name (excel/cell-type sheet row col))
                  ", format: " (excel/cell-data-format-string sheet row col)
                  ", empty: " (excel/cell-empty? sheet row col)
                  ", locked: " (excel/cell-locked? sheet row col)
                  ", hidden: " (excel/cell-hidden? sheet row col)))))
```



```
(let [wbook (excel/create :xlsx)
      sheet (excel/add-sheet wbook "Sheet 1")]
  (excel/write-values sheet 1 1 "John" "Doe" 28)
  (excel/write-values sheet 2 1 "Mary" "Smith" 28)
  (excel/auto-size-columns sheet)
  (excel/write->file wbook "sample.xlsx"))

(let [wbook      (excel/open "sample.xlsx")
      sheet      (excel/sheet wbook "Sheet 1")
      row        1
      col-range  (excel/sheet-col-range sheet 1)
      col-list    (range (first col-range) (inc (second col-range)))]
  (docoll #(print-cell-meta sheet (first %) (second %))
    (map vector (repeat row) col-list))))
```

SEE ALSO

[excel/sheet-row-range](#)

Returns the first and the last row with data in a sheet as vector. Returns -1 values if no row exists.

[section](#)

[top](#)

excel/sheet-count

```
(sheet-count wbook)
```

Returns the number of sheets in the Excel.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))]
    (excel/sheet-count wbook)))
```

SEE ALSO

[excel/sheet](#)

Returns a sheet from the Excel referenced by its name or sheet index.

[excel/evaluate-formulas](#)

Evaluate all formulas in a workbook or a sheet.

[section](#)

[top](#)

excel/sheet-index

```
(sheet-index sheet)
```

Returns the index of a sheet.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/sheet-index sheet)))
```

[section](#)[top](#)

excel/sheet-name

```
(sheet-name sheet)
```

Returns the name of a sheet.

```
(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/create :xlsx)
          sheet (excel/add-sheet wbook "Data")]
      (excel/write-data sheet [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/sheet-name sheet)))
```

[section](#)[top](#)

excel/sheet-row-range

```
(sheet-row-range sheet)
```

Returns the first and the last row with data in a sheet as vector. Returns -1 values if no row exists.

```
(do
  (load-module :excel)

  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Mary" "Smith" 28)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx"))

  (let [wbook (excel/open "sample.xlsx")
        sheet (excel/sheet wbook "Sheet 1")]
    (excel/sheet-row-range sheet)))
```

SEE ALSO

[excel/sheet-col-range](#)

Returns the first and the last col with data in a sheet row as vector. Returns -1 values if the row does not exist or the row does ...

[section](#)

[top](#)

excel/sum-formula

(sum-formula sheet row-from row-to col-from col-to)

Returns a sum formula for the given cell area

```
(do
  (load-module :excel)
  (let [data [ {:a 100 :b 200 }
                {:a 101 :b 201 }
                {:a 102 :b 202 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1" { :no-header-row true })]
    (excel/add-column sheet "A" { :field :a })
    (excel/add-column sheet "B" { :field :b })
    (excel/add-column sheet "C" { :field :c })
    (excel/write-items sheet data)
    (excel/cell-formula sheet 1 3 (excel/sum-formula sheet 1 1 1 2))
    (excel/cell-formula sheet 2 3 (excel/sum-formula sheet 2 2 1 2))
    (excel/cell-formula sheet 3 3 (excel/sum-formula sheet 3 3 1 2))
    (excel/evaluate-formulas wbook)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/addr->string](#)

Returns an Excel A1-style cell address string representation for a row and column address

[section](#)

[top](#)

excel/write->bytebuf

(write->bytebuf wbook)

Writes the excel to a bytebuf. Returns the bytebuf.

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-columns sheet)
    (excel/write->bytebuf wbook)))
```

SEE ALSO

[excel/write->file](#)

Writes the excel to a file.

[excel/write->stream](#)

Writes the excel to a Java :OutputStream.

[section](#)

[top](#)

excel/write->file

```
(write->file wbook f)
```

Writes the excel to a file.

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/write->stream](#)

Writes the excel to a Java :OutputStream.

[excel/write->bytebuf](#)

Writes the excel to a bytebuf. Returns the bytebuf.

[section](#)

[top](#)

excel/write->stream

```
(write->stream wbook os)
```

Writes the excel to a Java :OutputStream.

```
(do
  (load-module :excel)
  (let [os (io/file-out-stream "sample.xlsx")
        data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
```

```
(excel/write-items sheet data)
(excel/auto-size-columns sheet)
(excel/write->stream wbook os)))
```

SEE ALSO

[excel/write->file](#)

Writes the excel to a file.

[excel/write->bytebuf](#)

Writes the excel to a bytebuf. Returns the bytebuf.

[section](#)

[top](#)

excel/write-data

```
(write-data sheet data)
(write-data sheet data row col)
```

Writes the data of a 2D array to an excel sheet.

Optionally the data can be written to a region starting at a row/col position.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")
        dt (time/local-date 2021 1 1)
        ts (time/local-date-time 2021 1 1 15 30 45)
        data [[100 101 102 103 104 105]
               [200 "ab" 1.23 dt ts false]]]
    (excel/write-data sheet data)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Data")]
    (excel/write-data sheet [[100 101 102] [200 201 203]])
    (excel/write-data sheet [[300 301 302] [400 401 403]] 3 4)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/write->stream](#)

Writes the excel to a Java :OutputStream.

[excel/write->bytebuf](#)

Writes the excel to a bytebuf. Returns the bytebuf.

[section](#)

[top](#)

excel/write-item

```
(write-item sheet item)
```

Render a single data item to the sheet

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-item sheet {:first "John" :last "Doe" :age 28 })
    (excel/write-item sheet {:first "Sue" :last "Ford" :age 26 })
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/write-items

```
(write-items sheet items)
```

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
               {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (excel/write-items sheet data)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)[top](#)

excel/write-value

```
(write-value sheet row col val)
(write-value sheet row col val style)
```

Writes a value with an optional to a specific cell given by its row and col.

If style is not passed or is `nil` uses a default style to render the value according to its data type:

- string: no format
- boolean: no format
- integer: `###0`
- double: `###0.00`
- date: `dd.mm.yyyy`
- datetime: `dd.mm.yyyy hh:mm:ss`

To use the existing cell's style without changing it when modifying the cell's value pass `:keep-style` as style!

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-value sheet 1 1 "John")
    (excel/write-value sheet 1 2 "Doe")
    (excel/write-value sheet 1 3 28)
    (excel/write-value sheet 2 1 "Sue")
    (excel/write-value sheet 2 2 "Ford")
    (excel/write-value sheet 2 3 26)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))

(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/add-font wbook :italic { :italic true })
    (excel/add-font wbook :bold { :bold true })
    (excel/add-style wbook :italic { :font :italic })
    (excel/add-style wbook :bold { :font :bold })
    (excel/write-value sheet 1 1 "John" :italic)
    (excel/write-value sheet 1 2 "Doe" :italic)
    (excel/write-value sheet 1 3 28 :bold)
    (excel/write-value sheet 2 1 "Sue" :italic)
    (excel/write-value sheet 2 2 "Ford" :italic))
```

```
(excel/write-value sheet 2 3 26 :bold)
(excel/auto-size-columns sheet)
(excel/write->file wbook "sample.xlsx"))
```

SEE ALSO

[excel/write-values](#)

Writes multiples value to a row starting at col and incrementing col for each value

[excel/write-values-keep-style](#)

Writes multiples value to a row starting at col and incrementing col for each value. Keeps the existing cell styles.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)[top](#)

excel/write-values

```
(write-values sheet row col & vals)
```

Writes multiples value to a row starting at col and incrementing col for each value

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/write-values-keep-style](#)

Writes multiples value to a row starting at col and incrementing col for each value. Keeps the existing cell styles.

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

excel/write-values-keep-style

```
(write-values-keep-style sheet row col & vals)
```

Writes multiples value to a row starting at col and incrementing col for each value. Keeps the existing cell styles.

```
(do
  (load-module :excel)
  (let [wbook (excel/create :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1")]
    (excel/write-values sheet 1 1 "John" "Doe" 28)
    (excel/write-values sheet 2 1 "Sue" "Ford" 26)
    (excel/auto-size-columns sheet)
    (excel/write->file wbook "sample.xlsx")))
```

SEE ALSO

[excel/write-value](#)

Writes a value with an optional to a specific cell given by its row and col.

[excel/write-values](#)

Writes multiples value to a row starting at col and incrementing col for each value

[excel/write-items](#)

Writes the passed data items, a sequence of maps of name/value pairs, to the sheet.

[excel/write-item](#)

Render a single data item to the sheet

[excel/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[excel/auto-size-columns](#)

Auto size the width of all columns in the sheet.

[excel/auto-size-column](#)

Auto size the width of column col (1..n) in the sheet.

[excel/row-height](#)

Set the height of a row (1..n) in the sheet.

[section](#)

[top](#)

exists-class?

```
(exists-class? name)
```

Returns true the Java class for the given name exists otherwise returns false.

```
(exists-class? :java.util.ArrayList)
=> true
```

[section](#)[top](#)

exp

```
(exp x)
```

Returns Euler's number e raised to the power of a value.

```
(exp 10)
=> 22026.465794806718
```

```
(exp 10.23)
=> 27722.51006805505
```

```
(exp 10.23M)
=> 27722.51006805505
```

SEE ALSO

[exp](#)

Returns Euler's number e raised to the power of a value.

[section](#)[top](#)

extend

```
(extend type protocol fns*)
```

Extends protocol for type with the supplied functions.

Formats:

- `(extend :core/long P (foo [x] x))`
- `(extend :core/long P (foo [x] x) (foo [x y] x))`
- `(extend :core/long P (foo [x] x) (bar [x] x))`

```
(do
  (ns foo)
  (deftype :complex [re :long, im :long])
  (defprotocol XMath (+ [x y]
                      (- [x y]))
    (extend :foo/complex XMath
      (+ [x y] (complex. (core/+ (:re x) (:re y))
                          (core/+ (:im x) (:im y)))))
      (- [x y] (complex. (core/- (:re x) (:re y))
                          (core/- (:im x) (:im y)))))
    (extend :core/long XMath
      (+ [x y] (core/+ x y))
      (- [x y] (core/- x y))
      (foo/+ (complex. 1 1) (complex. 4 5)))
  => {:custom-type* :foo/complex :re 5 :im 6})
```

SEE ALSO

[defprotocol](#)

Defines a new protocol with the supplied function specs.

[extends?](#)

Returns true if the type extends the protocol.

[section](#)[top](#)

extends?

(extends? type protocol)

Returns true if the type extends the protocol.

```
(do
  (ns foo)
  (deftype :complex [re :long, im :long])
  (defprotocol XMath (+ [x y])
                    (- [x y]))
  (extend :foo/complex XMath
    (+ [x y] (complex. (core/+ (:re x) (:re y))
                      (core/+ (:im x) (:im y)))))
    (- [x y] (complex. (core/- (:re x) (:re y))
                      (core/- (:im x) (:im y)))))

  (extend :core/long XMath
    (+ [x y] (core/+ x y))
    (- [x y] (core/- x y)))
  (extends? :foo/complex XMath))
=> true
```

SEE ALSO

[defprotocol](#)

Defines a new protocol with the supplied function specs.

[extend](#)

Extends protocol for type with the supplied functions.

[section](#)[top](#)

false?

(false? x)

Returns true if x is false, false otherwise

```
(false? true)
=> false
```

```
(false? false)
=> true
```

```
(false? nil)
=> false
```

```
(false? 0)
=> false

(false? (== 1 2))
=> true
```

SEE ALSO

[true?](#)

Returns true if x is true, false otherwise

[not](#)

Returns true if x is logical false, false otherwise.

[section](#)[top](#)

filter

```
(filter predicate coll)
```

Returns a collection of the items in coll for which `(predicate item)` returns logical true.
Returns a transducer when no collection is provided.

```
(filter even? [1 2 3 4 5 6 7])
=> (2 4 6)

(filter #(even? (val %)) {:a 1 :b 2})
=> ([:b 2])

(filter even? #{1 2 3})
=> (2)
```

SEE ALSO

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[reduce](#)

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

[map-indexed](#)

Returns a collection of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item of coll, etc. until ...

[tee](#)

Applies f to every item of each the coll and returns the unaltered coll.

[section](#)[top](#)

filter-k

```
(filter-k f map)
```

Returns a map with entries for which the predicate (f key) returns logical true. f is a function with one arguments.

```
(filter-k #(= % :a) {:a 1 :b 2 :c 3})
=> {:a 1}
```

SEE ALSO

[filter-kv](#)

Returns a map with entries for which the predicate (f key value) returns logical true. f is a function with two arguments.

[section](#)[top](#)

filter-kv

```
(filter-kv f map)
```

Returns a map with entries for which the predicate (f key value) returns logical true. f is a function with two arguments.

```
(filter-kv (fn [k v] (= k :a)) {:a 1 :b 2 :c 3})  
=> {:a 1}
```

```
(filter-kv (fn [k v] (= v 2)) {:a 1 :b 2 :c 3})  
=> {:b 2}
```

SEE ALSO

[filter-k](#)

Returns a map with entries for which the predicate (f key) returns logical true. f is a function with one arguments.

[section](#)[top](#)

find

```
(find map key)
```

Returns the map entry for key, or nil if key not present.

```
(find {:a 1 :b 2} :b)  
=> [:b 2]
```

```
(find {:a 1 :b 2} :z)  
=> nil
```

[section](#)[top](#)

finder

```
(finder & args)
```

Finds symbols that match one more glob patterns or regular expressions.

Filters the symbol names by 0 to n glob patterns or regular expressions.

Glob patterns and regular expressions are ANDed, flags are ORed.

Flags:

:function filter functions

:macro filter macros

:special-form	filter special forms
:protocol	filter protocols
:value	filter values
:tracing	filter symbols under tracing (functions only)
:machine	return the result as Venice data otherwise print it in table format

```
(finder "io/zip*")
```

```
io/zip                :core/function
io/zip-append         :core/function
io/zip-file           :core/function
io/zip-list           :core/function
io/zip-list-entry-names :core/function
io/zip-remove         :core/function
io/zip-size           :core/function
io/zip?               :core/function
=> nil
```

```
(finder "*delete-file*")
```

```
io/delete-file        :core/function
io/delete-file-on-exit :core/function
io/delete-file-tree    :core/function
io/delete-files-glob   :core/function
openai-java/delete-file :core/function
=> nil
```

```
(finder "io/zip*" :machine)
```

```
=> ([io/zip :core/function] [io/zip-append :core/function] [io/zip-file :core/function] [io/zip-list :core/function] [io/zip-list-entry-names :core/function] [io/zip-remove :core/function] [io/zip-size :core/function] [io/zip? :core/function])
```

```
(finder #"io/zip.*")
```

```
io/zip                :core/function
io/zip-append         :core/function
io/zip-file           :core/function
io/zip-list           :core/function
io/zip-list-entry-names :core/function
io/zip-remove         :core/function
io/zip-size           :core/function
io/zip?               :core/function
=> nil
```

```
(finder #".*delete-file*")
```

```
io/delete-file        :core/function
io/delete-file-on-exit :core/function
io/delete-file-tree    :core/function
io/delete-files-glob   :core/function
openai-java/delete-file :core/function
=> nil
```

```
(finder #"io/zip.*" :machine)
```

```
=> ([io/zip :core/function] [io/zip-append :core/function] [io/zip-file :core/function] [io/zip-list :core/function] [io/zip-list-entry-names :core/function] [io/zip-remove :core/function] [io/zip-size :core/function] [io/zip? :core/function])
```

```
(finder zip)
```

```
geoip/download-maxmind-db-to-zipfile :core/function
grep/grep-zip                         :core/function
io/gzip                               :core/function
io/gzip-to-stream                     :core/function
io/gzip?                             :core/function
```

```

io/ungzip                :core/function
io/ungzip-to-stream      :core/function
io/unzip                 :core/function
io/unzip-all            :core/function
io/unzip-first           :core/function
io/unzip-nth             :core/function
io/unzip-to-dir          :core/function
io/wrap-is-with-gzip-input-stream :core/function
io/wrap-os-with-gzip-output-stream :core/function
io/zip                   :core/function
io/zip-append            :core/function
io/zip-file              :core/function
io/zip-list              :core/function
io/zip-list-entry-names  :core/function
io/zip-remove            :core/function
io/zip-size              :core/function
io/zip?                  :core/function
swiss-cities/lv95-zip-entryname :core/string
swiss-cities/lv95-zip-filename  :core/string
zipmap                   :core/function
zipvault/add-empty-folder :core/function
zipvault/add-file         :core/function
zipvault/add-files        :core/function
zipvault/add-folder       :core/function
zipvault/add-stream       :core/function
zipvault/encrypted?       :core/function
zipvault/entries          :core/function
zipvault/entropy          :core/function
zipvault/extract-all     :core/function
zipvault/extract-file     :core/function
zipvault/extract-file-data :core/function
zipvault/remove-files     :core/function
zipvault/valid-zip-file?  :core/function
zipvault/zip              :core/function
zipvault/zip-folder       :core/function
=> nil

(do
  (load-module :trace)
  (trace/trace-var bigint)
  (finder "big*" :tracing))
bigint :core/function
=> nil

```

SEE ALSO

[doc](#)

Prints documentation for a var or special form given x as its name. Prints the definition of custom types.

[ns-list](#)

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

[modules](#)

Lists the available Venice modules

[section](#)

[top](#)

first

```
(first coll)
```

Returns the first element of coll or nil if coll is nil or empty.

```
(first nil)
=> nil

(first [])
=> nil

(first [1 2 3])
=> 1

(first '())
=> nil

(first '(1 2 3))
=> 1

(first "abc")
=> #\a
```

[section](#)

[top](#)

flatten

```
(flatten coll)
```

Takes any nested combination of collections (lists, vectors, etc.) and returns their contents as a single, flat sequence. `(flatten nil)` returns an empty list.
Returns a transducer when no collection is provided.

```
(flatten [])
=> []

(flatten [[1 2 3] [4 [5 6]] [7 [8 [9]]]])
=> [1 2 3 4 5 6 7 8 9]

(flatten [1 2 {:a 3 :b [4 5 6]}])
=> [1 2 {:a 3 :b [4 5 6]}]

(flatten (seq {:a 1 :b 2}))
=> (:a 1 :b 2)
```

SEE ALSO

[mapcat](#)

Returns the result of applying concat to the result of applying map to fn and colls. Thus function fn should return a collection.

[section](#)

[top](#)

float

```
(float x)
```

Converts x to `:java.lang.Float`.

Note: Venice does not have a built-in float type!

```
(float 1.2F)
=> 1.2F
```

```
(float 1)
=> 1.0F
```

```
(float nil)
=> 0.0F
```

```
(float false)
=> 0.0F
```

```
(float true)
=> 1.0F
```

```
(float 1.2)
=> 1.2F
```

```
(float 1.2M)
=> 1.2F
```

```
(float "1.2")
=> 1.2F
```

[section](#)

[top](#)

float-array

```
(float-array coll)
(float-array len)
(float-array len init-val)
```

Returns an array of Java primitive floats containing the contents of coll or returns an array with the given length and optional init value.

To create an array of :java.lang.Float use:

```
(make-array :java.lang.Long 3)
```

```
(float-array '(1.0F 2.0F 3.0F))
=> [1.0, 2.0, 3.0]
```

```
(float-array '(1I 2 3.2 3.56M))
=> [1.0, 2.0, 3.2, 3.56]
```

```
(float-array 10)
=> [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
(float-array 10 42.0F)
=> [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0]
```

SEE ALSO

[java-float-list](#)

Converts a Venice list/vector to a Java Float list

float?

(float? n)

Returns true if n is a float

```
(float? 4.0F)  
=> true
```

```
(float? 4.0)  
=> false
```

```
(float? 3)  
=> false
```

```
(float? 3I)  
=> false
```

```
(float? 3.0M)  
=> false
```

```
(float? true)  
=> false
```

```
(float? nil)  
=> false
```

```
(float? {})  
=> false
```

floor

(floor x)

Returns the largest integer that is less than or equal to x

```
(floor 1.4)  
=> 1.0
```

```
(floor -1.4)  
=> -2.0
```

```
(floor 1.23M)  
=> 1.00M
```

```
(floor -1.23M)  
=> -2.00M
```

SEE ALSO

ceil

Returns the largest integer that is greater than or equal to x

[section](#)[top](#)

flush

```
(flush)
(flush os)
```

Without arg flushes the output stream that is the current value of `*out*`. With arg flushes the passed stream that must be a subclass of either `:java.io.OutputStream` or `:java.io.Writer`.

Returns `nil`.

```
(flush)
=> nil

(flush *out*)
=> nil

(flush *err*)
=> nil
```

SEE ALSO

[io/flush](#)

Flushes a `:java.io.OutputStream` or a `:java.io.Writer`.

[io/close](#)

Closes a `:java.io.InputStream`, `:java.io.OutputStream`, `:java.io.Reader`, or a `:java.io.Writer`.

[section](#)[top](#)

fn

```
(fn name? [params*] condition-map? expr*)
```

Defines an anonymous function.

```
(do
  (def sum (fn [x y] (+ x y)))
  (sum 2 3))
=> 5

;; multi-arity anonymous function
(let [f (fn ([x] x) ([x y] (+ x y)))]
  [(f 1) (f 4 6)])
=> [1 10]

(map (fn double [x] (* 2 x)) (range 1 5))
=> (2 4 6 8)

(map #(* 2 %) (range 1 5))
=> (2 4 6 8)
```

```

(map #(* 2 %1) (range 1 5))
=> (2 4 6 8)

;; anonymous function with two params, the second is destructured
(reduce (fn [m [k v]] (assoc m v k)) {} {:b 2 :a 1 :c 3})
=> {1 :a 2 :b 3 :c}

;; defining a pre-condition
(do
  (def square-root
    (fn [x]
      { :pre [(>= x 0)] }
      (. :java.lang.Math :sqrt x)))
  (square-root 4))
=> 2.0

;; closures
(do
  (defn pow [n]
    (fn [x] (apply * (repeat n x)))) ; closes over n

  ;; n is provided here as 2 and 3, then n goes out of scope
  (def square (pow 2))
  (def cubic (pow 3))
  (square 4))
=> 16

;; higher-order function
(do
  (def discount
    (fn [percentage]
      { :pre [(and (>= percentage 0) (<= percentage 100))] }
      (fn [price] (- price (* price percentage 0.01)))))
  ((discount 50) 300))
=> 150.0

```

SEE ALSO

defn

Same as (def name (fn name [args*] condition-map? expr*)) or (def name (fn name ([args*] condition-map? expr*)+))

defn-

Same as defn, yielding non-public def

def

Creates a global variable.

[section](#)

[top](#)

fn-about

```
(fn-about f)
```

Returns the meta information about a function

```

(fn-about and)
=> {:name "and" :ns "core" :type :macro :visibility :public :native false :class :VncMultiArityFunction :source
{:file "core" :line 482 :column 3}}

```

```
(fn-about println)
=> {:name "println" :ns "core" :type :function :visibility :public :native false :class :VncMultiArityFunction :
source {:file "core" :line 1698 :column 3}}
```

```
(fn-about +)
=> {:name "+" :ns "core" :type :function :visibility :public :native true :class :VncFunction :source {}}
```

SEE ALSO

[fn-name](#)

Returns the qualified name of a function or macro

[fn-body](#)

Returns the body (a list of forms) of a function.

[fn-args](#)

Returns the argument list of a function.

[fn-pre-conditions](#)

Returns the pre-conditions (a vector of forms) of a function.

[section](#)

[top](#)

fn-args

```
(fn-args fn)
```

Returns the argument list of a function.

Returns `nil` if `fn` is not a function or if `fn` is a native function.

```
;; single arity
(do
  (defn sum [x y]
    (+ x y))
  (fn-args (var-get sum)))
=> ({:params ["x" "y"] :variadic false})

;; single arity, vargs
(do
  (defn sum [x & z]
    (apply + x z))
  (fn-args (var-get sum)))
=> ({:params ["x"] :variadic true :variadic-name "z"})

;; multi arity
(do
  (defn sum
    ([x] x)
    ([x y] (+ x y)))

  (fn-args (var-get sum)))
=> ({:params ["x"] :variadic false} {:params ["x" "y"] :variadic false})
```

SEE ALSO

[fn-name](#)

Returns the qualified name of a function or macro

[fn-about](#)

Returns the meta information about a function

fn-body

Returns the body (a list of forms) of a function.

fn-pre-conditions

Returns the pre-conditions (a vector of forms) of a function.

[section](#)[top](#)

fn-body

```
(fn-body fn)
(fn-body fn arity)
```

Returns the body (a list of forms) of a function.

Returns `nil` if fn is not a function or if fn is a native function.

```
(do
  (defn calc [& x]
    (->> x
      (filter even?)
      (map #(* % 10)))))
  (fn-body (var-get calc)))
=> ((->> x (filter even?) (map (fn [%] (* % 10)))))
```

SEE ALSO

fn-name

Returns the qualified name of a function or macro

fn-about

Returns the meta information about a function

fn-args

Returns the argument list of a function.

fn-pre-conditions

Returns the pre-conditions (a vector of forms) of a function.

[section](#)[top](#)

fn-name

```
(fn-name f)
```

Returns the qualified name of a function or macro

```
(fn-name (fn sum [x y] (+ x y)))
=> "user/sum"
```

```
(let [f str/digit?]
  (fn-name f))
=> "str/digit?"
```

SEE ALSO

name

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[fn-about](#)

Returns the meta information about a function

[fn-body](#)

Returns the body (a list of forms) of a function.

[fn-pre-conditions](#)

Returns the pre-conditions (a vector of forms) of a function.

[section](#)[top](#)

fn-pre-conditions

```
(fn-pre-conditions fn)
(fn-pre-conditions fn arity)
```

Returns the pre-conditions (a vector of forms) of a function.

Returns `nil` if fn is not a function.

```
(do
  (defn sum [x y]
    { :pre [(> x 0) (> y 0)] }
    (+ x y))
  (fn-pre-conditions (var-get sum)))
=> [(> x 0) (> y 0)]
```

SEE ALSO

[fn-name](#)

Returns the qualified name of a function or macro

[fn-about](#)

Returns the meta information about a function

[fn-body](#)

Returns the body (a list of forms) of a function.

[section](#)[top](#)

fn?

```
(fn? x)
```

Returns true if x is a function

```
(do
  (def sum (fn [x] (+ 1 x)))
  (fn? sum))
=> true
```

[section](#)[top](#)

fnil

```
(fnil f x)
(fnil f x y)
(fnil f x y z)
```

Takes a function `f`, and returns a function that calls `f`, replacing a `nil` first argument to `f` with the supplied value `x`. Higher arity versions can replace arguments in the second and third positions (`y`, `z`). Note that the function `f` can take any number of arguments, not just the one(s) being `nil`-patched.

```
;; e.g.: change the `str/lower-case` handling of nil arguments by
;; returning an empty string instead of nil.
((fnil str/lower-case "") nil)
=> ""
```

```
((fnil + 10) nil)
=> 10
```

```
((fnil + 10) nil 1)
=> 11
```

```
((fnil + 10) nil 1 2)
=> 13
```

```
((fnil + 10) 20 1 2)
=> 23
```

```
((fnil + 10) nil 1 2 3 4)
=> 20
```

```
((fnil + 1000 100) nil nil)
=> 1100
```

```
((fnil + 1000 100) 2000 nil 1)
=> 2101
```

```
((fnil + 1000 100) nil 200 1 2)
=> 1203
```

```
((fnil + 1000 100) nil nil 1 2 3 4)
=> 1110
```

[section](#)[top](#)

fonts/download-demo-fonts

```
(fonts/download-demo-fonts dir)
(fonts/download-demo-fonts dir silent)
```

Downloads the Venice demo fonts

Family	Download family ref	Type	License
Open Sans	open-sans	TTF	Apache License v2
Roboto	roboto	TTF	Apache License v2
Source Code Pro	source-code-pro	OTF	SIL Open Font License v1.10

JetBrains Mono jetbrains-mono TTF Apache License v2

to the specified dir.

Downloads the font families from the [Font Squirrel](#) repository

URLS:

- <https://www.fontsquirrel.com/fonts/download/open-sans>
- <https://www.fontsquirrel.com/fonts/download/roboto>
- <https://www.fontsquirrel.com/fonts/download/source-code-pro>
- <https://www.fontsquirrel.com/fonts/download/jetbrains-mono>

```
(do
  (load-module :fonts)
  (fonts/download-demo-fonts (repl/libs-dir) false))
```

SEE ALSO

[fonts/download-font-family](#)

Downloads a font family manuall from the Font Squirrel (<https://www.fontsquirrel.com/>) repository

section

top

fonts/download-font-family

```
(fonts/download-font-family family-name options*)
```

Downloads a font family manuall from the [Font Squirrel](#) repository

Some useful font families:

Family	Download family ref	Type	License
Open Sans	open-sans	TTF	Apache License v2
Roboto	roboto	TTF	Apache License v2
Source Code Pro	source-code-pro	OTF	SIL Open Font License v1.10
JetBrains Mono	jetbrains-mono	TTF	Apache License v2

Options:

:extract {true,false} if true extract the TTF files from the font family ZIP, else just download the ZIP

:dir path download dir, defaults to "."

:silent {true,false} if silent is true does not print download info, defaults to true

```
(do
  (load-module :fonts)

  (fonts/download-font-family "open-sans"
    :dir (repl/libs-dir)
    :extract true
    :glob-pattern "*.ttf"
    :silent false)

  (fonts/download-font-family "roboto"
    :dir (repl/libs-dir)
    :extract true
    :glob-pattern "*.ttf"
    :silent false)

  (fonts/download-font-family "source-code-pro"
```

```

      :dir (repl/libs-dir)
      :extract true
      :glob-pattern "*.otf"
      :silent false)

(fonts/download-font-family "jetbrains-mono"
  :dir (repl/libs-dir)
  :extract true
  :glob-pattern "*.ttf"
  :silent false))

```

SEE ALSO

[fonts/download-demo-fonts](#)

Downloads the Venice demo fonts

[section](#)

[top](#)

for

```
(for seq-exprs body-expr)
```

List comprehension. Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a sequence of evaluations of expr.

Collections are iterated in a nested fashion, rightmost fastest, and nested coll-exprs can refer to bindings created in prior binding-forms.

Supported modifiers are: `:let` [binding-form expr ...], `:while` test, `:when` test.

List comprehensions are used when multiple lists have to be processed down to a single list and some filtering has to be applied.

```

(for [x (range 10)] x)
=> [0 1 2 3 4 5 6 7 8 9]

(for [x (range 5)] (* x 2))
=> [0 2 4 6 8]

(for [x (range 10) :when (odd? x) :let [x (* x 2)]] x)
=> [2 6 10 14 18]

(for [x (range 10) :when (odd? x) :let [y (* x 2)]] [x y])
=> [[1 2] [3 6] [5 10] [7 14] [9 18]]

(for [x [1 2 3] :when (odd? x) y [1 2 3 4] :when (even? y)] [x y])
=> [[1 2] [1 4] [3 2] [3 4]]

(for [x (range 10) :while (< x 4) y (range 2)] [x y])
=> [[0 0] [0 1] [1 0] [1 1] [2 0] [2 1] [3 0] [3 1]]

(for [x (range 10) :while (< x 4) y (range 6) :while (< y 6) :when (even? y)] [x y])
=> [[0 0] [0 2] [0 4] [1 0] [1 2] [1 4] [2 0] [2 2] [2 4] [3 0] [3 2] [3 4]]

```

SEE ALSO

[mapcat](#)

Returns the result of applying concat to the result of applying map to fn and colls. Thus function fn should return a collection.

[doseq](#)

Repeatedly executes body (presumably for side-effects) with bindings and filtering as provided by list-comp. Does not retain the head ...

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[section](#)[top](#)

force

(force x)

If x is a delay, returns its value, else returns x

```
(do
  (def x (delay (println "working...") 100))
  (force x))
```

working...

=> 100

```
(force (+ 1 2))
```

=> 3

SEE ALSO

[delay](#)

Takes a body of expressions and yields a Delay object that will invoke the body only the first time it is forced (with force or deref ...

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[section](#)[top](#)

formal-type

(formal-type object)

Returns the *formal type* of a Java object.

The *formal type* of an object is defined as the explicit Java return type given by the function's definition. The *formal type* may differ from the real type of the returned Java object. A type cast will also change the object's formal type and set it to the cast type.

Venice must honor Java's static type system while interacting with Java objects. Therefore Venice adheres to *formal types* strictly when calling methods of Java objects.

Venice

```
;; The Circle constructor returns an object of type Circle
(let [c (. :Circle :new 1.5)]
  (. c :area)      ;; OK   Circle::area()
  (. c :radius))   ;; OK   Circle::radius()

;; Builder::circle returns an object of the formal type Shape
(let [c (. :Builder :circle 1.5)]
  (. c :area)      ;; OK   Shape::area()
  (. c :radius))   ;; FAIL  Shape::radius(), undefined method
```

Java

```

public class Builder {
    public static Shape circle(double radius) {
        return new Circle(radius);
    }
}

public interface Shape {
    double area();
}

public class Circle implements Shape {
    public Circle(double radius) {...}
    public double area() {...}
    public double radius() {...}
}

```

```

(do
  (import :java.awt.Point)
  (import :java.awt.geom.Point2D)

  ;; upcasting :java.awt.Point to :java.awt.geom.Point2D
  ;; Point2D does not define the translate method!
  (let [p1 (. :Point :new 1.0 1.0)
        p2 (cast :Point2D p1)]
    (println "p1 ->" p1)
    (println "p2 ->" p2)
    (println "Formal type p1 ->" (formal-type p1))
    (println "Formal type p2 ->" (formal-type p2))
    (println "p1' ->" (doto p1 (. :translate 2.0 2.0)))
    ;; the translate method is not defined by Point2D
    ;; and will fail with a JavaMethodInvocationException!
    ;; (doto p2 (. :translate 2.0 2.0))
  ))
p1 -> java.awt.Point[x=1,y=1]
p2 -> java.awt.Point[x=1,y=1]
Formal type p1 -> :java.awt.Point
Formal type p2 -> :java.awt.geom.Point2D
p1' -> java.awt.Point[x=3,y=3]
=> nil

```

SEE ALSO

[remove-formal-type](#)

Removes the formal type from a Java object.

[cast](#)

Cast a Java object to a specific type

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[section](#)

[top](#)

format-micro-time

```

(format-micro-time time)
(format-micro-time time & options)

```

Formats a time given in microseconds as long or double.

Options:

`:precision p` e.g `:precision 4` (defaults to 3)

```
(format-micro-time 203)
=> "203μs"
```

```
(format-micro-time 20389.0 :precision 2)
=> "0.02ms"
```

```
(format-micro-time 20389 :precision 2)
=> "0.02ms"
```

```
(format-micro-time 20389 :precision 0)
=> "0ms"
```

```
(format-micro-time 20386766)
=> "20.387s"
```

```
(format-micro-time 20386766 :precision 2)
=> "20.39s"
```

```
(format-micro-time 20386766 :precision 6)
=> "20.386766s"
```

SEE ALSO

[format-milli-time](#)

Formats a time given in milliseconds as long or double.

[format-nano-time](#)

Formats a time given in nanoseconds as long or double.

[section](#)

[top](#)

format-milli-time

```
(format-milli-time time)
(format-milli-time time & options)
```

Formats a time given in milliseconds as long or double.

Options:

`:precision p` e.g `:precision 4` (defaults to 3)

```
(format-milli-time 203)
=> "203ms"
```

```
(format-milli-time 20389.0 :precision 2)
=> "20.39s"
```

```
(format-milli-time 20389 :precision 2)
=> "20.39s"
```

```
(format-milli-time 20389 :precision 0)
=> "20s"
```

SEE ALSO

[format-micro-time](#)

Formats a time given in microseconds as long or double.

[format-nano-time](#)

Formats a time given in nanoseconds as long or double.

[section](#)[top](#)

format-nano-time

```
(format-nano-time time)
(format-nano-time time & options)
```

Formats a time given in nanoseconds as long or double.

Options:

:precision p e.g :precision 4 (defaults to 3)

```
(format-nano-time 203)
=> "203ns"
```

```
(format-nano-time 20389.0 :precision 2)
=> "20.39µs"
```

```
(format-nano-time 20389 :precision 2)
=> "20.39µs"
```

```
(format-nano-time 20389 :precision 0)
=> "20µs"
```

```
(format-nano-time 203867669)
=> "203.868ms"
```

```
(format-nano-time 20386766988 :precision 2)
=> "20.39s"
```

```
(format-nano-time 20386766988 :precision 6)
=> "20.386767s"
```

SEE ALSO

[format-milli-time](#)

Formats a time given in milliseconds as long or double.

[format-micro-time](#)

Formats a time given in microseconds as long or double.

[nano-time](#)

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

[section](#)[top](#)

fourth

```
(fourth coll)
```

Returns the fourth element of coll.

```
(fourth nil)
=> nil

(fourth [])
=> nil

(fourth [1 2 3 4 5])
=> 4

(fourth '())
=> nil

(fourth '(1 2 3 4 5))
=> 4
```

[section](#)

[top](#)

frequencies

(frequencies coll)

Returns a map from distinct items in coll to the number of times they appear.

```
(frequencies [:a :b :a :a])
=> {:a 3 :b 1}

;; Turn a frequency map back into a coll.
(mapcat (fn [[x n]] (repeat n x)) {:a 2 :b 1 :c 3})
=> (:a :a :b :c :c :c)
```

[section](#)

[top](#)

future

(future fn)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result and return it on all subsequent calls to deref. If the computation has not yet finished, calls to deref will block, unless the variant of deref with timeout is used.

Thread local vars will be inherited by the future child thread. Changes of the child's thread local vars will not be seen on the parent.

```
(do
  (defn wait [] (sleep 300) 100)
  (let [f (future wait)]
    (deref f)))
=> 100

(let [f (future #(do (sleep 300) 100))]
  (deref f))
=> 100
```

```

(do
  (defn wait [x] (sleep 300) (+ x 100))
  (let [f (future (partial wait 10))]
    (deref f)))
=> 110

(do
  (defn sum [x y] (+ x y))
  (let [f (future (partial sum 3 4))]
    (deref f)))
=> 7

;; demonstrates the use of thread locals with futures
(do
  ;; parent thread locals
  (binding [a 10 b 20]
    ;; future with child thread locals
    (let [f (future (fn [] (binding [b 90] {:a a :b b})))]
      {:child @f :parent {:a a :b b}})))
=> {:parent {:a 10 :b 20} :child {:a 10 :b 90}}

```

SEE ALSO

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[done?](#)

Returns true if the future or promise is done otherwise false

[cancel](#)

Cancels a future or a promise

[cancelled?](#)

Returns true if the future or promise is cancelled otherwise false

[future-task](#)

Takes a function f without arguments and yields a future object that will invoke the function in another thread.

[promise](#)

Returns a promise object that can be read with deref, and set, once only, with deliver. Calls to deref prior to delivery will block, ...

[futures-fork](#)

Creates a list of count futures. The worker factory is single argument function that gets the worker index (0..count-1) as argument ...

[futures-wait](#)

Waits for all futures to get terminated. If the waiting thread is interrupted the futures are cancelled.

[section](#)

[top](#)

future-task

```

(future-task f completed-fn)
(future-task f success-fn failure-fn)

```

Takes a function f without arguments and yields a future object that will invoke the function in another thread.

If a single completed function is passed it will be called with the future as its argument as soon as the future has completed. If a success and a failure function are passed either the success or failure function will be called as soon as the future has completed. Upon success the success function will be called with the future's result as its argument, upon failure the failure function will be called with the exception as its argument.

In combination with a queue a completion service can be built. The tasks appear in the queue in the order they have completed.

Thread local vars will be inherited by the future child thread. Changes of the child's thread local vars will not be seen on the parent.

```
;; building a completion service
;; CompletionService = incoming worker queue + worker threads + output data queue
(do
  (def q (queue 10))
  (defn process [s v] (sleep s) v)
  (defn failure [s m] (sleep s) (throw (ex :VncException m))))
  (future-task (partial process 200 2) #(offer! q %) #(offer! q %))
  (future-task (partial process 400 4) #(offer! q %) #(offer! q %))
  (future-task (partial process 100 1) #(offer! q %) #(offer! q %))
  (future-task (partial failure 300 "Failed 3") #(offer! q %) #(offer! q %))
  (println (poll! q 1000))
  (println (poll! q 1000))
  (println (poll! q 1000))
  (println (poll! q 1000)))
1
2
com.github.jlangch.venice.VncException: Failed 3
4
=> nil

;; building a completion service (future-task API variant)
(do
  (def q (queue 10))
  (defn process [s v] (sleep s) v)
  (defn failure [s m] (sleep s) (throw (ex :VncException m))))
  (defn print_result [f] (try (println @f) (catch :Exception e (println e))))
  (future-task (partial process 200 2) #(offer! q %))
  (future-task (partial process 400 4) #(offer! q %))
  (future-task (partial process 100 1) #(offer! q %))
  (future-task (partial failure 300 "Failed 3") #(offer! q %))
  (print_result (poll! q 1000))
  (print_result (poll! q 1000))
  (print_result (poll! q 1000))
  (print_result (poll! q 1000)))
1
2
com.github.jlangch.venice.VncException: Failed 3
4
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[section](#)

[top](#)

future?

(future? f)

Returns true if f is a Future otherwise false

```
(future? (future (fn [] 100)))
=> true
```

[section](#)

[top](#)

futures-fork

```
(futures-fork count worker-factory-fn)
```

Creates a list of count futures. The worker factory is single argument function that gets the worker index (0..count-1) as argument and returns a worker function. Returns a list with the created futures.

```
(do
  (def mutex 0)
  (defn log [& xs]
    (locking mutex (println (apply str xs))))
  (defn factory [n]
    (fn [] (log "Worker" n)))
  (apply futures-wait (futures-fork 3 factory)))
Worker0
Worker2
Worker1
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[futures-wait](#)

Waits for all futures to get terminated. If the waiting thread is interrupted the futures are cancelled.

[section](#) [top](#)

futures-thread-pool-info

```
(futures-thread-pool-info)
```

Returns the thread pool info of the ThreadPoolExecutor serving the futures.

<i>core-pool-size</i>	the number of threads to keep in the pool, even if they are idle
<i>maximum-pool-size</i>	the maximum allowed number of threads
<i>current-pool-size</i>	the current number of threads in the pool
<i>largest-pool-size</i>	the largest number of threads that have ever simultaneously been in the pool
<i>active-thread-count</i>	the approximate number of threads that are actively executing tasks
<i>scheduled-task-count</i>	the approximate total number of tasks that have ever been scheduled for execution
<i>completed-task-count</i>	the approximate total number of tasks that have completed execution

```
(futures-thread-pool-info)
=> {:core-pool-size 0 :maximum-pool-size 200 :current-pool-size 4 :largest-pool-size 4 :active-thread-count 0 :
scheduled-task-count 24 :completed-task-count 24}
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[section](#) [top](#)

futures-wait

```
(futures-wait & futures)
```

Waits for all futures to get terminated. If the waiting thread is interrupted the futures are cancelled.

```
(do
  (def mutex 0)
  (defn log [& xs]
    (locking mutex (println (apply str xs))))
  (defn factory [n]
    (fn [] (log "Worker" n)))
  (apply futures-wait (futures-fork 3 factory)))
Worker0
Worker2
Worker1
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[futures-fork](#)

Creates a list of count futures. The worker factory is single argument function that gets the worker index (0..count-1) as argument ...

[section](#)[top](#)

gc

```
(gc)
```

Run the Java garbage collector.

```
(gc)
=> nil
```

[section](#)[top](#)

gensym

```
(gensym)
(gensym prefix)
```

Generates a symbol.

```
(gensym)
=> G__31814
```

```
(gensym "prefix_")
=> prefix_31843
```

[section](#)[top](#)

geoip/addr-ranges->trie

(geoip/addr-ranges->trie ranges)

Creates a trie map from a sequence of address ranges.

```
(do
  (def private-ip4-trie (geoip/addr-ranges->trie geoip/private-ip4-addresses))

  (defn private-ip? [ip]
    (some? (cidr/lookup-reverse private-ip4-trie ip)))

  (private-ip? "192.168.0.1"))
=> true
```

[section](#)

[top](#)

geoip/build-maxmind-city-db-url

(geoip/build-maxmind-city-db-url)

Build the URL for downloading the MaxMind city GEO IP database.

The download requires an account ID and a license key that is sent as part of the basic authentication.

The license key to download the free MaxMind GeoLite databases can be obtained from the [MaxMind](#) home page.

```
(do
  (load-module :geoip)
  (geoip/build-maxmind-city-db-url))
=> "https://download.maxmind.com/geoip/databases/GeoLite2-City-CSV/download?suffix=zip"
```

SEE ALSO

[geoip/download-maxmind-db](#)

Downloads the MaxMind country or city GEO IP database. Returns the DB as bytebuffer. The type is either :country or :city.

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[section](#)

[top](#)

geoip/build-maxmind-country-db-url

(geoip/build-maxmind-country-db-url)

Build the URL for the MaxMind country GEO IP database.

The download requires an account ID and a license key that is sent as part of the basic authentication.

The license key to download the free MaxMind GeoLite databases can be obtained from the [MaxMind](#) home page.

```
(do
  (load-module :geoup)
  (geoup/build-maxmind-country-db-url))
=> "https://download.maxmind.com/geoup/databases/GeoLite2-Country-CSV/download?suffix=zip"
```

SEE ALSO

[geoup/download-maxmind-db](#)

Downloads the MaxMind country or city GEO IP database. Returns the DB as bytebuffer. The type is either :country or :city.

[geoup/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[section](#)[top](#)

geoup/country-to-location-resolver

```
(geoup/country-to-location-resolver location-csv)
```

Returns a resolve function that resolves countries given by a country 2-digit ISO code to its latitude/longitude location. The resolve function returns the latitude/longitude or nil if the country is not supported.

The resolver loads Google country database and caches the data for location resolves.

```
(do
  (def rv (geoup/country-to-location-resolver geoup/download-google-country-db))
  (rv "PL")) ;; => ["51.919438", "19.145136"]
```

SEE ALSO

[geoup/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoup/ip-to-country-resolver](#)

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information ...

[geoup/ip-to-country-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated country and latitude/longitude location. The resolve function ...

[geoup/ip-to-city-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoup/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[section](#)[top](#)

geoup/download-google-country-db-to-csvfile

```
(geoup/download-google-country-db-to-csvfile csvfile)
```

Downloads the Google country GPS database to the given CSV file location. The database holds a mapping from country to location (latitude /longitude).

The Google country database URL is defined in the global var 'geoup/google-country-url'.

```
(do
  (load-module :geoup)
  (geoup/download-google-country-db-to-csvfile "./country-gps.csv"))
```

SEE ALSO

[geoiP/download-google-country-db](#)

Downloads the Google country database. The database holds a mapping from country to location (latitude/longitude).

[section](#)

[top](#)

geoiP/download-maxmind-db

```
(geoiP/download-maxmind-db type account-id lic-key)
```

Downloads the MaxMind country or city GEO IP database. Returns the DB as bytearray. The type is either :country or :city.

The download requires an account ID and a license key that is sent as part of the basic authentication.

The license key to download the free MaxMind GeoLite databases can be obtained from the [MaxMind](#) home page.

Please ensure that your servers can make HTTPS connections to the following hostname:

mm-prod-geoiP-databases.a2649acb697e2c09b632799562c076f2.r2.cloudflarestorage.com

```
(do
  (load-module :geoiP)
  (geoiP/download-maxmind-db :country
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY"))
```

SEE ALSO

[geoiP/build-maxmind-country-db-url](#)

Build the URL for the MaxMind country GEO IP database.

[geoiP/build-maxmind-city-db-url](#)

Build the URL for downloading the MaxMind city GEO IP database.

[section](#)

[top](#)

geoiP/download-maxmind-db-to-zipfile

```
(geoiP/download-maxmind-db-to-zipfile zipfile type account-id lic-key)
```

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

The download requires your personal MaxMind license key. The license to download the free MaxMind GeoLite databases can be obtained from the [MaxMind](#) home page.

```
(do
  (load-module :geoiP)
  (geoiP/download-maxmind-db-to-zipfile "./geoiP-country.zip"
    :country
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY"))
```

SEE ALSO

[geoiP/build-maxmind-country-db-url](#)

Build the URL for the MaxMind country GEO IP database.

[geoiP/build-maxmind-city-db-url](#)

Build the URL for downloading the MaxMind city GEO IP database.

[section](#)

[top](#)

geoip/ip-to-city-loc-resolver

(geoip/ip-to-city-loc-resolver geoip-zip)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function returns the city and the latitude/longitude or nil if no data is found.

The MindMax city geoip-zip may be a bytebuf, a file, a string (file path) or an InputStream.

The resolver loads the MindMax IPv4 and IPv6 city database and caches the data for IP address resolves.

As of July 2020 the MaxMind city database has:

2'917'097	IPv4 blocks
459'294	IPv6 blocks
118'189	cities

Note:

The MaxMind city IPv4 and IPv6 databases have 220MB of size on disk. It takes considerable time to load the data. Preprocessed and ready to work in the GEO IP modules ~3GB of memory is required.

Once the resolver has loaded the data the lookups are very fast.

```
(do
  (def rv (geoip/ip-to-city-loc-resolver "./geoip-city.zip"))

  (rv "192.241.235.46")) ;; => {:ip "192.241.235.46"
                                ;;   :loc ["37.7353" "-122.3732"]
                                ;;   :country-name "United States"
                                ;;   :country-iso "US"
                                ;;   :region "California"
                                ;;   :city "San Francisco"}
```

SEE ALSO

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoip/ip-to-country-resolver](#)

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information ...

[geoip/ip-to-country-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated country and latitude/longitude location. The resolve function ...

[geoip/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoip/country-to-location-resolver](#)

Returns a resolve function that resolves countries given by a country 2-digit ISO code to its latitude/longitude location. The resolve ...

[section](#)

[top](#)

geoip/ip-to-city-loc-resolver-mem-optimized

(geoip/ip-to-city-loc-resolver-mem-optimized geoip-zip)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function returns the city and the latitude/longitude or nil if no data is found.

The MindMax city geoip-zip may be a bytebuf, a file, a string (file path) or an InputStream.

The resolver loads the MindMax IPv4 and IPv6 city database and caches the data for IP address resolves.

As of July 2020 the MaxMind city database has:

2'917'097	IPv4 blocks
459'294	IPv6 blocks
118'189	cities

Note:

The MaxMind city IPv4 and IPv6 databases have 220MB of size on disk. It takes considerable time to load the data. This is a memory optimized resolver version on the cost of performance.

For best performance on the cost of memory use the resolver 'geoip/ip-to-city-loc-resolver' instead!

```
(do
  (def rv (geoip/ip-to-city-loc-resolver-mem-optimized "./geoip-city.zip"))

  (rv "192.241.235.46")) ;; => {:ip "192.241.235.46"
                                ;;   :loc ["37.7353" "-122.3732"]
                                ;;   :country-name "United States"
                                ;;   :country-iso "US"
                                ;;   :region "California"
                                ;;   :city "San Francisco"}
```

SEE ALSO

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoip/ip-to-country-resolver](#)

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information ...

[geoip/ip-to-country-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated country and latitude/longitude location. The resolve function ...

[geoip/ip-to-city-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoip/country-to-location-resolver](#)

Returns a resolve function that resolves countries given by a country 2-digit ISO code to its latitude/longitude location. The resolve ...

[section](#)

[top](#)

geoip/ip-to-country-loc-resolver

```
(geoip/ip-to-country-loc-resolver geoip-zip location-csv)
```

Returns a resolve function that resolves an IP address to its associated country and latitude/longitude location. The resolve function returns the country and the latitude/longitude or nil if no data is found.

The MindMax country geoip-zip may be a bytebuf, a file, a string (file path) or an InputStream.

The resolver loads the MindMax IPv4 and IPv6 country and the Google country database and caches the data for IP address resolves.

```
(do
  (def rv (geoip/ip-to-country-loc-resolver
            "./geoip-country.zip"
            (geoip/download-google-country-db)))

  (rv "91.223.55.1")) ;; => {:ip "91.223.55.6"
```



```
;;      :loc ["51.919438" "19.145136"]
;;      :country-name "Poland"
;;      :country-iso "PL"}
```

SEE ALSO

[geoup/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoup/ip-to-country-resolver](#)

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information ...

[geoup/ip-to-city-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoup/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoup/country-to-location-resolver](#)

Returns a resolve function that resolves countries given by a country 2-digit ISO code to its latitude/longitude location. The resolve ...

[section](#)

[top](#)

geoup/ip-to-country-resolver

```
(geoup/ip-to-country-resolver geoup-zip)
```

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information for a given IP address.

The MindMax country geoup-zip may be a bytebuf, a file, a string (file path) or an InputStream.

The resolver loads the MindMax IPv4 and IPv6 country databases and caches the data for subsequent IP resolves.

As of July 2020 the MaxMind country database has:

303'448	IPv4 blocks
107'641	IPv6 blocks
253	countries

```
(do
  (def rv (geoup/ip-to-country-resolver "./geoup-country.zip"))
  (rv "91.223.55.1")) ;; => { :country-name "Poland"
                           ;;      :country-iso "PL" }
```

SEE ALSO

[geoup/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoup/ip-to-country-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated country and latitude/longitude location. The resolve function ...

[geoup/ip-to-city-loc-resolver](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoup/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves an IP address to its associated city and latitude/longitude location. The resolve function ...

[geoup/country-to-location-resolver](#)

Returns a resolve function that resolves countries given by a country 2-digit ISO code to its latitude/longitude location. The resolve ...

[section](#)

[top](#)

geoip/map-location-to-numeric

```
(map-location-to-numeric loc)
```

Maps a location to numerical coordinates. A location is given as a vector of a latitude and a longitude.

Returns a location vector with a numerical latitude and a longitude.

```
(do
  (load-module :geoip)
  (geoip/map-location-to-numeric ["51.919438", "19.145136"]))
=> [51.919438 19.145136]
```

[section](#)[top](#)

geoip/parse-maxmind-city-db

```
(geoip/parse-maxmind-city-db zip)
```

Parses the MaxMind city-location CSV file. Returns a map with the city geoname-id as key and the city/country data as value.

Return:

```
{ "2643743" {:country-iso "GB" :country-name "England"
              :region "England" :city "London"}
  "2661881" {:country-iso "CH" :country-name "Switzerland"
              :region "Aargau" :city "Aarau"} }
```

```
(do
  (load-module :geoip)
  (geoip/download-maxmind-db-to-zipfile "./geoip-city.zip"
    :city
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY")
  (geoip/parse-maxmind-city-db "./geoip-city.zip"))
```

SEE ALSO

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoip/parse-maxmind-country-db](#)

Parses the MaxMind country-location CSV file. Returns a map with the country geoname-id as key and the country data as value.

[section](#)[top](#)

geoip/parse-maxmind-city-ip-db

```
(geoip/parse-maxmind-city-ip-db ip-type zip maxmind-cities)
```

Parses the MaxMind city IP blocks database. Expects a MaxMind city IP database zip. ip-type is either :IPv4 or :IPv6. The zip may be a bytebuf, a file, a string (file path) or an InputStream.

The maxmind-countries are optional and map the geoname-id to country data.

Returns a trie datastructure with the CIDR address as the key and a map with city/country data as the value.

maxmind-cities:

```
{ "2643743" {:country-iso "GB" :country-name "England"
           :region "England" :city "London"}
  "2661881" {:country-iso "CH" :country-name "Switzerland"
           :region "Aargau"  :city "Aarau"} }
```

```
(do
  (load-module :geoip)
  (geoip/download-maxmind-db-to-zipfile "./geoip-city.zip"
    :city
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY")

  (geoip/parse-maxmind-city-ip-db
    :IPv4
    "./geoip-city.zip"
    nil))

(do
  (load-module :geoip)
  (geoip/download-maxmind-db-to-zipfile "./geoip-city.zip"
    :city
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY")

  (geoip/parse-maxmind-city-ip-db
    :IPv6
    "./geoip-city.zip"
    (geoip/parse-maxmind-city-db "./geoip-city.zip")))
```

SEE ALSO

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoip/parse-maxmind-city-db](#)

Parses the MaxMind city-location CSV file. Returns a map with the city geoname-id as key and the city/country data as value.

[geoip/parse-maxmind-country-ip-db](#)

Parses the MaxMind country IP blocks database. Expects a Maxmind country IP database zip. ip-type is either :IPv4 or :IPv6. The zip ...

[section](#)

[top](#)

geoip/parse-maxmind-country-db

(geoip/parse-maxmind-country-db zip)

Parses the MaxMind country-location CSV file. Returns a map with the country geoname-id as key and the country data as value.

Return:

```
{ "49518" {:country-iso "RW" :country-name "Rwanda"}
  "51537" {:country-iso "SO" :country-name "Somalia"} }
```

```
(do
  (load-module :geoip)
  (geoip/download-maxmind-db-to-zipfile "./geoip-country.zip"
    :country
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY")

  (geoip/parse-maxmind-country-db "./geoip-country.zip"))
```

SEE ALSO

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoip/parse-maxmind-city-db](#)

Parses the MaxMind city-location CSV file. Returns a map with the city geoname-id as key and the city/country data as value.

[section](#)[top](#)

geoip/parse-maxmind-country-ip-db

```
(geoip/parse-maxmind-country-ip-db ip-type zip maxmind-countries)
```

Parses the MaxMind country IP blocks database. Expects a Maxmind country IP database zip. ip-type is either :IPv4 or :IPv6. The zip may be a bytebuf, a file, a string (file path) or an InputStream.

The maxmind-countries are optional and map the geoname-id to country data.

Returns a trie datastructure with the CIDR address as the key and a map with country data as the value.

maxmind-countries:

```
{ "49518" {:country-iso "RW" :country-name "Rwanda"}
  "51537" {:country-iso "SO" :country-name "Somalia"} }
```

Return:

```
{ 223 [ [(cidr-parse "223.255.254.0/24") {:country-iso "SG"
                                           :country-name "Singapore"}]
         [(cidr-parse "223.255.255.0/24") {:country-iso "AU"
                                           :country-name "Australia"}]
       ] }
```

```
(do
  (load-module :geoip)
  (geoip/download-maxmind-db-to-zipfile "./geoip-country.zip"
    :country
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY")

  (geoip/parse-maxmind-country-ip-db
    :IPv4
    "./geoip-country.zip"
    nil))

(do
  (load-module :geoip)
  (geoip/download-maxmind-db-to-zipfile "./geoip-country.zip"
    :country
    "YOUR-MAXMIND-ACCOUNT-ID"
    "YOUR-MAXMIND-LIC-KEY")

  (geoip/parse-maxmind-country-ip-db
    :IPv6
    "./geoip-country.zip"
    (geoip/parse-maxmind-country-db "./geoip-country.zip")))
```

SEE ALSO

[geoip/download-maxmind-db-to-zipfile](#)

Downloads the MaxMind country or city GEO IP database to the given ZIP file. The type is either :country or :city.

[geoip/parse-maxmind-country-db](#)

Parses the MaxMind country-location CSV file. Returns a map with the country geoname-id as key and the country data as value.

geoip/parse-maxmind-city-ip-db

Parses the MaxMind city IP blocks database. Expects a MaxMind city IP database zip. ip-type is either :IPv4 or :IPv6. The zip may be ...

[section](#)[top](#)

get

```
(get map key)
(get map key not-found)
```

Returns the value mapped to key, not-found or nil if key not present.

Note: `(get :x foo)` is almost twice as fast as `(:x foo)`

```
(get {:a 1 :b 2} :b)
=> 2
```

```
(get [0 1 2 3] 1)
=> 1
```

```
;; keywords act like functions on maps
(:b {:a 1 :b 2})
=> 2
```

[section](#)[top](#)

get-in

```
(get-in m ks)
(get-in m ks not-found)
```

Returns the value in a nested associative structure, where ks is a sequence of keys. Returns nil if the key is not present, or the not-found value if supplied.

```
(get-in {:a 1 :b {:c 2 :d 3}} [:b :c])
=> 2
```

```
(get-in [:a :b :c] [0])
=> :a
```

```
(get-in [:a :b [:c :d :e]] [2 1])
=> :d
```

```
(get-in {:a 1 :b {:c [4 5 6]}} [:b :c 1])
=> 5
```

[section](#)[top](#)

gradle/task

```
(gradle/task name & options)
(gradle/task name out-fn & options)
(gradle/task name out-fn err-fn throw-ex & options)
```

Runs a gradle task

```
(gradle/with-home "/Users/foo/Documents/Tools/gradle-5.6.2"
                  "/Users/foo/Documents/Projects/my-project"
  (gradle/task compile)
  (gradle/task compile "--warning-mode=all" "--stacktrace")
  (gradle/task compile println)
  (gradle/task compile println println true)
  (gradle/task compile println println true "--stacktrace"))
```

[section](#)

[top](#)

gradle/version

```
(gradle/version)
```

Returns the Gradle version

```
(gradle/with-home "/Users/foo/Documents/Tools/gradle-5.6.2"
                  "/Users/foo/Documents/Projects/my-project"
  (gradle/version))
```

[section](#)

[top](#)

gradle/with-home

```
(with-home gradle-dir proj-dir & forms)
```

Sets the Gradle home and the project directory for all subsequent forms.

```
(gradle/with-home "/Users/foo/Documents/Tools/gradle-5.6.2"
                  "/Users/foo/Documents/Projects/my-project"
  (gradle/version))
```

[section](#)

[top](#)

gradlew/run

```
(gradlew/run proj-home out-fn err-fn & args)
```

Runs one or more Gradle tasks.

Note: Use this module only for projects based on the Gradle wrapper

Arguments:

proj-home	The project directory
out-fn	a function with a single string argument that receives line by line from the process' stdout.
err-fn	a function with a single string argument that receives line by line from the process' stderr.
args	Any number of task names and Gradle options

```
(do
  (load-module :gradlew)
  (let [java-home (system-env :JAVA_11_HOME)]
    (gradlew/run "/Users/foo/projects/bar"
      println
      println
      ;; tasks
      "clean"
      "build"
      ;; options
      "--warning-mode=all"
      "--console=plain"
      "--stacktrace"
      (str "-Dorg.gradle.java.home=\"" java-home "\""))))
```

[section](#)
[top](#)

gradlew/run*

```
(gradlew/run* proj-home out-fn err-fn & args)
```

Runs one or more Gradle tasks and prints a list of the tasks and the options taken from the passed arguments.

Note: Use this module only for projects based on the Gradle wrapper

Apart from printing the passed tasks and options the function is identical to `gradlew/run`.

Arguments:

proj-home	The project directory
out-fn	a function with a single string argument that receives line by line from the process' stdout. May be nil.
err-fn	a function with a single string argument that receives line by line from the process' stderr. May be nil.
args	Any number of task names and Gradle options

```
(do
  (load-module :gradlew)
  (let [java-home (system-env :JAVA_11_HOME)]
    (gradlew/run* "/Users/foo/projects/bar"
      println
      println
      ;; tasks
      "clean"
      "build"
      ;; options
      "--warning-mode=all"
      "--console=plain"
      "--stacktrace"
      (str "-Dorg.gradle.java.home=\"" java-home "\""))))
```

[section](#)
[top](#)

gradlew/version

```
(gradlew/version)
```

Returns the Gradle version

Note: Use this module only for projects based on the Gradle wrapper

```
(do
  (load-module :gradlew)
  (gradlew/version "/Users/foo/projects/bar"))
```

[section](#)[top](#)

grep/grep

```
(grep dir file-glob line-pattern & options)
```

Search for lines that match a regular expression in text files. The search starts from a base directory and chooses all files that match a globbing pattern.

Options:

:print b e.g :print false, defaults to true

With the print option :print true, grep prints the matches to stdout in a human readable form, one line per match in the format `"{{filename}}:{{lineno}}:{{line}}"`.

With the print option :print false, grep returns the matches as a list of tuples `[[{{filename}}, {{lineno}}, {{line}}]`.

```
(do
  (load-module :grep)
  (grep/grep "/Users/foo/logs" "*.log" ".*Error.*"))
```

SEE ALSO

[grep/grep-zip](#)

Search for lines that match a regular expression in text files within ZIP files. The search chooses all files in the ZIP that match ...

[io/file-matches-glob?](#)

Returns true if the file f matches the glob pattern. f must be a file or a string (file path).

[section](#)[top](#)

grep/grep-zip

```
(grep/grep-zip dir zipfile-glob file-glob line-pattern & options)
```

Search for lines that match a regular expression in text files within ZIP files. The search chooses all files in the ZIP that match a globbing pattern. The search starts from a base directory and chooses all ZIP files that match the zipfile globbing pattern.

Options:

:print b e.g :print false, defaults to true

With the print option :print true, grep-zip prints the matches to stdout in a human readable form, one line per match in the format `"{{zipfile}}!{{filename}}:{{lineno}}:{{line}}"`.

With the print option :print false, grep returns the matches as a list of tuples `[[{{zipname}}, {{filename}}, {{lineno}}, {{line}}]`.

```
(do
  (load-module :grep)
  (grep/grep-zip "/Users/foo/logs" "logs*.zip" "**/*.log" ".*Error.*"))
```

SEE ALSO

[grep/grep](#)

Search for lines that match a regular expression in text files. The search starts from a base directory and chooses all files that ...

[io/file-matches-glob?](#)

Returns true if the file `f` matches the glob pattern. `f` must be a file or a string (file path).

[section](#)

[top](#)

group-by

```
(group-by f coll)
```

Returns a map of the elements of `coll` keyed by the result of `f` on each element. The value at each key will be a vector of the corresponding elements, in the order they appeared in `coll`.

```
(group-by count ["a" "as" "asd" "aa" "asdf" "qwer"])
=> {1 ["a"] 2 ["as" "aa"] 3 ["asd"] 4 ["asdf" "qwer"]}
```

```
(group-by odd? (range 10))
=> {false [0 2 4 6 8] true [1 3 5 7 9]}
```

```
(group-by identity (seq "abracadabra"))
=> {#\a [#\a #\a #\a #\a #\a #\a] #\b [#\b #\b] #\r [#\r #\r] #\c [#\c] #\d [#\d]}
```

[section](#)

[top](#)

halt-when

```
(halt-when pred)
(halt-when pred retf)
```

Returns a transducer that ends transduction when `pred` returns true for an input. When `retf` is supplied it must be a fn of 2 arguments - it will be passed the (completed) result so far and the input that triggered the predicate, and its return value (if it does not throw an exception) will be the return value of the transducer. If `retf` is not supplied, the input that triggered the predicate will be returned. If the predicate never returns true the transduction is unaffected.

```
(do
  (def xf (comp (halt-when #(== % 10)) (filter odd?)))
  (transduce xf conj [1 2 3 4 5 6 7 8 9]))
=> [1 3 5 7 9]
```

```
(do
  (def xf (comp (halt-when #(> % 5)) (filter odd?)))
  (transduce xf conj [1 2 3 4 5 6 7 8 9]))
=> 6
```

[section](#)

[top](#)

hash-map

```
(hash-map & keyvals)
(hash-map map)
```

Creates a new hash map containing the items.

```
(hash-map :a 1 :b 2)
=> {:a 1 :b 2}

(hash-map (sorted-map :a 1 :b 2))
=> {:a 1 :b 2}
```

hash-map?

```
(hash-map? obj)
```

Returns true if obj is a hash map

```
(hash-map? (hash-map :a 1 :b 2))
=> true
```

hexdump/dump

```
(dump s & opts)
```

Prints a hexdump of the given argument to `*out*`. Optionally supply byte offset (`:offset`, default: 0) and size (`:size`, default: `:all`) arguments. Can create hexdump from a collection of values, a bytebuf, a java.io.File, or a string representing a path to a file.

Example: `(hexdump/dump (range 100))`

```
00000000: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f .....
00000010: 1011 1213 1415 1617 1819 1a1b 1c1d 1e1f .....
00000020: 2021 2223 2425 2627 2829 2a2b 2c2d 2e2f !"#$%&'()*+,-./
00000030: 3031 3233 3435 3637 3839 3a3b 3c3d 3e3f 0123456789;<=>?
00000040: 4041 4243 4445 4647 4849 4a4b 4c4d 4e4f @ABCDEFGHJKLMNO
00000050: 5051 5253 5455 5657 5859 5a5b 5c5d 5e5f PQRSTUVWXYZ[\]^_
00000060: 6061 6263 `abc
```

```
(hexdump/dump [0 1 2 3])
```

```
(hexdump/dump (range 1000))
```

```
(hexdump/dump (range 10000) :offset 9000 :size 256)
```

```
(hexdump/dump "./img.png")
```

```
(hexdump/dump "./img.png" :offset 0 :size 64)
```

```
(try-with [ps (io/capturing-print-stream)]
  (binding [*out* ps]
    (hexdump/dump [0 1 2 3])
    (str ps)))
```

highlight

(highlight form)

Syntax highlighting. Reads the form and returns a list of (token, token-class) tuples.

Token classes:

:comment	;
:whitespaces	" ", "\n", " \n"
:string	"lorem", """"lorem"""
:number	100, 100I, 100.0, 100.23M
:constant	nil, true, false
:keyword	:alpha
:symbol	alpha
:symbol-special-form	def, loop, ...
:symbol-function-name	+, println, ...
:quote	'
:quasi-quote	`
:unquote	~
:unquote-splicing	~@
:meta	^private, ^{:arglist '() :doc "..."}
:at	@
:hash	#
:brace-begin	{
:brace-end	}
:bracket-begin	[
:bracket-end	]
:parenthesis-begin	(
:parenthesis-end	)
:unknown	anything that could not be classified

```
(highlight "(+ 10 20)")
```

```
=> ((("(:parenthesis-begin) ( "+" :symbol-function-name) (" " :whitespaces) ("10" :number) (" " :whitespaces) ("20" :number) (")" :parenthesis-end))
```

```
(highlight "(if (= 1 2) true false)")
```

```
=> ((("(:parenthesis-begin) ("(" :parenthesis-special-form) (" " :whitespaces) ("(" :parenthesis-begin) ("=" :symbol-function-name) (" " :whitespaces) ("1" :number) (" " :whitespaces) ("2" :number) (")" :parenthesis-end) (" " :whitespaces) ("true" :constant) (" " :whitespaces) ("false" :constant) (")" :parenthesis-end))
```

[section](#)

[top](#)

host-address

(host-address)

Returns this host's ip address.

```
(host-address)
```

```
=> "192.168.178.37"
```

SEE ALSO

host-name

Returns this host's name.

[section](#)

[top](#)

host-name

(host-name)

Returns this host's name.

(host-name)

```
=> "Mac.fritz.box"
```

SEE ALSO

[host-address](#)

Returns this host's ip address.

[section](#)

[top](#)

http-client-j8/build-url

(build-url url)

(build-url url query-params)

(build-url url query-params fragment)

Builds an url

Arguments:

:url An url

:url-query-params A optional map of url query parameters (name/value). The names and values will be appropriately encodes.

:url-fragment An optional url fragment

URL: https://foo.example.org:12346/crm/organisations?name=aviron#detailed

 scheme host port pathname query fragment

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])
  (hc/build-url "https://foo.example.org:12346/crm/organisations"
    { "name" "aviron 1" }))
=> "https://foo.example.org:12346/crm/organisations?name=aviron+1"
```

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])
  (hc/build-url "https://foo.example.org:12346/crm/organisations"
    { "name" "aviron 1" }
    "detailed"))
=> "https://foo.example.org:12346/crm/organisations?name=aviron+1#detailed"
```

SEE ALSO

[http-client-j8/create-authorization-bearer](#)

Create a base64 encoded bearer token for an 'Authorization' HTTP header.

[section](#)[top](#)

http-client-j8/create-authorization-bearer

(create-authorization-bearer token)
(create-authorization-bearer user password)

Create a base64 encoded bearer token for an 'Authorization' HTTP header.

User: "john.doe"
Password: "12345"

=> "Bearer am9obi5kb2U6MTIzNDU="

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])
  (str "Bearer " (hc/create-authorization-bearer "john.doe" "12345"))))
```

=> "Bearer am9obi5kb2U6MTIzNDU="

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])
  (str "Bearer " (hc/create-authorization-bearer "rfd3@90000"))))
```

=> "Bearer cmZkM0A5MDAwMA=="

SEE ALSO

[http-client-j8/build-url](#)

Builds an url

[section](#)[top](#)

http-client-j8/process-server-side-events

(process-server-side-events response handler)

Processes server side events (SSE) and calls for every event the handler 'handler'.

Note: The response must be of the mimetype "text/event-stream" otherwise the processor throws an exception!

The event handler is a three argument function:

```
(defn handler [type event event-count] ...)
```

type the notification type:

- `:opened` - streaming started
- `:data` - streamed event
- `:closed` - streaming closed by the server

event the streamed event, available only if the notification type is `:data`, else `nil`

event-count the streamed event count, starting with 1 and incremented with every event sent

If the event handler returns the value `:stop` the processor stops handling any further events and closes the data stream to signal the server not to send any further events and close the server side stream as well.

If the event handler throws an exception while processing a data event the event processing will be stopped.

The handler blocks until the stream is either closed by the server or by the client. The timeout can be controlled via the connections' `:read-timeout` parameter.

Server side events are passed as maps to the handler. E.g. :

```
{ :id      "1"
  :event   "score"
  :data    [ "GOAL Liverpool 1 - 1 Arsenal"
            "GOAL Manchester United 3 - 3 Manchester City" ] }
```

Warning:

When not used over HTTP/2, SSE suffers from a limitation to the maximum number of open connections, which can be especially painful when opening multiple tabs, as the limit is per browser and is set to a very low number (6). The issue has been marked as "Won't fix" in Chrome and Firefox. This limit is per browser + domain, which means that you can open 6 SSE connections across all of the tabs.

When using HTTP/2, the maximum number of simultaneous HTTP streams is negotiated between the server and the client (defaults to 100).

The Java 8 Http Client does not support HTTP/2!

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :get
                        "http://localhost:8080/events"
                        :headers { "Accept"      "text/event-stream"
                                  "Cache-Control" "no-cache"
                                  "Connection"   "keep-alive"}
                        :conn-timeout 0
                        :read-timeout 0
                        :debug true)]
    (println "Status:" (:http-status response))

    ;; process the first 10 events and close the stream
    (hc/process-server-side-events
     response
     (fn [type event event-count]
       (case type
         :opened (do (println "\nStreaming started")
                     :ok)
         :data   (do (println "Event: " (pr-str event))
                     ;; only process 10 events
                     (if (< event-count 10) :ok :stop))
         :closed (do (println "Streaming closed")
                     :ok))))))
```

SEE ALSO

[http-client-j8/slurp-response](#)

Slurps the response data from the response' input stream.

[http-client-j8/send](#)

Send a request

[section](#)

[top](#)

http-client-j8/send

(send method uri & options)

Send a request

Request Options:

:url-query-params	A map of url query parameters (name/value). The names and values will be appropriately encoded.
:url-fragment	An url fragment. The fragment will be appropriately encoded.
:headers	A map of request headers. Headers can be single- or multi-value (comma separated): { "X-Header-1" "value1" "X-Header-2" "value1, value2, value3" }
:body	An optional body to send with the request The body may be of type <i>string</i> , <i>bytebuf</i> , <code>:java.io.InputStream</code>
:conn-timeout	An optional connection timeout in milliseconds
:read-timeout	An optional read timeout in milliseconds
:follow-redirects	Sets whether HTTP redirects (requests with response code 3xx) should be automatically followed.
:hostname-verifier	Sets the hostname verifier. An object of type <code>:javax.net.ssl.HostnameVerifier</code> . Use only for HTTPS requests
:ssl-socket-factory	Sets the SSL socket factory. An object of type <code>:javax.net.ssl.SSLSocketFactory</code> . Use only for HTTPS requests
:use-caches	A boolean indicating whether or not to allow caching. Defaults to false
:user-agent	User agent. Defaults to "Venice HTTP client (legacy)"
:debug	Debug true/false. Defaults to false. In debug mode prints the HTTP request and response data

Returns a map with the response fields:

:http-status	The HTTP status (a long)
:content-type	The content type. E.g.: "text/plain; charset=utf8"
:content-type-mimetype	The content type's mimetype. E.g.: "text/plain"
:content-type-charset	The content type's charset. E.g.: :utf-8
:content-encoding	The content transfer encoding (a keyword), if available else nil. E.g.: :gzip or :deflate
:content-length	The content length (a long), if available else -1
:headers	A map of headers. key: header name, value: list of header values
:data-stream	The response data input stream. If the response content encoding is 'gzip' due to a request header "Accept-Encoding: gzip" wrap the data stream with a gzip input stream: (io/wrap-is-with-gzip-input-stream (:data-stream response)) to uncompress the data. See <code>http-client-j8/slurp-response</code> to painlessly process responses.

URL: `https://foo.example.org:12346/crm/organisations?name=aviron#detailed`
 | | | | |
 scheme host port pathname query fragment

```
;; GET (get)
;;
;; 1/ Start the example REST Server {Venice}/doc/examples/scripts/rest-webapp.venice
;;    (drag & drop the file into a REPL and HIT [RET] to start it)
;; 2/ Run the Http Client commands listed in these examples
;;    (run the Http Client commands from a 2nd REPL)
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :get
                          "http://localhost:8080/employees"
                          :headers { "Accept" "application/json, text/plain" }
                          :query-params { "status" "active" }
                          :debug true)
        status   (:http-status response)]
```

```

    (println "Status:" status)
    (println (slurp-response response :json-parse-mode :pretty-print))))

;; POST (create)
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :post
                          "http://localhost:8080/employees"
                          :headers {"Accept" "application/json, text/plain"
                                     "Content-Type" "application/json"}
                          :body (json/write-str { "name" "hanna",
                                                  "role" "secretary" })
                          :debug true)
        status (:http-status response)]
    (println "Status:" status)
    (println (hc/slurp-response response :json-parse-mode :pretty-print))))

;; PUT (update)
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :put
                          "http://localhost:8080/employees/1001"
                          :headers {"Accept" "application/json, text/plain"
                                     "Content-Type" "application/json"}
                          :body (json/write-str { "id" "1001",
                                                  "name" "john",
                                                  "role" "clerk" })
                          :debug true)
        status (:http-status response)]
    (println "Status:" status)
    (println (hc/slurp-response response :json-parse-mode :pretty-print))))

;; DELETE (delete)
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :delete
                          "http://localhost:8080/employees/1000"
                          :headers { "Accept" "text/plain" }
                          :debug true)
        status (:http-status response)]
    (println "Status:" status)
    (println (hc/slurp-response response))))

;; GET over SSL
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])
  (load-module :java ['java :as 'j])

  (import :com.github.jlangch.venice.util.ssl.CustomHostnameVerifier)
  (import :com.github.jlangch.venice.util.ssl.Server_X509TrustManager)
  (import :com.github.jlangch.venice.util.ssl.TrustAll_X509TrustManager)
  (import :com.github.jlangch.venice.util.ssl.SSLSocketFactory)
  (import :java.security.cert.X509Certificate)

  (defn verify-host [hostname]
    (case hostname
      "localhost" true
      "foo.org" true
      false))

```



```

(defn check-trust-server [certs auth-type]
  (doseq [c certs] (. c :checkValidity))
  (any? #(= "Foo" (. (% :getIssuerDN) :getName)) certs))

(let [trust-manager-all (. :TrustAll_X509TrustManager :new)
      trust-manager-server (. :Server_X509TrustManager :new (j/as-bipredicate check-trust-server))
      hostname-verifier (. :CustomHostnameVerifier :new verify-host)
      response (hc/send :get
                        "https://localhost:8080/employees"
                        :headers { "Accept" "application/json, text/plain" }
                        :hostname-verifier hostname-verifier
                        :ssl-socket-factory (. :SSLSocketFactory trust-manager-all)
                        :debug true)
      status (:http-status response)]
  (println "Status:" status)
  (println (hc/slurp-response response))))

;; OAuth blueprint
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (defn get-access-token [api-key api-key-secret]
    (let [encoded-secret (-> (str api-key ":" api-key-secret)
                              (bytebuf-from-string :utf-8)
                              (str/encode-base64))
          response (hc/send :post
                            "https://.../oauth2/token"
                            :headers { "Accept" "application/json, text/plain"
                                      "Authorization" (str "Basic " encoded-secret)
                                      "Content-Type" "application/x-www-form-urlencoded" }
                            :body "grant_type=client_credentials")
          status (:http-status response)
          mimetype (:content-type-mimetype response)
          charset (:content-type-charset response)]
      (if (and (= 200 status) (= "application/json" mimetype))
        (as-> (:data-stream response) v
              (hc/slurp-json v charset)
              (get v "access_token"))
        (throw (ex VncException "Failed to get OAuth access token")))))

  (defn list-member [access-token list-id]
    (let [response (hc/send :get
                          (str "https://.../1.1/lists/members.json?list_id=" list-id)
                          :headers { "Accept" "application/json, text/plain"
                                    "Authorization" (str "Bearer " accessToken)})
          status (:http-status response)]
      (println "Status:" status)
      (println (hc/slurp-response response :json-parse-mode :pretty-print)))))

```

SEE ALSO

[http-client-j8/upload-file](#)

Upload a file.

[http-client-j8/upload-multipart](#)

Upload multiple parts.

[http-client-j8/slurp-response](#)

Slurps the response data from the response' input stream.

[http-client-j8/build-url](#)

Builds an url

[http-client-j8/create-authorization-bearer](#)

http-client-j8/slurp-response

(slurp-response response & options)

Slurps the response data from the response' input stream.

Returns the data according to the mimetype and charset of the 'Content-Type' response header.

Handles a 'Content-Encoding' transparently. Supports the encodings 'gzip' and 'deflate'. Other encodings are rejected with an exception.

application/xml	Returns a string according to the content type charset
application/json	Returns the JSON response according to the content type charset. Depending on the option <code>:json-parse-mode</code> returns the JSON parsed to a Venice map, as a JSON pretty printed string, or as a raw JSON string
text/plain	Returns a string according to the content type charset
text/html	Returns a string according to the content type charset
text/xml	Returns a string according to the content type charset
text/csv	Returns a string according to the content type charset
text/css	Returns a string according to the content type charset
text/json	Returns the JSON response according to the content type charset. Depending on the option <code>:json-parse-mode</code> returns the JSON parsed to a Venice map, as a JSON pretty printed string, or as a raw JSON string
text/event-stream	Throws an exception. An event stream can not be slurped. Use the function <code>process-server-side-events</code> instead!
else	Returns the response data as a byte buffer

Options:

:json-parse-mode	The option is used with JSON mimetypes. <code>:data</code> - parse the response to a Venice data map <code>:raw</code> - return the reponse as received <code>:pretty-print</code> - return a pretty print JSON string Defaults to <code>:data</code>
:json-key-fn	A single argument function that transforms JSON property names. This option is only available in <code>:data</code> parse mode E.g.: <code>:json-key-fn keyword</code>

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :get
                        "http://localhost:8080/employees"
                        :headers {"Accept" "application/json, text/plain"})
        status (:http-status response)]
    (println "Status:" status)
    (println (hc/slurp-response response :json-parse-mode :pretty-print))))

(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/send :get
                        "http://localhost:8080/employees"
                        :headers {"Accept" "application/json, text/plain"})
```

```
status (:http-status response)]  
(println "Status:" status)  
(prn (hc/slurp-response response :json-parse-mode :data :json-key-fn keyword))))
```

SEE ALSO

[http-client-j8/process-server-side-events](#)

Processes server side events (SSE) and calls for every event the handler 'handler'.

[http-client-j8/send](#)

Send a request

[section](#)

[top](#)

http-client-j8/status-client-range?

```
(status-client-range? status)
```

Returns true if the passed HTTP status code is in the range of the CLIENT codes (400 ... 499) else false.

```
(http-client-j8/status-client-range? 400)  
=> true
```

SEE ALSO

[http-client-j8/status-ok-range?](#)

Returns true if the passed HTTP status code is in the range of the OK codes (200 ... 299) else false.

[http-client-j8/status-redirect-range?](#)

Returns true if the passed HTTP status code is in the range of the REDIRECT codes (300 ... 399) else false.

[http-client-j8/status-server-error-range?](#)

Returns true if the passed HTTP status code is in the range of the SERVER ERROR codes (500 ... 599) else false.

[section](#)

[top](#)

http-client-j8/status-ok-range?

```
(status-ok-range? status)
```

Returns true if the passed HTTP status code is in the range of the OK codes (200 ... 299) else false.

```
(http-client-j8/status-ok-range? 200)  
=> true
```

SEE ALSO

[http-client-j8/status-redirect-range?](#)

Returns true if the passed HTTP status code is in the range of the REDIRECT codes (300 ... 399) else false.

[http-client-j8/status-client-range?](#)

Returns true if the passed HTTP status code is in the range of the CLIENT codes (400 ... 499) else false.

[http-client-j8/status-server-error-range?](#)

Returns true if the passed HTTP status code is in the range of the SERVER ERROR codes (500 ... 599) else false.

[section](#)

[top](#)

http-client-j8/status-redirect-range?

```
(status-redirect-range? status)
```

Returns true if the passed HTTP status code is in the range of the REDIRECT codes (300 ... 399) else false.

```
(http-client-j8/status-redirect-range? 300)  
=> true
```

SEE ALSO

[http-client-j8/status-ok-range?](#)

Returns true if the passed HTTP status code is in the range of the OK codes (200 ... 299) else false.

[http-client-j8/status-client-range?](#)

Returns true if the passed HTTP status code is in the range of the CLIENT codes (400 ... 499) else false.

[http-client-j8/status-server-error-range?](#)

Returns true if the passed HTTP status code is in the range of the SERVER ERROR codes (500 ... 599) else false.

[section](#)

[top](#)

http-client-j8/status-server-error-range?

```
(status-server-error-range? status)
```

Returns true if the passed HTTP status code is in the range of the SERVER ERROR codes (500 ... 599) else false.

```
(http-client-j8/status-server-error-range? 500)  
=> true
```

SEE ALSO

[http-client-j8/status-ok-range?](#)

Returns true if the passed HTTP status code is in the range of the OK codes (200 ... 299) else false.

[http-client-j8/status-redirect-range?](#)

Returns true if the passed HTTP status code is in the range of the REDIRECT codes (300 ... 399) else false.

[http-client-j8/status-client-range?](#)

Returns true if the passed HTTP status code is in the range of the CLIENT codes (400 ... 499) else false.

[section](#)

[top](#)

http-client-j8/upload-file

```
(upload-file file uri & options)
```

Upload a file.

Sets an implicit "Content-Type" header that is derived from the file's mimetype.

Request Options:

:url-query-params A map of url query parameters (name/value). The names and values will be appropriately encoded.

:url-fragment	An url fragment. The fragment will be appropriately encoded.
:headers	A map of request headers. Headers can be single- or multi-value (comma separated): {"X-Header-1" "value1" "X-Header-2" "value1, value2, value3"}
:conn-timeout	An optional connection timeout in milliseconds
:read-timeout	An optional read timeout in milliseconds
:follow-redirects	Sets wether HTTP redirects (requests with response code 3xx) should be automatically followed.
:hostname-verifier	Sets the hostname verifier. An object of type <code>:javax.net.ssl.HostnameVerifier</code> . Use only for HTTPS requests
:ssl-socket-factory	Sets the SSL socket factory. An object of type <code>:javax.net.ssl.SSLSocketFactory</code> . Use only for HTTPS requests
:use-caches	A boolean indicating wether or not to allow caching. Defaults to false
:user-agent	User agent. Defaults to "Venice HTTP client (legacy)"
:debug	Debug true/false. Defaults to false. In debug mode prints the HTTP request and response data and info on the uploaded file

Returns a map with the response fields:

:http-status	The HTTP status (a long)
:content-type-mimetype	The content type's mimetype. E.g.: "text/plain"
:content-type-charset	The content type's charset. E.g.: :utf-8
:content-encoding	The content transfer encoding (a keyword), if available else nil. E.g.: :gzip or :deflate
:content-length	The content length (a long), if available else -1
:headers	A map of headers. key: header name, value: list of header values
:data-stream	The response data input stream. See <code>http-client-j8/slurp-response</code> to painlessly process responses.

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/upload-file
                  (io/file "/Users/foo/image.png")
                  "http://localhost:8080/upload"
                  :headers { "Accept" "text/plain" }
                  :debug true)
        status   (:http-status response)]
    (println "Status:" status)))
```

SEE ALSO

[http-client-j8/send](#)

Send a request

[http-client-j8/upload-multipart](#)

Upload multiple parts.

[http-client-j8/build-url](#)

Builds an url

[http-client-j8/create-authorization-bearer](#)

Create a base64 encoded bearer token for an 'Authorization' HTTP header.

section

top

http-client-j8/upload-multipart

(upload-multipart parts uri & options)

Upload multiple parts.

The upload supports string parts, file parts, and generic parts. Any number of parts can be uploaded.

Sets the "Content-Type" header to "multipart/form-data".

The parts are passed as a map of part data:

```
{ ;; a string part
  "Part-1" "xxxxxxxxxxxx"

  ;; a file part
  "Part-2" (io/file "/Users/juerg/Desktop/image.png")

  ;; a x-www-form-urlencoded (generic) part
  "Part-3" { :mimetype "application/x-www-form-urlencoded"
             :charset   :utf-8
             :data       "color=blue" }

  ;; a generic part
  ;; The charset of a generic part is only required for text based
  ;; data. When passing binary data the charset can be left out.
  "Part-4" { :filename  "data.xml"
             :mimetype  "application/xml"
             :charset   :utf-8
             :data       "<user><name>foo</name></user>" }}}}
```

Request Options:

:url-query-params	A map of url query parameters (name/value). The names and values will be appropriately encoded.
:url-fragment	An url fragment. The fragment will be appropriately encoded.
:headers	A map of request headers. Headers can be single- or multi-value (comma separated): { "X-Header-1" "value1" "X-Header-2" "value1, value2, value3" }
:conn-timeout	An optional connection timeout in milliseconds
:read-timeout	An optional read timeout in milliseconds
:follow-redirects	Sets wether HTTP redirects (requests with response code 3xx) should be automatically followed.
:hostname-verifier	Sets the hostname verifier. An object of type <code>:javax.net.ssl.HostnameVerifier</code> . Use only for HTTPS requests
:ssl-socket-factory	Sets the SSL socket factory. An object of type <code>:javax.net.ssl.SSLSocketFactory</code> . Use only for HTTPS requests
:use-caches	A boolean indicating wether or not to allow caching. Defaults to false
:user-agent	User agent. Defaults to "Venice HTTP client (legacy)"
:debug	Debug true/false. Defaults to false. In debug mode prints the HTTP request (multipart data included) and the response data

Returns a map with the response fields:

:http-status	The HTTP status (a long)
:content-type-mimetype	The content type's mimetype. E.g.: "text/plain"
:content-type-charset	The content type's charset. E.g.: :utf-8
:content-encoding	The content transfer encoding (a keyword), if available else nil. E.g.: :gzip or :deflate
:content-length	The content length (a long), if available else -1
:headers	A map of headers. key: header name, value: list of header values
:data-stream	The response data input stream. See <code>http-client-j8/slurp-response</code> to painlessly process responses.

```
(do
  (load-module :http-client-j8 ['http-client-j8 :as 'hc])

  (let [response (hc/upload-multipart
    { "image1" (io/file "/Users/foo/image1.png")
      "image2" (io/file "/Users/foo/image2.png") }
    "http://localhost:8080/upload"
    :headers { "Accept" "text/plain" }
    :debug true)
    status (:http-status response)]
    (println "Status:" status)))
```

SEE ALSO

[http-client-j8/upload-file](#)

Upload a file.

[http-client-j8/upload-multipart](#)

Upload multiple parts.

[http-client-j8/build-url](#)

Builds an url

[http-client-j8/create-authorization-bearer](#)

Create a base64 encoded bearer token for an 'Authorization' HTTP header.

[section](#)

[top](#)

identity

```
(identity x)
```

Returns its argument.

```
(identity 4)
```

```
=> 4
```

```
(filter identity [1 2 3 nil 4 false true 1234])
```

```
=> (1 2 3 4 true 1234)
```

[section](#)

[top](#)

if

```
(if test then else)
```

```
(if test then)
```

Evaluates test. If logical true, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

```
(if (< 10 20) "yes" "no")
```

```
=> "yes"
```

```
(if true "yes")
```

```
=> "yes"
```

```
(if false "yes")  
=> nil
```

SEE ALSO

[if-let](#)

bindings is a vector with 2 elements: binding-form test.

[if-not](#)

Evaluates test. If logical false, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

[when](#)

Evaluates test. If logical true, evaluates body in an implicit do.

[when-not](#)

Evaluates test. If logical false, evaluates body in an implicit do.

[when-let](#)

bindings is a vector with 2 elements: binding-form test.

[section](#)[top](#)

if-let

```
(if-let bindings then)  
(if-let bindings then else)
```

bindings is a vector with 2 elements: binding-form test.
If test is true, evaluates then with binding-form bound to the value of test, if not, yields else

```
(if-let [value (* 100 2)]  
  (str "The expression is true. value=" value)  
  (str "The expression is false."))  
=> "The expression is true. value=200"
```

SEE ALSO

[when-let](#)

bindings is a vector with 2 elements: binding-form test.

[let](#)

Evaluates the expressions and binds the values to symbols in the new local context.

[section](#)[top](#)

if-not

```
(if-not test then else)  
(if-not test then)
```

Evaluates test. If logical false, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

```
(if-not (== 1 2) 100 0)  
=> 100
```

```
(if-not false 100)  
=> 100
```



```
(if-not true 100)
=> nil
```

SEE ALSO

if

Evaluates test. If logical true, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

if-let

bindings is a vector with 2 elements: binding-form test.

when

Evaluates test. If logical true, evaluates body in an implicit do.

when-not

Evaluates test. If logical false, evaluates body in an implicit do.

when-let

bindings is a vector with 2 elements: binding-form test.

[section](#)[top](#)

images/alpha-channel?

```
(alpha-channel? img)
```

Returns `true` if the image supports an alpha channel else `false`. 'img' is a `:java.awt.Image`.

```
(do
  (load-module :images)
  (->> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/alpha-channel?)))
```

[section](#)[top](#)

images/anti-alias

```
(anti-alias g2d on)
```

Turns anti-alias on/off on the Graphics2D context

`(images/anti-alias on)` is a short form of:

```
(let [key (. :RenderingHints :KEY_ANTIALIASING)
      val (. :RenderingHints :VALUE_ANTIALIAS_ON)]
  (. g2d :setRenderingHint key val))

(let [key (. :RenderingHints :KEY_TEXT_ANTIALIASING)
      val (. :RenderingHints :VALUE_TEXT_ANTIALIAS_ON)]
  (. g2d :setRenderingHint key val))
```

See: [Rendering Quality](#)

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
```

```

(images/anti-alias g true)    ;; enable anti-alias
(images/fg-color g images/blue)
(images/fill-round-rect g 80 60 100 50 12 12)

(images/anti-alias g false)  ;; disable anti-alias
(images/fg-color g images/red)
(images/fill-round-rect g 220 60 100 50 12 12)

(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

```

SEE ALSO

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip](#)

Returns the current clip shape.

[images/get-clip-bounds](#)

Returns the current clip bounds as a `:java.awt.Rectangle`.

[section](#)
[top](#)

images/apply-ops

```
(apply-ops img ops)
```

Applies one or multiple image operators (`:java.awt.image.BufferedImageOp`) to the image. Returns a new image.

Examples for image operators:

```

(import :java.awt.color.ColorSpace
        :java.awt.image.ColorConvertOp
        :java.awt.image.ConvolveOp
        :java.awt.image.Kernel
        :java.awt.image.RescaleOp)

(def convolve-op-antialias (. :ConvolveOp
                             :new
                             (. :Kernel :new 3 3
                                [ 0.00F, 0.08F, 0.00F,
                                  0.08F, 0.68F, 0.08F,
                                  0.00F, 0.08F, 0.00F ]))
                             (. :ConvolveOp :EDGE_NO_OP)
                             nil))

```

```

(def rescale-op-darker (. :RescaleOp :new 0.9F 0 nil))

(def rescale-op-brighter (. :RescaleOp :new 1.1F 0 nil))

(def color-convert-op-grayscale (as-> (. :ColorSpace :CS_GRAY) cs
                                       (. :ColorSpace :getInstance cs)
                                       (. :ColorConvertOp :new cs nil)))

;; make the image brighter
(do
  (load-module :images)
  (import :java.awt.image.RescaleOp)

  (let [op-brighter (. :RescaleOp :new 1.3F 0 nil)]
    (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
        (images/apply-ops [op-brighter])
        (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg")))))

;; convert the image to grayscale
(do
  (load-module :images)
  (import :java.awt.color.ColorSpace)
  (import :java.awt.image.ColorConvertOp)

  (let [op-grayscale (as-> (. :ColorSpace :CS_GRAY) cs
                          (. :ColorSpace :getInstance cs)
                          (. :ColorConvertOp :new cs nil))]
    (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
        (images/apply-ops [op-grayscale])
        (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg")))))

```

[section](#)
[top](#)

images/bg-color

```
(bg-color g2d color)
```

Sets the background color on the graphics context.

```

(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/bg-color g images/blue)
    (images/clear-rect g 150 80 100 50)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip](#)

Returns the current clip shape.

[images/get-clip-bounds](#)

Returns the current clip bounds as a :java.awt.Rectangle.

[section](#)

[top](#)

images/clear-rect

```
(clear-rect g2d x y width height)
```

Clears the specified rectangle by filling it with the current background color.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/bg-color g images/blue)
    (images/clear-rect g 150 80 100 50)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/copy-area](#)

Copies an area of the component by a distance specified by dx and dy.

[section](#)

[top](#)

images/convert-to-rgb

```
(convert-to-rgb img)
```

Convert an image to RGB.

Returns a new image.

```
(do
  (load-module :images)
  (let [src (io/file "/Users/foo/Desktop/Pink-Forest.jpg")
        dst (io/file "/Users/foo/Desktop/Pink-Forest.png")]
    (-> (images/load src)
        (images/convert-to-rgba)
        (images/save "png" dst))))
```

SEE ALSO

[images/resize-fit](#)

Resizes an image to a new size, a square of width and height the image should fit within the size.

[section](#)

[top](#)

images/convert-to-rgba

```
(convert-to-rgba img)
```

Convert an image to RGBA.

Returns a new image.

```
(do
  (load-module :images)
  (let [src (io/file "/Users/foo/Desktop/Pink-Forest.jpg")
        dst (io/file "/Users/foo/Desktop/Pink-Forest.png")]
    (-> (images/load src)
        (images/convert-to-rgba)
        (images/save "png" dst))))
```

SEE ALSO

[images/resize-fit](#)

Resizes an image to a new size, a square of width and height the image should fit within the size.

[section](#)

[top](#)

images/copy

```
(copy img)
```

Creates a copy of `:java.awt.image.BufferedImage`.

```
(do
  (load-module :images)
  (-> (images/create 400 200 images/TYPE_INT_ARGB)
      (images/copy)
      (images/save :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/load](#)

Loads an image from a `:java.io.File`, a `:java.io.InputStream`, or a `:java.net.URL`.

[images/save](#)

Saves an image to `'java.io.File'` or an `'java.io.OutputStream'`.

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[section](#)[top](#)

images/copy-area

`(copy-area g2d x y with height dx dy)`

Copies an area of the component by a distance specified by `dx` and `dy`.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-circle g 100 100 100)
    (images/copy-area g 50 50 100 100 200 30)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png")))))
```

SEE ALSO

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/clear-rect](#)

Clears the specified rectangle by filling it with the current background color.

[section](#)[top](#)

images/create

`(create width height type)`

`(create width height type color)`

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

Without color, creates a transparent image (type 'TYPE_INT_ARGB') or black image (type 'TYPE_INT_RGB').

```
(do
  (load-module :images)
  (-> (images/create 400 200 images/TYPE_INT_ARGB)
      (images/save :png (io/file "/Users/foo/Desktop/test.png")))))

(do
  (load-module :images)
  (-> (images/create 400 200 images/TYPE_INT_ARGB images/white)
      (images/save :png (io/file "/Users/foo/Desktop/test.png")))))
```

SEE ALSO

[images/load](#)

Loads an image from a `:java.io.File`, a `:java.io.InputStream`, or a `:java.net.URL`.

[images/save](#)

Saves an image to `:java.io.File` or an `:java.io.OutputStream`.

[images/copy](#)

Creates a copy of `:java.awt.image.BufferedImage`.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[section](#)[top](#)

images/crop

`(crop img x y width height)`

Crops an image. Returns a new image.

```
(do
  (load-module :images)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/crop 1000 1000 400 200)
    (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg"))))
```

[section](#)[top](#)

images/dimension

`(dimension f)`

Returns the images dimensions as a vector `[width height]`. 'f' may be a `:java.io.File` or a `:java.awt.Image`.

Note: Do not load a file first to get the dimension, passing a `:java.io.File` is much faster!

```
(do
  (load-module :images)
  (images/dimension (io/file "/Users/foo/Desktop/Pink-Forest.jpg")))
```

[section](#)[top](#)

images/draw-circle

`(draw-circle g2d x y radius)`

Draws a circle.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
```

```

      g (images/g2d img)]
      (images/fg-color g images/blue)
      (images/draw-circle g 200 100 100)
      (images/dispose g)
      (images/save img :png (io/file "/Users/foo/Desktop/test.png")))))

;; create a mask
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g (images/g2d img)]
    (. g :setComposite (. :java.awt.AlphaComposite :Clear))
    (images/stroke g 10.0)
    (images/draw-circle g 200 100 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png")))))

```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/draw-oval](#)

Draws an oval.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-line](#)

Draws a line.

[images/draw-image](#)

Draws an image to the position x,y in the graphics context.

[section](#)

[top](#)

images/draw-image

```

(draw-image g2d x y)
(draw-image g2d x y width height)

```

Draws an image to the position x,y in the graphics context.


```
(do
  (load-module :images)

  (let [img1 (images/create 50 50 images/TYPE_INT_ARGB images/blue)
        img2 (images/create 50 50 images/TYPE_INT_ARGB images/magenta)
        img  (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g     (images/g2d img)]

    (images/draw-image g img1 100 30)
    (images/draw-image g img2 130 60)
    (images/draw-image g img1 160 90)

    (images/draw-image g img1 250 30 30 30)
    (images/draw-image g img2 280 60 30 30)
    (images/draw-image g img1 310 90 30 30)

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/draw-circle](#)

Draws a circle.

[images/draw-oval](#)

Draws an oval.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-line](#)

Draws a line.

[section](#)[top](#)

images/draw-line

```
(draw-line g2d x1 y1 x2 y2)
```

Draws a line.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
```

```
g (images/g2d img)]
(images/anti-alias g true) ;; enable anti-alias
(images/stroke g 10.0 images/cap-round images/join-miter)
(images/fg-color g images/blue)
(images/draw-line g 50 50 350 150)
(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/draw-circle](#)

Draws a circle.

[images/draw-oval](#)

Draws an oval.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-image](#)

Draws an image to the position `x,y` in the graphics context.

[section](#)

[top](#)

images/draw-oval

```
(draw-oval g2d center-x center-y width height)
```

Draws an oval.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g (images/g2d img)]
    (images/stroke g 10.0)
    (images/fg-color g images/blue)
    (images/draw-oval g 200 100 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/draw-circle](#)

Draws a circle.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-line](#)

Draws a line.

[images/draw-image](#)

Draws an image to the position `x,y` in the graphics context.

[section](#)

[top](#)

images/draw-polygon

`(draw-polygon g2d polygon)`

Draws a polygon.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
    (images/stroke g 5.0 images/cap-round images/join-miter)

    ;; hexagon
    (->> (images/hexagon-poly)
          (images/scale-points 80)
          (images/translate-points 200 100)
          (images/polygon)
          (images/draw-polygon g))

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
```

```

      g (images/g2d img)]
(images/fg-color g images/blue)
(images/anti-alias g true) ;; enable anti-alias

;; draw an arc of hexagons
(let [anchor-x 200, anchor-y 170]
  (doseq [angle (range 0 190 10)]
    (->> (images/hexagon-poly)
      (images/scale-points 20)
      (images/rotate-points 90)
      (images/translate-points 50 170)
      (images/rotate-points angle anchor-x anchor-y)
      (images/polygon)
      (images/draw-polygon g))))

(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/draw-circle](#)

Draws a circle.

[images/draw-oval](#)

Draws an oval.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-line](#)

Draws a line.

[images/draw-image](#)

Draws an image to the position `x,y` in the graphics context.

[section](#)

[top](#)

images/draw-rect

```

(draw-rect g2d x y width height)
(draw-rect g2d x y width height color)

```

Draws a rectangle.

```

(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)

```

```
      g      (images/g2d img)]
(images/stroke g 10.0)
(images/fg-color g images/blue)
(images/draw-rect g 150 70 100 50)
(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/draw-circle](#)

Draws a circle.

[images/draw-oval](#)

Draws an oval.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-line](#)

Draws a line.

[images/draw-image](#)

Draws an image to the position `x,y` in the graphics context.

[section](#)

[top](#)

images/draw-round-rect

(draw-round-rect g2d x y width height arc-width arc-height)

Draws a rounded rectangle.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]

    (images/stroke g 10.0)

    (images/fg-color g images/blue)
    (images/anti-alias g true)      ;; enable anti-alias
    (images/draw-round-rect g 80 60 100 50 12 12)

    (images/fg-color g images/red)
    (images/anti-alias g false)     ;; disable anti-alias
```

```
(images/draw-round-rect g 220 60 100 50 12 12)

(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/draw-circle](#)

Draws a circle.

[images/draw-oval](#)

Draws an oval.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-string](#)

Draws text with an optional size at the given position.

[images/draw-line](#)

Draws a line.

[images/draw-image](#)

Draws an image to the position x,y in the graphics context.

[section](#)

[top](#)

images/draw-string

```
(draw-string g2d text x y)
(draw-string g2d text x y size)
```

Draws text with an optional size at the given position.

```
(do
  (load-module :images)
  (import :java.awt.Font)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g (images/g2d img)]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
    (images/draw-string g "Hello, world!" 50 120 50)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/draw-circle](#)

Draws a circle.

[images/draw-oval](#)

Draws an oval.

[images/draw-rect](#)

Draws a rectangle.

[images/draw-round-rect](#)

Draws a rounded rectangle.

[images/draw-polygon](#)

Draws a polygon.

[images/draw-line](#)

Draws a line.

[images/draw-image](#)

Draws an image to the position `x,y` in the graphics context.

[section](#)

[top](#)

images/fg-color

```
(fg-color g2d color)
```

Sets the foreground color on the graphics context.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-oval g 200 100 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip](#)

Returns the current clip shape.

[images/get-clip-bounds](#)

Returns the current clip bounds as a :java.awt.Rectangle.

[section](#)

[top](#)

images/fill-circle

```
(fill-circle g2d x y radius)
```

Draws a filled circle.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-circle g 200 100 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

;; create a mask (fill circle with transparent pixels)
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (. g :setComposite (. :java.awt.AlphaComposite :Clear))
    (images/fill-circle g 200 100 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/fill-oval](#)

Draws a filled oval.

[images/fill-rect](#)

Draws a filled rectangle.

[images/fill-round-rect](#)

Draws a filled rounded rectangle.

images/fill-polygon

Draws a filled polygon.

[section](#)[top](#)

images/fill-oval

```
(fill-oval g2d center-x center-y width height)
```

Draws a filled oval.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-oval g 200 100 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/fill-circle](#)

Draws a filled circle.

[images/fill-rect](#)

Draws a filled rectangle.

[images/fill-round-rect](#)

Draws a filled rounded rectangle.

[images/fill-polygon](#)

Draws a filled polygon.

[section](#)[top](#)

images/fill-polygon

```
(fill-polygon g2d polygon)
```

Draws a filled polygon.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
```

```
;; hexagon
(->> (images/hexagon-poly)
      (images/scale-points 80)
      (images/translate-points 200 100)
      (images/polygon)
      (images/fill-polygon g))

(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/fill-circle](#)

Draws a filled circle.

[images/fill-oval](#)

Draws a filled oval.

[images/fill-rect](#)

Draws a filled rectangle.

[images/fill-round-rect](#)

Draws a filled rounded rectangle.

[section](#)[top](#)

images/fill-rect

```
(fill-rect g2d x y width height)
```

Draws a filled rectangle.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-rect g 150 80 100 50)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/fill-circle](#)

Draws a filled circle.

[images/fill-oval](#)

Draws a filled oval.

[images/fill-round-rect](#)

Draws a filled rounded rectangle.

[images/fill-polygon](#)

Draws a filled polygon.

[section](#)[top](#)

images/fill-round-rect

(fill-round-rect g2d x y width height arc-width arc-height)

Draws a filled rounded rectangle.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
    (images/fill-round-rect g 80 60 100 50 12 12)

    (images/fg-color g images/red)
    (images/anti-alias g false) ;; disable anti-alias
    (images/fill-round-rect g 220 60 100 50 12 12)

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/fill-circle](#)

Draws a filled circle.

[images/fill-oval](#)

Draws a filled oval.

[images/fill-rect](#)

Draws a filled rectangle.

[images/fill-polygon](#)

Draws a filled polygon.

[section](#)[top](#)

images/flip

```
(flip img mode)
```

Flips an image vertically or horizontally. Returns a new image.

Mode is either :vertical or :horizontal.

```
(do
  (load-module :images)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/flip :vertical)
    (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg")))))
```

[section](#)[top](#)

images/format-names

```
(format-names)
```

Returns a list of format that the image writer supports.

```
(do
  (load-module :images)
  (images/format-names))
=> ["JPG" "jpg" "bmp" "BMP" "gif" "GIF" "WBMP" "png" "PNG" "jpeg" "wbmp" "JPEG"]
```

[section](#)[top](#)

images/g2d

```
(g2d img)
```

Creates a Graphics2D context from an image.

See: [Rendering Quality](#)

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/stroke g 10.0)
    (images/fg-color g images/blue)
    (images/fill-oval g 200 100 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png")))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip](#)

Returns the current clip shape.

[images/get-clip-bounds](#)

Returns the current clip bounds as a :java.awt.Rectangle.

[section](#)

[top](#)

images/get-clip

```
(get-clip g2d)
```

Returns the current clip shape.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)
        old (images/get-clip g)]
    (images/anti-alias g true) ;; enable anti-alias
    (images/fg-color g images/blue)
    (images/set-clip g 0 0 200 200) ;; left half of the img
    (images/fill-oval g 200 100 300 150)
    (images/set-clip g old) ;; restore previous clipping
    (images/stroke g 10.0 images/cap-round images/join-miter)
    (images/fg-color g images/red)
    (images/draw-line g 50 50 350 150)

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip-bounds](#)

Returns the current clip bounds as a :java.awt.Rectangle.

[section](#)[top](#)

images/get-clip-bounds

```
(get-clip-bounds g2d)
```

Returns the current clip bounds as a :java.awt.Rectangle.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/get-clip-bounds g)))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip](#)

Returns the current clip shape.

[section](#)[top](#)

images/get-transform

```
(get-transform g2d)
```

Returns a copy of the current Transform in the Graphics2D context.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-oval g 200 100 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/set-transform](#)

Overwrites the Transform in the Graphics2D context. 'tx' may be nil.

[images/transform](#)

Add an affine transformer to the Graphics2D context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-translate](#)

Create a translate transformer.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

images/hexagon-poly

(hexagon-poly)

Creates a normalized (center at (0,0), radius 1) hexagon polygon point list.

```
(do
  (load-module :images)
  (images/hexagon-poly))
```

[section](#)

[top](#)

images/load

(load file)

Loads an image from a `:java.io.File`, a `:java.io.InputStream`, or a `:java.net.URL`.

```
(do
  (load-module :images)
  (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg")))
```

SEE ALSO

[images/save](#)

Saves an image to 'java.io.File' or an 'java.io.OutputStream'.

[images/create](#)

Creates a new `:java.awt.image.BufferedImage` with the given width, height, and type.

[images/copy](#)

Creates a copy of `:java.awt.image.BufferedImage`.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[section](#)

[top](#)

images/pad

```
(pad img padding color)
(pad img pad-top pad-right pad-bottom pad-left color)
```

Pads an image. Returns a new image.

```
(do
  (load-module :images)
  (import :java.awt.Color)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/pad 200 images/white)
    (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg"))))
```

[section](#)

[top](#)

images/point

```
(point x y)
```

Creates a `:java.awt.Point`

```
(do
  (load-module :images)
  (images/point 0 0))
```

[section](#)

[top](#)

images/polygon

```
(polygon points)
```


Creates a `:java.awt.Polygon`

```
(do
  (load-module :images)
  (images/polygon [[0 0] [0 100] [100 100] [100 0] [0 0]]))
```

[section](#)

[top](#)

images/rectangle

(rectangle x y width height)

Creates a `:java.awt.Rectangle`

```
(do
  (load-module :images)
  (images/rectangle 0 0 200 100))
```

[section](#)

[top](#)

images/rectangle-poly

(rectangle-poly x y width height)

Creates a rectangle polygon point list.

```
(do
  (load-module :images)
  (images/rectangle-poly 0 0 200 100))
```

[section](#)

[top](#)

images/resize

(resize img width height)
(resize img width height resize-mode)

Resizes an image to a new width and height.

Resize modes:

<code>:resize-auto</code>	(default) chooses best mode
<code>:resize-performance</code>	resize for best performance
<code>:resize-balanced</code>	balance between performance and quality
<code>:resize-quality</code>	resize for quality
<code>:resize-high-quality</code>	resize for high quality

Returns a new image.

```
(do
  (load-module :images)
  (let [src (io/file "/Users/foo/Desktop/Pink-Forest.jpg")
```

```
dst (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg")]
(-> (images/load src)
    (images/resize 500 200 :resize-balanced)
    (images/save "jpg" dst))

(println (io/file-name src) ":" (images/dimension src))
(println (io/file-name dst) ":" (images/dimension dst)))
```

SEE ALSO

[images/resize-fit](#)

Resizes an image to a new size, a square of width and height the image should fit within the size.

[section](#)

[top](#)

images/resize-fit

```
(resize-fit img size fit-mode)
(resize img size fit-mode resize-mode)
```

Resizes an image to a new size, a square of width and height the image should fit within the size.

Resize modes:

<code>:fit-best</code>	(default), fit within a square box of size 'size', keeps width / height ratio
<code>:fit-exact</code>	fit exactly to a square of size 'size', causes distortions
<code>:fit-width</code>	fit to width, keeps width / height ratio
<code>:fit-height</code>	fit to height, keeps width / height ratio

Resize modes:

<code>:resize-auto</code>	(default) chooses best mode
<code>:resize-performance</code>	resize for best performance
<code>:resize-balanced</code>	balance between performance and quality
<code>:resize-quality</code>	resize for quality
<code>:resize-high-quality</code>	resize for high quality

Returns a new image.

```
(do
  (load-module :images)
  (let [src (io/file "/Users/foo/Desktop/Pink-Forest.jpg")
        dst (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg")]
    (-> (images/load src)
        (images/resize-fit 500 :fit-best :resize-balanced)
        (images/save "jpg" dst))

    (println (io/file-name src) ":" (images/dimension src))
    (println (io/file-name dst) ":" (images/dimension dst))))
```

SEE ALSO

[images/resize](#)

Resizes an image to a new width and height.

[section](#)

[top](#)

images/rotate

```
(rotate img angle)
```

Rotates an image by 0°, 90°, 180°, or 270° clockwise. Returns a new image.

```
(do
  (load-module :images)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/rotate 90)
    (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg"))))
```

[section](#)[top](#)

images/rotate-points

```
(rotate-points angle points)
(rotate-points angle anchor-x anchor-y points)
```

Rotate the points of a polygon point list with an angle in degrees [0..360] around an anchor point. The default anchor point is (0,0)

```
(do
  (load-module :images)
  (->> (images/rectangle-poly 0 0 200 100)
    (images/rotate-points 45 (square-poly 100 100 100 100))))
```

[section](#)[top](#)

images/save

```
(save img format-name f)
```

Saves an image to 'java.io.File' or an 'java.io.OutputStream'.

Supported formats: "png", "jpg", "jpeg", "gif", "bmp", ...

```
(do
  (load-module :images)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/save img :png (io/file "/Users/foo/Desktop/Pink-Forest.png"))))
```

SEE ALSO

[images/load](#)

Loads an image from a :java.io.File, a :java.io.InputStream, or a :java.net.URL.

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/copy](#)

Creates a copy of :java.awt.image.BufferedImage.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/format-names](#)

Returns a list of format that the image writer supports.

images/scale-points

```
(scale-points factor points)
(scale-points factor-x factor-y points)
```

Scales the points of a polygon point list with factor.

```
(do
  (load-module :images)
  (->> (images/rectangle-poly 0 0 200 100)
    (images/scale-points 4.0 100 50)))
```

images/set-clip

```
(set-clip g2d shape)
(set-clip g2d x y width height)
```

Sets the current clip to the rectangle specified by the given coordinates.

A shape can be `nil` to remove any clipping.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/set-clip g 0 0 200 200) ;; left half of the img
    (images/fill-oval g 200 100 300 150)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the `Graphics2D` context

[images/stroke](#)

Sets a new stroke on the graphics context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/get-clip](#)

Returns the current clip shape.

[images/get-clip-bounds](#)

Returns the current clip bounds as a :java.awt.Rectangle.

[section](#)

[top](#)

images/set-transform

```
(set-transform g2d tx)
```

Overwrites the Transform in the Graphics2D context. 'tx' may be `nil`.

Note: This function should **never** be used to apply a new coordinate transform on top of an existing transform. It should only be used to restore the original transform of the graphics.

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/fill-oval g 200 100 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/get-transform](#)

Returns a copy of the current Transform in the Graphics2D context.

[images/transform](#)

Add an affine transformer to the Graphics2D context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-translate](#)

Create a translate transformer.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

images/shear

```
(shear img shx shy)
(shear img shx shy color)
```

Shears an image vertically and/or horizontally. Returns a new image.

```
(do
  (load-module :images)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/shear 0.05 0.0)
    (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg")))))
```

[section](#)

[top](#)

images/square-poly

(square-poly x y length)

Creates a square polygon point list.

```
(do
  (load-module :images)
  (images/square-poly 0 0 100))
```

[section](#)

[top](#)

images/stroke

(stroke g2d)
(stroke g2d width)
(stroke g2d width cap join)
(stroke g2d width cap join meter-limit)
(stroke g2d width cap join meter-limit dash dash-phase)

Sets a new stroke on the graphics context

See: [BasicStroke](#)

```
(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/stroke g 10.0 images/cap-round images/join-miter)
    (images/draw-oval g 100 50 200 100)
    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

(do
  (load-module :images)

  (let [img (images/create 400 200 images/TYPE_INT_ARGB)
        g   (images/g2d img)]
    (images/fg-color g images/blue)
    (images/stroke g
      10.0
      images/cap-round
      images/join-miter
      5.0
```

```
        [20.0]
        20.0)
(images/draw-oval g 200 100 200 100)
(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new :java.awt.image.BufferedImage with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/dispose](#)

Disposes of this graphics context and releases any system resources that it is using.

[images/anti-alias](#)

Turns anti-alias on/off on the Graphics2D context

[images/fg-color](#)

Sets the foreground color on the graphics context.

[images/bg-color](#)

Sets the background color on the graphics context.

[images/set-clip](#)

Sets the current clip to the rectangle specified by the given coordinates.

[images/get-clip](#)

Returns the current clip shape.

[images/get-clip-bounds](#)

Returns the current clip bounds as a :java.awt.Rectangle.

[section](#)

[top](#)

images/transform

```
(transform g2d tx)
```

Add an affine transformer to the Graphics2D context.

Note: The last specified transform on the graphics context is applied first!

```
(do
  (load-module :images)

  (let [img      (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g        (images/g2d img)
        square [[-1 -1] [1 -1] [1 1] [-1 1]]]
    (images/fg-color g images/blue)
    (images/anti-alias g true)    ;; enable anti-alias
    (images/stroke g 5.0 images/cap-round images/join-miter)
    (images/transform g (images/tx-translate 200 100))
    (images/transform g (images/tx-scale 15))
    (images/draw-polygon g (images/polygon square))

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/set-transform](#)

Overwrites the `Transform` in the `Graphics2D` context. 'tx' may be nil.

[images/get-transform](#)

Returns a copy of the current `Transform` in the `Graphics2D` context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-translate](#)

Create a translate transformer.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

images/translate

```
(translate img x y)
(translate img x y color)
```

Translates from (0, 0) to the current (x, y) position. Returns a new image.

```
(do
  (load-module :images)
  (-> (images/load (io/file "/Users/foo/Desktop/Pink-Forest.jpg"))
    (images/translate 200 100)
    (images/save "jpg" (io/file "/Users/foo/Desktop/Pink-Forest-1.jpg"))))
```

[section](#)

[top](#)

images/translate-points

```
(translate-points dx dy points)
```

Translate the points of a polygon point list.

```
(do
  (load-module :images)
  (->> (images/rectangle-poly 0 0 200 100)
    (images/translate-points 4.0)))
```

[section](#)

[top](#)

images/tx-identity

(tx-identity)

Create identity transformer.

Note: The last specified transform on the graphics context is applied first!

```
(do
  (load-module :images)

  (let [img      (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g        (images/g2d img)
        square [[30 30] [50 30] [50 50] [30 50]]]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
    (images/stroke g 5.0 images/cap-round images/join-miter)
    (images/transform g (images/tx-identity))
    (images/draw-polygon g (images/polygon square))

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/set-transform](#)

Overwrites the `Transform` in the `Graphics2D` context. 'tx' may be nil.

[images/get-transform](#)

Returns a copy of the current `Transform` in the `Graphics2D` context.

[images/transform](#)

Add an affine transformer to the `Graphics2D` context.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

images/tx-rotate

(tx-rotate angle)
(tx-rotate angle anchor-x anchor-y)

Create a rotate transformer.

Note: The last specified transform on the graphics context is applied first!

```
(do
  (load-module :images)

  (let [img      (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g        (images/g2d img)
        square [[[-1 -1] [1 -1] [1 1] [-1 1]]]
        (images/fg-color g images/blue)
        (images/anti-alias g true)    ;; enable anti-alias
        (images/stroke g 5.0 images/cap-round images/join-miter)
        (images/transform g (images/tx-translate 200 100))
        (images/transform g (images/tx-rotate 45))
        (images/transform g (images/tx-scale 15))
        (images/draw-polygon g (images/polygon square))

        (images/dispose g)
        (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/set-transform](#)

Overwrites the `Transform` in the `Graphics2D` context. 'tx' may be nil.

[images/get-transform](#)

Returns a copy of the current `Transform` in the `Graphics2D` context.

[images/transform](#)

Add an affine transformer to the `Graphics2D` context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-translate](#)

Create a translate transformer.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

images/tx-scale

```
(tx-scale s)
(tx-scale sx sy)
```

Create a scale transformer.

Take care for strokes when scaling up/down. A stroke width will be scaled as well!

Note: The last specified transform on the graphics context is applied first!

```
(do
  (load-module :images)

  (let [img      (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g        (images/g2d img)
```

```

square [[-1 -1] [1 -1] [1 1] [-1 1]]]
(images/fg-color g images/blue)
(images/anti-alias g true) ;; enable anti-alias
(images/stroke g 5.0 images/cap-round images/join-miter)
(images/transform g (images/tx-translate 200 100))
(images/transform g (images/tx-scale 15))
(images/draw-polygon g (images/polygon square))

(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))

```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/set-transform](#)

Overwrites the Transform in the `Graphics2D` context. 'tx' may be nil.

[images/get-transform](#)

Returns a copy of the current Transform in the `Graphics2D` context.

[images/transform](#)

Add an affine transformer to the `Graphics2D` context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-translate](#)

Create a translate transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

images/tx-shear

```
(tx-shear sx sy)
```

Create a shear transformer.

Note: The last specified transform on the graphics context is applied first!

```

(do
  (load-module :images)

  (let [img      (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g        (images/g2d img)
        square [[-1 -1] [1 -1] [1 1] [-1 1]]]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
    (images/stroke g 5.0 images/cap-round images/join-miter)
    (images/transform g (images/tx-translate 200 100))
    (images/transform g (images/tx-shear -0.3 0.0))
    (images/transform g (images/tx-scale 15))
    (images/draw-polygon g (images/polygon square))
  )

```

```
(images/dispose g)
(images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a `Graphics2D` context from an image.

[images/set-transform](#)

Overwrites the `Transform` in the `Graphics2D` context. 'tx' may be nil.

[images/get-transform](#)

Returns a copy of the current `Transform` in the `Graphics2D` context.

[images/transform](#)

Add an affine transformer to the `Graphics2D` context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-translate](#)

Create a translate transformer.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[section](#)[top](#)

images/tx-translate

```
(tx-translate x y)
```

Create a translate transformer.

Note: The last specified transform on the graphics context is applied first!

```
(do
  (load-module :images)

  (let [img      (images/create 400 200 images/TYPE_INT_ARGB images/white)
        g        (images/g2d img)
        square [[-1 -1] [1 -1] [1 1] [-1 1]]]
    (images/fg-color g images/blue)
    (images/anti-alias g true) ;; enable anti-alias
    (images/stroke g 5.0 images/cap-round images/join-miter)
    (images/transform g (images/tx-translate 200 100))
    (images/transform g (images/tx-scale 15))
    (images/draw-polygon g (images/polygon square))

    (images/dispose g)
    (images/save img :png (io/file "/Users/foo/Desktop/test.png"))))
```

SEE ALSO

[images/create](#)

Creates a new `java.awt.image.BufferedImage` with the given width, height, and type.

[images/g2d](#)

Creates a Graphics2D context from an image.

[images/set-transform](#)

Overwrites the Transform in the Graphics2D context. 'tx' may be nil.

[images/get-transform](#)

Returns a copy of the current Transform in the Graphics2D context.

[images/transform](#)

Add an affine transformer to the Graphics2D context.

[images/tx-identity](#)

Create identity transformer.

[images/tx-scale](#)

Create a scale transformer.

[images/tx-rotate](#)

Create a rotate transformer.

[images/tx-shear](#)

Create a shear transformer.

[section](#)

[top](#)

import

```
(import class & classes)
(import class :as alias)
```

Imports one or multiple Java classes. Imports are bound to the current namespace.

Example

Without import

```
(. :java.lang.Math :max 2 10)
```

With import

```
(do
  (import :java.lang.Math)
  (. :Math :max 2 10))
```

Aliases

Aliases are helpful if Java classes have the same name but different packages like `java.util.Date` and `java.sql.Date` :

```
(do
  (import :java.util.Date)
  (import :java.sql.Date :as :SqlDate)

  (println (. :Date :new))
  (println (. :SqlDate :valueOf "2022-06-24")))
```

Static nested classes

Venice

```
(do
  (import :foo.OuterClass)
  (import :foo.OuterClass$NestedStaticClass)

  (-> (. :OuterClass :new)
    (. :message))
  (-> (. :OuterClass$NestedStaticClass :new)
    (. :message)))
```

Java

```

package foo;

public class OuterClass {
    public String message() {
        return "OuterClass::message()";
    }

    public static class NestedStaticClass {
        public String message() {
            return "NestedStaticClass::message()";
        }
    }
}

```

```

(do
  (import :java.lang.Math)
  (. :Math :max 2 10))
=> 10

```

```

(do
  (import :java.awt.Point
          :java.lang.Math)
  (. :Math :max 2 10))
=> 10

```

```

(do
  (import :java.awt.Color :as :AwtColor)
  (. :AwtColor :new 200I 230I 255I 180I))
=> java.awt.Color[r=200,g=230,b=255]

```

```

(do
  (ns util)
  (defn import? [clazz ns_]
    (any? #(== % clazz) (map first (imports ns_))))

  (ns alpha)
  (import :java.lang.Math)
  (println "alpha:" (util/import? :java.lang.Math 'alpha))

  (ns beta)
  (println "beta:" (util/import? :java.lang.Math 'beta))

  (ns alpha)
  (println "alpha:" (util/import? :java.lang.Math 'alpha))
)
alpha: true
beta: false
alpha: true
=> nil

```

SEE ALSO

[imports](#)

Without namespace arg returns a list with the registered imports for the current namespace. With namespace arg returns a list with ...

[section](#)
[top](#)

imports

```

(imports & options)
(imports ns & options)

```

Without namespace arg returns a list with the registered imports for the current namespace. With namespace arg returns a list with the registered imports for the given namespace.

Options:

:print print the import list to the current value of `*out*`

```
(do
  (import :java.lang.Math)
  (imports))
=> ([:com.github.jlangch.venice.AssertionException :AssertionException] [:com.github.jlangch.venice.
SecurityException :SecurityException] [:com.github.jlangch.venice.ValueException :ValueException] [:com.github.
jlangch.venice.VncException :VncException] [:java.lang.Exception :Exception] [:java.lang.
IllegalArgumentException :IllegalArgumentException] [:java.lang.Math :Math] [:java.lang.NullPointerException :
NullPointerException] [:java.lang.RuntimeException :RuntimeException] [:java.lang.Throwable :Throwable])

(do
  (import :java.lang.Math)
  (imports :print))
:com.github.jlangch.venice.AssertionException :as :AssertionException
:com.github.jlangch.venice.SecurityException :as :SecurityException
:com.github.jlangch.venice.ValueException :as :ValueException
:com.github.jlangch.venice.VncException :as :VncException
:java.lang.Exception :as :Exception
:java.lang.IllegalArgumentException :as :IllegalArgumentException
:java.lang.Math :as :Math
:java.lang.NullPointerException :as :NullPointerException
:java.lang.RuntimeException :as :RuntimeException
:java.lang.Throwable :as :Throwable
=> nil

(do
  (ns foo)
  (import :java.lang.Math)
  (ns bar)
  (imports 'foo))
=> ([:com.github.jlangch.venice.AssertionException :AssertionException] [:com.github.jlangch.venice.
SecurityException :SecurityException] [:com.github.jlangch.venice.ValueException :ValueException] [:com.github.
jlangch.venice.VncException :VncException] [:java.lang.Exception :Exception] [:java.lang.
IllegalArgumentException :IllegalArgumentException] [:java.lang.Math :Math] [:java.lang.NullPointerException :
NullPointerException] [:java.lang.RuntimeException :RuntimeException] [:java.lang.Throwable :Throwable])
```

SEE ALSO

[import](#)

Imports one or multiple Java classes. Imports are bound to the current namespace.

[section](#)

[top](#)

inc

```
(inc x)
```

Increments the number x

```
(inc 10)
=> 11
```

```
(inc 10I)
=> 11I

(inc 10.1)
=> 11.1

(inc 10.12M)
=> 11.12M
```

SEE ALSO

[dec](#)
Decrements the number x

[section](#) [top](#)

index-of

```
(index-of sequence val)
(index-of comparefn sequence val)
```

Returns the first index of the sequence value that is equal to val or -1 if not found.

If no compare function is supplied, uses the natural compare to compare the sequence values to the supplied value. The compare function takes two arguments and returns -1, 0, or 1.

```
(index-of [1 2 2 3] 2)
=> 1

(index-of [1 2 3] 6)
=> -1

(index-of [1 2 3] nil)
=> -1

(index-of [1 nil 3] nil)
=> 1

(index-of nil 7)
=> -1

(index-of compare [1 2 2 3] 2)
=> 1

(index-of #(if (== %1 %2) 0 1) [1 2 2 3] 2)
=> 1
```

SEE ALSO

[last-index-of](#)
Returns the last index of the sequence value that is equal to val or -1 if not found.

[section](#) [top](#)

inet/inet-addr


```
(inet/inet-addr addr)
```

Converts a stringified IPv4 or IPv6 to a Java InetAddress.

```
(inet/inet-addr "222.192.0.0")  
=> /222.192.0.0
```

```
(inet/inet-addr "2001:0db8:85a3:08d3:1319:8a2e:0370:7347")  
=> /2001:db8:85a3:8d3:1319:8a2e:370:7347
```

[section](#)

[top](#)

inet/inet-addr-from-bytes

```
(inet/inet-addr-bytes addr)
```

Converts a IPv4 or IPv6 byte address (a vector of unsigned integers) to a Java InetAddress.

```
(inet/inet-addr-from-bytes [222I 192I 12I 0I])  
=> /222.192.12.0
```

```
(inet/inet-addr-from-bytes [32I 1I 13I 184I 133I 163I 8I 211I 19I 25I 138I 46I 3I 112I 115I 71I])  
=> /2001:db8:85a3:8d3:1319:8a2e:370:7347
```

[section](#)

[top](#)

inet/inet-addr-to-bytes

```
(inet/inet-addr-to-bytes addr)
```

Converts a stringified IPv4/IPv6 address or a Java InetAddress to an InetAddress byte vector.

```
(inet/inet-addr-to-bytes "222.192.12.0")  
=> [222I 192I 12I 0I]
```

```
(inet/inet-addr-to-bytes "2001:0db8:85a3:08d3:1319:8a2e:0370:7347")  
=> [32I 1I 13I 184I 133I 163I 8I 211I 19I 25I 138I 46I 3I 112I 115I 71I]
```

```
(inet/inet-addr-to-bytes (inet/inet-addr "222.192.0.0"))  
=> [222I 192I 0I 0I]
```

[section](#)

[top](#)

inet/ip4?

```
(inet/ip4? addr)
```

Returns true if addr is an IPv4 address.

```
(inet/ip4? "222.192.0.0")  
=> true
```

```
(inet/ip4? (inet/inet-addr "222.192.0.0"))  
=> true
```

[section](#) [top](#)

inet/ip6?

```
(inet/ip6? addr)
```

Returns true if addr is an IPv6 address.

```
(inet/ip6? "2001:0db8:85a3:08d3:1319:8a2e:0370:7347")  
=> true
```

```
(inet/ip6? (inet/inet-addr "2001:0db8:85a3:08d3:1319:8a2e:0370:7347"))  
=> true
```

[section](#) [top](#)

inet/linklocal-addr?

```
(inet/linklocal-addr? addr)
```

Returns true if addr is a link local address.

```
(inet/linklocal-addr? "169.254.0.0")  
=> true
```

```
(inet/linklocal-addr? (inet/inet-addr "169.254.0.0"))  
=> true
```

[section](#) [top](#)

inet/multicast-addr?

```
(inet/multicast-addr? addr)
```

Returns true if addr is a multicast address.

```
(inet/multicast-addr? "224.0.0.1")  
=> true
```

```
(inet/multicast-addr? (inet/inet-addr "224.0.0.1"))  
=> true
```

[section](#) [top](#)

inet/reachable?

```
(inet/reachable? addr timeout)
```

Test whether that address is reachable. Best effort is made by the implementation to try to reach the host, but firewalls and server configuration may block requests resulting in a unreachable status while some specific ports may be accessible. A typical implementation will use ICMP ECHO REQUESTs if the privilege can be obtained, otherwise it will try to establish a TCP connection on port 7 (Echo) of the destination host.

The timeout value, in milliseconds, indicates the maximum amount of time the try should take. If the operation times out before getting an answer, the host is deemed unreachable.

```
(inet/reachable? "google.com" 500)
=> false
```

```
(inet/reachable? "74.125.193.113" 500)
=> false
```

[section](#)[top](#)

inet/sitelocal-addr?

```
(inet/sitelocal-addr? addr)
```

Returns true if addr is a site local address.

```
(inet/sitelocal-addr? "192.168.0.0")
=> true
```

```
(inet/sitelocal-addr? (inet/inet-addr "192.168.0.0"))
=> true
```

[section](#)[top](#)

infinite?

```
(infinite? x)
```

Returns true if x is infinite else false. x must be a double!

```
(infinite? 1.0E300)
=> false
```

```
(infinite? (* 1.0E300 1.0E100))
=> true
```

```
(infinite? (/ 1.0 0))
=> true
```

```
(pr (/ 4.1 0))
:Infinite
=> nil
```

SEE ALSO

[nan?](#)

Returns true if x is a NaN else false. x must be a double!

[double](#)

Converts to double

[section](#)

[top](#)

installer/clean

(clean dir)

Remove Java libraries (except any Jansi library) and TTF font files from the specified directory.

The removal does NOT recursively traverse the directory tree.

```
(do
  (load-module :installer)
  (installer/clean (repl/libs-dir)))
```

SEE ALSO

[installer/install](#)

Install Java artifacts and its dependencies.

[installer/install-module](#)

Install the 3rdparty libraries for a Venice extension module.

[installer/install-libs](#)

Install Java libraries (artifacts). Does not install the library's dependencies!

[section](#)

[top](#)

installer/install

(install artifacts options*)

Install Java artifacts and its dependencies.

Options:

:dir path	download dir, defaults to "." except when run in a REPL where it defaults to the value of <code>(repl/libs-dir)</code>
:silent {true,false}	if silent is true does not show a progress bar. Defaults to <code>true</code>
:force {true,false}	if force is true download the artifact even if it exist already on the download dir, else skip the download if it exists. Defaults to <code>true</code> .
:dry-run {true,false}	just list the artifact dependencies without actually installing them. Defaults to <code>true</code> .

```
(do
  (load-module :installer)
  (installer/install [ "dev.langchain4j:langchain4j:0.30.0"
                      "dev.langchain4j:langchain4j-open-ai:0.30.0" ]
    :dir (repl/libs-dir)
    :silent false))
```

SEE ALSO

[installer/install-libs](#)

Install Java libraries (artifacts). Does not install the library's dependencies!

[installer/install-module](#)

Install the 3rdparty libraries for a Venice extension module.

[installer/clean](#)

Remove Java libraries (except any Jansi library) and TTF font files from the specified directory.

[section](#)

[top](#)

installer/install-demo

(install-demo options*)

Install all demo fonts and the 3rdparty libraries for all Venice extension modules that require Java libraries:

- :jansi
- :bouncycastle
- :docx4j-8
- :excel
- :jtokkit
- :langchain
- :pdf
- :postgresql-jdbc
- :qdrant-client
- :qrbill
- :qrcode
- :tomcat
- :xchart
- :yaml
- :postgresql-jdbc

Options:

:dir path	download dir, defaults to "." except when run in a REPL where it defaults to the value of <code>(repl/libs-dir)</code>
:silent {true,false}	if silent is true does not show a progress bar, defaults to true
:clean {true,false}	if clean is true cleans the install dir before installing, defaults to false
:force {true,false}	if force is true download the artifact even if it exist already on the download dir, else skip the download if it exists. Defaults to true.

In the REPL run:

```
venice> (load-module :installer)
venice> (installer/install-demo)
venice> !restart
```

The installed libraries and fonts can be cleaned with:

```
(installer/clean (repl/libs-dir))
```

This removes all JAR lib and the fonts, except the JAnsi and the Venice libs.

```
;; install the demo modules
(do
  (load-module :installer)
  (installer/install-demo :dir (repl/libs-dir) :silent false))

;; clean install dir before installing the demo modules
(do
```

```
(load-module :installer)
(installer/install-demo :dir (repl/libs-dir) :silent false :clean true))
```

SEE ALSO

[installer/install-demo-fonts](#)

Install the Venice demo fonts.

[installer/clean](#)

Remove Java libraries (except any Jansi library) and TTF font files from the specified directory.

[section](#)

[top](#)

installer/install-demo-fonts

```
(install-demo-fonts options*)
```

Install the Venice demo fonts.

Installs the open source font families from

Family	Download family ref	Type	License
Open Sans	open-sans	TTF	Apache License v2
Roboto	roboto	TTF	Apache License v2
Source Code Pro	source-code-pro	OTF	SIL Open Font License v1.10
JetBrains Mono	jetbrains-mono	TTF	Apache License v2

Downloads the font families from the [Font Squirrel](#) repository

Options:

:dir path download dir, defaults to "." except when run in a REPL where it defaults to the value of `(repl/libs-dir)`

:silent {true,false} if silent is true does not show a progress bar, defaults to true

In the REPL run:

```
venice> (load-module :installer)
venice> (installer/install-demo-fonts)
venice> !restart
```

The installed libraries and fonts can be cleaned with:

```
(installer/clean (repl/libs-dir))
```

```
(do
  (load-module :installer)
  (installer/install-demo-fonts :dir (repl/libs-dir) :silent false))
```

SEE ALSO

[installer/install-demo](#)

Install all demo fonts and the 3rdparty libraries for all Venice extension modules that require Java libraries:

[installer/clean](#)

Remove Java libraries (except any Jansi library) and TTF font files from the specified directory.

[section](#)

[top](#)

installer/install-libs

```
(install-libs libs options*)
```

Install Java libraries (artifacts). Does not install the library's dependencies!

Options:

:dir path	download dir, defaults to "." except when run in a REPL where it defaults to the value of <code>(repl/libs-dir)</code>
:silent {true,false}	if silent is true does not show a progress bar, defaults to true
:force {true,false}	if force is true download the artifact even if it exist already on the download dir, else skip the download if it exists. Defaults to true.

```
(do
  (load-module :installer)
  (installer/install-libs ["org.fusesource.jansi:jansi:2.4.1"]
    :dir (repl/libs-dir)
    :silent false))
```

SEE ALSO

[installer/install](#)

Install Java artifacts and its dependencies.

[installer/install-module](#)

Install the 3rdparty libraries for a Venice extension module.

[installer/clean](#)

Remove Java libraries (except any Jansi library) and TTF font files from the specified directory.

[section](#)

[top](#)

installer/install-module

```
(install-module name options*)
```

Install the 3rdparty libraries for a Venice extension module.

Options:

:dir path	download dir, defaults to "." except when run in a REPL where it defaults to the value of <code>(repl/libs-dir)</code>
:silent {true,false}	if silent is true does not show a progress bar, defaults to true
:force {true,false}	if force is true download the artifact even if it exist already on the download dir, else skip the download if it exists. Defaults to true.

```
(do
  (load-module :installer)
  (installer/install-module :pdf :dir (repl/libs-dir) :silent false))
```

SEE ALSO

[installer/install](#)

Install Java artifacts and its dependencies.

[installer/install-libs](#)

Install Java libraries (artifacts). Does not install the library's dependencies!

[installer/clean](#)

Remove Java libraries (except any Jansi library) and TTF font files from the specified directory.

instance-of?

```
(instance-of? type x)
```

Returns true if x is an instance of the given type

```
(instance-of? :long 500)  
=> true
```

```
(instance-of? :java.math.BigInteger 500)  
=> false
```

SEE ALSO

[type](#)

Returns the type of x.

[supertype](#)

Returns the super type of x.

[supertypes](#)

Returns the super types of x.

int

```
(int x)
```

Converts to int

```
(int 1)  
=> 1I
```

```
(int nil)  
=> 0I
```

```
(int false)  
=> 0I
```

```
(int true)  
=> 1I
```

```
(int 1.2)  
=> 1I
```

```
(int 1.2F)  
=> 1I
```

```
(int 1.2M)  
=> 1I
```



```
(int "1")  
=> 1I
```

```
(int (char "A"))  
=> 65I
```

[section](#)[top](#)

int-array

```
(int-array coll)  
(int-array len)  
(int-array len init-val)
```

Returns an array of Java primitive ints containing the contents of coll or returns an array with the given length and optional init value.

To create an array of :java.lang.Integer use:

```
(make-array :java.lang.Integer 3)
```

```
(int-array '(1I 2I 3I))  
=> [1, 2, 3]
```

```
(int-array '(1I 2 3.2 3.56M))  
=> [1, 2, 3, 3]
```

```
(int-array 10)  
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
(int-array 10 42I)  
=> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

SEE ALSO

[java-int-list](#)

Converts a Venice list/vector to a Java Integer list

[section](#)[top](#)

int?

```
(int? n)
```

Returns true if n is an int

```
(int? 4I)  
=> true
```

```
(int? 4)  
=> false
```

```
(int? 3.1)  
=> false
```

```
(int? true)
=> false
```

```
(int? nil)
=> false
```

```
(int? {})
=> false
```

[section](#)[top](#)

interleave

```
(interleave c1 c2)
(interleave c1 c2 & colls)
```

Returns a collection of the first item in each coll, then the second etc.

Supports lazy sequences as long as at least one collection is not a lazy sequence.

```
(interleave [:a :b :c] [1 2])
=> (:a 1 :b 2)
```

```
(interleave [:a :b :c] (lazy-seq 1 inc))
=> (:a 1 :b 2 :c 3)
```

```
(interleave (lazy-seq (constantly :v)) [1 2 3])
=> (:v 1 :v 2 :v 3)
```

[section](#)[top](#)

interpose

```
(interpose sep coll)
```

Returns a collection of the elements of coll separated by sep.

```
(interpose ", " [1 2 3])
=> (1 " , " 2 " , " 3)
```

```
(apply str (interpose ", " [1 2 3]))
=> "1, 2, 3"
```

[section](#)[top](#)

intersection

```
(intersection s1)
(intersection s1 s2)
(intersection s1 s2 & sets)
```

Return a set that is the intersection of the input sets

```
(intersection (set 1))
=> #{1}

(intersection (set 1 2) (set 2 3))
=> #{2}

(intersection (set 1 2) (set 3 4))
=> #{} 
```

SEE ALSO

[union](#)

Return a set that is the union of the input sets

[difference](#)

Return a set that is the first set without elements of the remaining sets

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[conj](#)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item) and (conj item) returns item.

[disj](#)

Returns a new set with the x, xs removed.

[section](#)

[top](#)

into

```
(into)
(into to)
(into to from) 
```

Returns a new coll consisting of to coll with all of the items of from coll conjoined.

```
(into (sorted-map) [ [:a 1] [:c 3] [:b 2] ])
=> {:a 1 :b 2 :c 3}

(into (sorted-map) [ {:a 1} {:c 3} {:b 2} ])
=> {:a 1 :b 2 :c 3}

(into (sorted-map) [(map-entry :b 2) (map-entry :c 3) (map-entry :a 1)])
=> {:a 1 :b 2 :c 3}

(into (sorted-map) {:b 2 :c 3 :a 1})
=> {:a 1 :b 2 :c 3}

(into [] {:a 1, :b 2})
=> [[:a 1] [:b 2]]

(into [] '(1 2 3))
=> [1 2 3]

(into '() '(1 2 3))
=> (3 2 1) 
```

```

(into [1 2 3] '(4 5 6))
=> [1 2 3 4 5 6]

(into '(1 2 3) '(4 5 6))
=> (6 5 4 1 2 3)

(into [] (bytebuf [0 1 2]))
=> [0 1 2]

(into '() (bytebuf [0 1 2]))
=> (0 1 2)

(into [] "abc")
=> [#\a #\b #\c]

(into '() "abc")
=> (#\a #\b #\c)

```

SEE ALSO

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[merge](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from ...

[section](#)
[top](#)

into!

```

(into!)
(into! to)
(into! to from)

```

Adds all of the items of 'from' conjoined to the mutable 'to' collection

```

(into! (queue) [1 2 3 4])
=> (1 2 3 4)

(into! (stack) [1 2 3 4])
=> (4 3 2 1)

(do
  (into! (. :java.util.concurrent.CopyOnWriteArrayList :new)
    (doto (. :java.util.ArrayList :new)
      (. :add 3)
      (. :add 4))))
=> (3 4)

(do
  (into! (. :java.util.concurrent.CopyOnWriteArrayList :new)
    '(3 4)))
=> (3 4)

```

SEE ALSO

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[merge](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from ...

[section](#)[top](#)

io/->path

```
(io/->path f)
```

Converts to a `java.nio.Path`. `f` must be a file or a string (file path).

```
(io/->path "some.txt")  
=> some.txt
```

```
(io/->path "/tmp/test/some.txt")  
=> /tmp/test/some.txt
```

```
(io/->path (io/file "/tmp/test/some"))  
=> /tmp/test/some
```

SEE ALSO

[io/path?](#)

Returns true if `f` is a `java.nio.Path`.

[section](#)[top](#)

io/->uri

```
(io/->uri s)  
(io/->uri scheme user-info host port path)  
(io/->uri scheme user-info host port path query)  
(io/->uri scheme user-info host port path query fragment)
```

Converts `s` to an URI or builds an URI from its spec elements.

`s` may be:

- a string (a spec string to be parsed as a URI.)
- a `java.io.File`
- a `java.nio.file.Path`
- a `java.net.URL`

Arguments:

scheme Scheme name
userInfo User name and authorization information
host Host name
port Port number
path Path
query Query
fragment Fragment

```
(io/->uri "file:/tmp/test.txt")  
=> file:/tmp/test.txt
```

```
(io/->uri (io/file "/tmp/test.txt"))
=> file:/tmp/test.txt

(io/->uri (io/->url (io/file "/tmp/test.txt")))
=> file:/tmp/test.txt

(str (io/->uri (io/file "/tmp/test.txt")))
=> "file:/tmp/test.txt"

;; to create an URL from spec details:
(io/->uri "http" nil "foo.org" 8080 "/info.html" nil nil)
=> http://foo.org:8080/info.html
```

SEE ALSO

[io/file](#)

Returns a java.io.File from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[io/->url](#)

Converts s to an URL or builds an URL from its spec elements.

[section](#)

[top](#)

io/->url

```
(io/->url s)
(io/->url protocol host port file)
```

Converts s to an URL or builds an URL from its spec elements.

s may be:

- a string (a spec string to be parsed as a URL.)
- a java.io.File
- a java.nio.file.Path
- a java.net.URI

Arguments:

protocol the name of the protocol to use.
host the name of the host.
port the port number on the host.
file the file on the host

```
(io/->url "file:/tmp/test.txt")
=> file:/tmp/test.txt

(io/->url (io/file "/tmp/test.txt"))
=> file:/tmp/test.txt

(io/->url (io/->uri (io/file "/tmp/test.txt")))
=> file:/tmp/test.txt

(str (io/->url (io/file "/tmp/test.txt")))
=> "file:/tmp/test.txt"

;; to create an URL from spec details:
(io/->url "http" "foo.org" 8080 "/info.html")
=> http://foo.org:8080/info.html
```

SEE ALSO

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[io/->uri](#)

Converts `s` to an URI or builds an URI from its spec elements.

[section](#)

[top](#)

io/await-for

```
(io/await-for timeout time-unit file & modes)
```

Blocks the current thread until the file has been created, deleted, or modified according to the passed modes `{:created, :deleted, :modified}`, or the timeout has elapsed. Returns logical `false` if returning due to timeout, logical `true` otherwise.

Supported time units are: `{:milliseconds, :seconds, :minutes, :hours, :days}`

```
(io/await-for 10 :seconds "/tmp/data.json" :created)
```

SEE ALSO

[io/watch-dir](#)

Watch a directory for changes, and call the function `event-fn` when it does. Calls the optional `failure-fn` if errors occur. On closing ...

[section](#)

[top](#)

io/buffered-reader

```
(io/buffered-reader f & options)
```

Create a `java.io.Reader` from `f`.

`f` may be a:

- `string`
- `bytebuffer`
- `java.io.File`, e.g. `(io/file "/temp/foo.json")`
- `java.nio.file.Path`
- `java.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `java.net.URI`

Options:

`:encoding enc` e.g.: `:encoding :utf-8`, defaults to `:utf-8`

`io/buffered-reader` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

Note: The caller is responsible for closing the reader!

```
(let [data (bytebuf [108 105 110 101 32 49 10 108 105 110 101 32 50])]
  (try-with [rd (io/buffered-reader data :encoding :utf-8)]
    (println (read-line rd))
    (println (read-line rd))))
```

```
line 1
line 2
=> nil

(try-with [rd (io/buffered-reader "1\n2\n3\n4")])
  (println (read-line rd))
  (println (read-line rd)))
1
2
=> nil
```

SEE ALSO

[read-line](#)

Without arg reads the next line from the stream that is the current value of *in*. With arg reads the next line from the passed stream ...

[io/string-reader](#)

Creates a java.io.StringReader from a string.

[io/buffered-writer](#)

Creates a java.io.Writer for f.

[section](#)

[top](#)

io/buffered-writer

(io/buffered-writer f & options)

Creates a `java.io.Writer` for f.

f may be a:

- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.file.Path`
- `java.io.OutputStream`
- `java.io.Writer` *Options:*

:append true/false e.g.: `:append true`, defaults to false

:encoding enc e.g.: `:encoding :utf-8`, defaults to :utf-8

`io/buffered-writer` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

SEE ALSO

[println](#)

Prints the values xs to the stream that is the current value of *out* or to the passed output stream os if given followed by a (newline).

[io/string-writer](#)

Creates a java.io.StringWriter.

[io/buffered-reader](#)

Create a java.io.Reader from f.

[section](#)

[top](#)

io/bytebuf-in-stream

(io/bytebuf-in-stream buf)

Returns a `java.io.InputStream` from a bytebuf.

Note: The caller is responsible for closing the stream!

```
(try-with [is (io/bytebuf-in-stream (bytebuf [97 98 99]))])
  ; do something with is
)
```

SEE ALSO

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` is. Supports the option `:binary` to either slurp binary or string data. For ...

[io/file-in-stream](#)

Returns a `java.io.InputStream` for the file `f`.

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string `s`.

[section](#)

[top](#)

io/bytebuf-out-stream

(io/bytebuf-out-stream)

Returns a new `java.io.ByteArrayOutputStream`.

Dereferencing a `:ByteArrayOutputStream` returns the captured bytebuf.

Note: The caller is responsible for closing the stream!

```
(try-with [os (io/bytebuf-out-stream)]
  (io/spit-stream os (bytebuf [97 98 99]) :flush true)
  (str/format-bytebuf @os ", " :prefix0x))
=> "0x61, 0x62, 0x63"
```

SEE ALSO

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` is. Supports the option `:binary` to either slurp binary or string data. For ...

[io/file-in-stream](#)

Returns a `java.io.InputStream` for the file `f`.

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string `s`.

[section](#)

[top](#)

io/capturing-print-stream

(io/capturing-print-stream)

Creates a new capturing print stream.

Dereferencing a capturing print stream returns the captured string.

Note: The caller is responsible for closing the stream!

```
(try-with [ps (io/capturing-print-stream)]
  (binding [*out* ps]
    (println 100)
    (println 200)
    (flush)
    @ps))
=> "100\n200\n"
```

```
(try-with [ps (io/capturing-print-stream)]
  (println ps 100)
  (println ps 200)
  (flush ps)
  @ps)
=> "100\n200\n"
```

[section](#)[top](#)

io/classpath-resource?

```
(io/classpath-resource? name)
```

Returns true if the classpath resource exists otherwise false.

```
(io/classpath-resource? "com/github/jlangch/venice/images/venice.png")
=> true
```

SEE ALSO

[io/load-classpath-resource](#)

Loads a classpath resource. Returns a bytebuf

[section](#)[top](#)

io/close

```
(io/close s)
```

Closes a `:java.io.InputStream`, `:java.io.OutputStream`, `:java.io.Reader`, or a `:java.io.Writer`.

Often it is more elegant to use try-with to let Venice implicitly close the stream when its leaves the scope:

```
(let [file (io/file "foo.txt")]
  (try-with [is (io/file-in-stream file)]
    (io/slurp-stream is :binary false)))
```

SEE ALSO

[io/flush](#)

Flushes a `:java.io.OutputStream` or a `:java.io.Writer`.

[section](#)[top](#)

io/close-watcher

```
(io/close-watcher watcher)
```

Closes a watcher created from 'io/watch-dir'.

SEE ALSO

[io/watch-dir](#)

Watch a directory for changes, and call the function event-fn when it does. Calls the optional failure-fn if errors occur. On closing ...

[io/registered-watch-dirs](#)

Returns the registred watch directories of a watcher.

[section](#)

[top](#)

io/copy-file

```
(io/copy-file source dest & options)
```

Copies source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must be a file, a string (file path), or an `java.io.OutputStream`.

Options:

:replace true/false	e.g.: if true replace an existing file, defaults to false
:copy-attributes true/false	e.g.: if true copy attributes to the new file, defaults to false
:no-follow-links true/false	e.g.: if true do not follow symbolic links, defaults to false

SEE ALSO

[io/copy-files-glob](#)

Copies all files that match the glob pattern from a source to a destination directory. src-dir and dst-dir must be a file or a string ...

[io/copy-file-tree](#)

Copies a file tree from source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must ...

[io/move-file](#)

Moves source to target. Returns nil or throws a VncException. Source and target must be a file or a string (file path).

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If f is a directory the directory must be empty. f ...

[io/touch-file](#)

Updates the lastModifiedTime of the file to the current time, or creates a new empty file if the file doesn't already exist. File must ...

[io/copy-stream](#)

Copies the input stream to the output stream. Returns nil on sucess or throws a VncException on failure. Input and output must be a ...

[section](#)

[top](#)

io/copy-file-tree

```
(io/copy-file-tree source dest & options)
```

Copies a file tree from source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must be a file, a string (file path), or an `java.io.OutputStream`.

Options:

:replace true/false	e.g.: if true replace an existing file, defaults to false

`:copy-attributes true/false` e.g.: if true copy attributes to the new file, defaults to false
`:no-follow-links true/false` e.g.: if true do not follow symbolic links, defaults to false

SEE ALSO

[io/copy-file](#)

Copies source to dest. Returns nil or throws a `VncException`. Source must be a file or a string (file path), dest must be a file, a ...

[io/copy-files-glob](#)

Copies all files that match the glob pattern from a source to a destination directory. `src-dir` and `dst-dir` must be a file or a string ...

[io/move-file](#)

Moves source to target. Returns nil or throws a `VncException`. Source and target must be a file or a string (file path).

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If `f` is a directory the directory must be empty. `f` ...

[io/touch-file](#)

Updates the `lastModifiedTime` of the file to the current time, or creates a new empty file if the file doesn't already exist. File must ...

[io/copy-stream](#)

Copies the input stream to the output stream. Returns nil on success or throws a `VncException` on failure. Input and output must be a ...

[section](#)

[top](#)

io/copy-files-glob

`(io/copy-files-glob src-dir dst-dir glob & options)`

Copies all files that match the glob pattern from a source to a destination directory. `src-dir` and `dst-dir` must be a file or a string (file path).

Options:

`:replace true/false` e.g.: if true replace an existing file, defaults to false
`:copy-attributes true/false` e.g.: if true copy attributes to the new file, defaults to false
`:no-follow-links true/false` e.g.: if true do not follow symbolic links, defaults to false

Globbing patterns

<code>*.txt</code>	Matches a path that represents a file name ending in <code>.txt</code>
<code>*.*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with <code>.txt</code> or <code>.xml</code>
<code>foo.?[xy]</code>	Matches file names starting with <code>foo.</code> and a single character extension followed by a 'x' or 'y' character
<code>/home/*/*</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

Ranges

The pattern `[A-E]` would match any character that included ABCDE. Ranges can be used in conjunction with each other to make powerful patterns. Alphanumerical strings are matched by `[A-Za-z0-9]`. This would match the following:

- `[A-Z]` All uppercase letters from A to Z
- `[a-z]` All lowercase letters from a to z
- `[0-9]` All numbers from 0 to 9

Complementation

Globs can be used in complement with special characters that can change how the pattern works. The two complement characters are exclamation marks `(!)` and backslashes `(\\)`.

The exclamation mark can negate a pattern that it is put in front of. As `[CBR]at` matches Cat, Bat, or Rat the negated pattern `[!CBR]at` matches anything like Kat, Pat, or Vat.

Backslashes are used to remove the special meaning of single characters `'?'`, `'*'`, and `'['`, so that they can be used in patterns.

```
(io/copy-files-glob "from" "to" "*.log")
```

SEE ALSO

[io/copy-file](#)

Copies source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must be a file, a ...

[io/copy-file-tree](#)

Copies a file tree from source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must ...

[io/move-files-glob](#)

Move all files that match the glob pattern from a source to a destination directory. src-dir and dst-dir must be a file or a string (file path).

[io/delete-files-glob](#)

Removes all files in a directory that match the glob pattern. dir must be a file or a string (file path).

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. dir must be a file or a string (file path). Returns files as java.io.File

[section](#)

[top](#)

io/copy-stream

```
(io/copy-stream in-stream out-stream)
```

Copies the input stream to the output stream. Returns `nil` on success or throws a VncException on failure. Input and output must be a `java.io.InputStream` and `java.io.OutputStream`.

SEE ALSO

[io/copy-file](#)

Copies source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must be a file, a ...

[section](#)

[top](#)

io/create-hard-link

```
(io/create-hard-link link target)
```

Creates a hard link to a target. link and target must be a file or a string (file path).

```
(io/create-hard-link "/tmp/hard-link" "/tmp/test.txt")
```

SEE ALSO

[io/create-symbolic-link](#)

Creates a symbolic link to a target. link and target must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file f exists and is a symbolic link. f must be a file or a string (file path).

[section](#)

[top](#)

io/create-symbolic-link

```
(io/create-symbolic-link link target)
```

Creates a symbolic link to a target. link and target must be a file or a string (file path).

```
(io/create-symbolic-link "/tmp/sym-link" "/tmp/test.txt")
```

SEE ALSO

[io/create-hard-link](#)

Creates a hard link to a target. link and target must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file f exists and is a symbolic link. f must be a file or a string (file path).

[section](#)

[top](#)

io/default-charset

```
(io/default-charset)
```

Returns the default charset.

[section](#)

[top](#)

io/deflate

```
(io/deflate bytebuf)
```

deflates (compresses) a bytebuf using ZLIB compression.

```
(> (bytebuf-from-string "abcdef" :utf-8)
  (io/deflate))
=> [120 156 75 76 74 78 73 77 3 0 8 30 2 86]
```

SEE ALSO

[io/inflate](#)

inflates (decompresses) a bytebuf using ZLIB compression.

[section](#)

[top](#)

io/delete-file

```
(io/delete-file f & files)
```

Deletes one or multiple files. Silently skips delete if the file does not exist. If f is a directory the directory must be empty. f must be a file or a string (file path).

SEE ALSO

[io/delete-files-glob](#)

Removes all files in a directory that match the glob pattern. dir must be a file or a string (file path).

[io/delete-file-tree](#)

Deletes a file or a directory with all its content. Silently skips delete if the file or directory does not exist. f must be a file ...

[io/delete-file-on-exit](#)

Requests that the files or directories be deleted when the virtual machine terminates. Files (or directories) are deleted in the reverse ...

[io/copy-file](#)

Copies source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must be a file, a ...

[io/move-file](#)

Moves source to target. Returns nil or throws a VncException. Source and target must be a file or a string (file path).

[section](#)[top](#)

io/delete-file-on-exit

```
(io/delete-file-on-exit f & fs)
```

Requests that the files or directories be deleted when the virtual machine terminates. Files (or directories) are deleted in the reverse order that they are registered. Invoking this method to delete a file or directory that is already registered for deletion has no effect. Deletion will be attempted only for normal termination of the virtual machine, as defined by the Java Language Specification.

f must be a file or a string (file path).

```
(let [file1 (io/temp-file "test-", ".data")
      file2 (io/temp-file "test-", ".data")]
  (io/delete-file-on-exit file1 file2)
  (io/spit file1 "123")
  (io/spit file2 "ABC"))
```

SEE ALSO

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If f is a directory the directory must be empty. f ...

[io/delete-file-tree](#)

Deletes a file or a directory with all its content. Silently skips delete if the file or directory does not exist. f must be a file ...

[io/delete-files-glob](#)

Removes all files in a directory that match the glob pattern. dir must be a file or a string (file path).

[section](#)[top](#)

io/delete-file-tree

```
(io/delete-file-tree f & files)
```

Deletes a file or a directory with all its content. Silently skips delete if the file or directory does not exist. f must be a file or a string (file path)

SEE ALSO

[io/delete-files-glob](#)

Removes all files in a directory that match the glob pattern. dir must be a file or a string (file path).

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If `f` is a directory the directory must be empty. `f ...`

[io/delete-file-on-exit](#)

Requests that the files or directories be deleted when the virtual machine terminates. Files (or directories) are deleted in the reverse ...

[section](#)

[top](#)

io/delete-files-glob

```
(io/delete-files-glob dir glob)
```

Removes all files in a directory that match the glob pattern. `dir` must be a file or a string (file path).

Globbering patterns

<code>*.txt</code>	Matches a path that represents a file name ending in <code>.txt</code>
<code>*.*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with <code>.txt</code> or <code>.xml</code>
<code>foo.?[xy]</code>	Matches file names starting with <code>foo.</code> and a single character extension followed by a 'x' or 'y' character
<code>/home/*/*</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

Ranges

The pattern `[A-E]` would match any character that included ABCDE. Ranges can be used in conjunction with each other to make powerful patterns. Alphanumerical strings are matched by `[A-Za-z0-9]`. This would match the following:

- `[A-Z]` All uppercase letters from A to Z
- `[a-z]` All lowercase letters from a to z
- `[0-9]` All numbers from 0 to 9

Complementation

Globs can be used in complement with special characters that can change how the pattern works. The two complement characters are exclamation marks `(!)` and backslashes `(\\)`.

The exclamation mark can negate a pattern that it is put in front of. As `[CBR]at` matches Cat, Bat, or Rat the negated pattern `[!CBR]at` matches anything like Kat, Pat, or Vat.

Backslashes are used to remove the special meaning of single characters `'?'`, `'*'`, and `'['`, so that they can be used in patterns.

```
(io/delete-files-glob "." "*.log")
```

SEE ALSO

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If `f` is a directory the directory must be empty. `f ...`

[io/delete-file-tree](#)

Deletes a file or a directory with all its content. Silently skips delete if the file or directory does not exist. `f` must be a file ...

[io/move-files-glob](#)

Move all files that match the glob pattern from a source to a destination directory. `src-dir` and `dst-dir` must be a file or a string (file path).

[io/copy-files-glob](#)

Copies all files that match the glob pattern from a source to a destination directory. `src-dir` and `dst-dir` must be a file or a string ...

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

io/download

```
(io/download uri & options)
```

Downloads the content from the uri and reads it as text (string) or binary (bytebuf). Supports http and https protocols!

Options:

:binary b	e.g.: <code>:binary true</code> , defaults to false
:user-agent agent	e.g.: <code>:user-agent "Mozilla"</code> , defaults to nil
:encoding enc	e.g.: <code>:encoding :utf-8</code> , defaults to :utf-8
:user u	optional user for basic authentication
:password p	optional password for basic authentication
:follow-redirects b	e.g.: <code>:follow-redirects true</code> , defaults to false
:conn-timeout val	e.g.: <code>:conn-timeout 10000</code> , connection timeout in milliseconds. 0 is interpreted as an infinite timeout.
:read-timeout val	e.g.: <code>:read-timeout 10000</code> , read timeout in milliseconds. 0 is interpreted as an infinite timeout.
:progress-fn fn	an optional progress function that takes 2 args [1] progress (0..100%) [2] status {:start :progress :end :failed}
:debug-fn fn	an optional debug function that takes a message as argument

Note:

If the server returns the HTTP response status code 403 (*Access Denied*) sending a user agent like "Mozilla" may fool the website and solve the problem.

To debug pass a printing function like: `(io/download https://foo.org/bar :debug-fn println)`

```
(-<> "https://live.staticflickr.com/65535/51007202541_ea453871d8_o_d.jpg"
  (io/download <> :binary true :user-agent "Mozilla")
  (io/spit "space-x.jpg" <>))

(do
  (load-module :ansi)
  (-<> "https://live.staticflickr.com/65535/51007202541_ea453871d8_o_d.jpg"
    (io/download <> :binary true
                  :user-agent "Mozilla"
                  :progress-fn (ansi/progress :caption "Download:"))
    (io/spit "space-x.jpg" <>)))
```

io/exists-dir?

```
(io/exists-dir? f)
```

Returns true if the file f exists and is a directory. f must be a file or a string (file path).

```
(io/exists-dir? (io/file "/temp"))
=> false
```

SEE ALSO

[io/exists-file?](#)

Returns true if the file `f` exists and is a file. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[section](#)[top](#)

io/exists-file?

```
(io/exists-file? f)
```

Returns true if the file `f` exists and is a file. `f` must be a file or a string (file path).

```
(io/exists-file? "/tmp/test.txt")  
=> false
```

SEE ALSO

[io/exists-dir?](#)

Returns true if the file `f` exists and is a directory. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[section](#)[top](#)

io/exists?

```
(io/exists? f)
```

Returns true if the file or directory `f` exists. `f` must be a file or a string (file path).

```
(io/exists? "/tmp/test.txt")  
=> false
```

SEE ALSO

[io/exists-file?](#)

Returns true if the file `f` exists and is a file. `f` must be a file or a string (file path).

[io/exists-dir?](#)

Returns true if the file `f` exists and is a directory. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[section](#)[top](#)

io/file

```
(io/file path)  
(io/file parent child)  
(io/file parent child & children)
```

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string (file path), child and children must be strings.

```
(io/file "/tmp/test.txt")
=> /tmp/test.txt

(io/file "/temp" "test.txt")
=> /temp/test.txt

(io/file "/" "temp" "test" "test.txt")
=> /temp/test/test.txt

(io/file (io/file "/" "temp") "test" "test.txt")
=> /temp/test/test.txt

(io/file (. :java.io.File :new "/tmp/test.txt"))
=> /tmp/test.txt

;; Windows:
;; (io/file "C:\\tmp\\test.txt")
;; (io/file "C:/tmp/test.txt")

;; (io/file "C:" "tmp" "test.txt")
```

SEE ALSO

[io/file-name](#)

Returns the name of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-parent](#)

Returns the parent file of the file `f`. `f` must be a file or a string (file path).

[io/file-path](#)

Returns the path of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-absolute](#)

Returns the absolute path of the file `f`. `f` must be a file or a string (file path).

[io/file-canonical](#)

Returns the canonical path of the file `f`. `f` must be a file or a string (file path).

[str/normalize-utf](#)

Normalizes an UTF string.

[section](#)

[top](#)

io/file-absolute

```
(io/file-absolute f)
```

Returns the absolute path of the file `f`. `f` must be a file or a string (file path).

```
(io/file-absolute (io/file "/tmp/test/x.txt"))
=> /tmp/test/x.txt
```

SEE ALSO

[io/file-path](#)

Returns the path of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-canonical](#)

Returns the canonical path of the file `f`. `f` must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[io/file-absolute?](#)

Returns true if file `f` has an absolute path else false. `f` must be a file or a string (file path).

[str/normalize-utf](#)

Normalizes an UTF string.

[section](#)[top](#)

io/file-absolute?

```
(io/file-absolute? f)
```

Returns true if file `f` has an absolute path else false. `f` must be a file or a string (file path).

```
(io/file-absolute? (io/file "/tmp/test/x.txt"))  
=> true
```

SEE ALSO

[io/file-path](#)

Returns the path of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-canonical](#)

Returns the canonical path of the file `f`. `f` must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[io/file-absolute](#)

Returns the absolute path of the file `f`. `f` must be a file or a string (file path).

[section](#)[top](#)

io/file-basename

```
(io/file-basename f)
```

Returns the base name (file name without file extension) of the file `f` as a string. `f` must be a file or a string (file path).

```
(io/file-basename (io/file "/tmp/test/x.txt"))  
=> "x"
```

SEE ALSO

[io/file-name](#)

Returns the name of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-parent](#)

Returns the parent file of the file `f`. `f` must be a file or a string (file path).

[io/file-ext](#)

Returns the file extension of a file. `f` must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[str/normalize-utf](#)

Normalizes an UTF string.

[section](#)[top](#)

io/file-can-execute?

```
(io/file-can-execute? f)
```

Returns true if the file or directory `f` exists and can be executed. `f` must be a file or a string (file path).

```
(io/file-can-execute? "/tmp/test.txt")
```

SEE ALSO

[io/file-set-executable](#)

Set the owner's execute permission to the file or directory `f`. `f` must be a file or a string (file path).

[io/file-can-read?](#)

Returns true if the file or directory `f` exists and can be read. `f` must be a file or a string (file path).

[io/file-can-write?](#)

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

[io/file-hidden?](#)

Returns true if the file or directory `f` exists and is hidden. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[section](#)[top](#)

io/file-can-read?

```
(io/file-can-read? f)
```

Returns true if the file or directory `f` exists and can be read. `f` must be a file or a string (file path).

```
(io/file-can-read? "/tmp/test.txt")
```

SEE ALSO

[io/file-set-readable](#)

Set the owner's read permission to the file or directory `f`. `f` must be a file or a string (file path).

[io/file-can-write?](#)

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

[io/file-can-execute?](#)

Returns true if the file or directory `f` exists and can be executed. `f` must be a file or a string (file path).

[io/file-hidden?](#)

Returns true if the file or directory `f` exists and is hidden. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

io/file-can-write?

```
(io/file-can-write? f)
```

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

```
(io/file-can-write? "/tmp/test.txt")
```

SEE ALSO

[io/file-set-writable](#)

Set the owner's write permission to the file or directory `f`. `f` must be a file or a string (file path).

[io/file-can-read?](#)

Returns true if the file or directory `f` exists and can be read. `f` must be a file or a string (file path).

[io/file-can-execute?](#)

Returns true if the file or directory `f` exists and can be executed. `f` must be a file or a string (file path).

[io/file-hidden?](#)

Returns true if the file or directory `f` exists and is hidden. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

io/file-canonical

```
(io/file-canonical f)
```

Returns the canonical path of the file `f`. `f` must be a file or a string (file path).

```
(io/file-canonical (io/file "/tmp/test/../x.txt"))  
=> /private/tmp/x.txt
```

SEE ALSO

[io/file-path](#)

Returns the path of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-absolute](#)

Returns the absolute path of the file `f`. `f` must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[str/normalize-utf](#)

Normalizes an UTF string.

io/file-ext

```
(io/file-ext f)
```

Returns the file extension of a file. *f* must be a file or a string (file path).

```
(io/file-ext "some.txt")  
=> "txt"
```

```
(io/file-ext "/tmp/test/some.txt")  
=> "txt"
```

```
(io/file-ext "/tmp/test/some")  
=> nil
```

SEE ALSO

[io/file-ext?](#)

Returns true if the file *f* has the extension *ext*. *f* must be a file or a string (file path).

[io/file-basename](#)

Returns the base name (file name without file extension) of the file *f* as a string. *f* must be a file or a string (file path).

[section](#)

[top](#)

io/file-ext?

```
(io/file-ext? f ext & exts)
```

Returns true if the file *f* has the extension *ext*. *f* must be a file or a string (file path).

```
(io/file-ext? "/tmp/test/x.txt" "txt")  
=> true
```

```
(io/file-ext? (io/file "/tmp/test/x.txt") ".txt")  
=> true
```

```
(io/file-ext? "/tmp/test/x.docx" "doc" "docx")  
=> false
```

SEE ALSO

[io/file-ext](#)

Returns the file extension of a file. *f* must be a file or a string (file path).

[section](#)

[top](#)

io/file-hidden?

```
(io/file-hidden? f)
```

Returns true if the file or directory *f* exists and is hidden. *f* must be a file or a string (file path).

```
(io/file-hidden? "/tmp/test.txt")
```

SEE ALSO

[io/file-can-read?](#)

Returns true if the file or directory `f` exists and can be read. `f` must be a file or a string (file path).

[io/file-can-write?](#)

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

[io/file-can-execute?](#)

Returns true if the file or directory `f` exists and can be executed. `f` must be a file or a string (file path).

[io/symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[section](#)[top](#)

io/file-in-stream

```
(io/file-in-stream f)
```

Returns a `java.io.InputStream` for the file `f`.

`f` may be a:

- string file path, e.g: `"/temp/foo.json"`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`

`io/file-in-stream` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

Note: The caller is responsible for closing the stream!

SEE ALSO

[io/slurp](#)

Reads the content of file `f` as text (string) or binary (bytebuf).

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` `is`. Supports the option `:binary` to either slurp binary or string data. For ...

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string `s`.

[io/bytebuf-in-stream](#)

Returns a `java.io.InputStream` from a bytebuf.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)[top](#)

io/file-last-modified

```
(io/file-last-modified f)
```

Returns the last modification time (a Java `LocalDateTime`) of `f` or nil if `f` does not exist. `f` must be a file or a string (file path).

```
(io/file-last-modified "/tmp/test.txt")
```

SEE ALSO

[io/file-can-read?](#)

Returns true if the file or directory `f` exists and can be read. `f` must be a file or a string (file path).

io/file-can-write?

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

io/file-can-execute?

Returns true if the file or directory `f` exists and can be executed. `f` must be a file or a string (file path).

[section](#)[top](#)

io/file-matches-glob?

```
(io/file-matches-glob? glob f)
```

Returns true if the file `f` matches the glob pattern. `f` must be a file or a string (file path).

Globbing patterns

<code>*.txt</code>	Matches a path that represents a file name ending in <code>.txt</code>
<code>*,*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with <code>.txt</code> or <code>.xml</code>
<code>foo.?[xy]</code>	Matches file names starting with <code>foo.</code> and a single character extension followed by a <code>'x'</code> or <code>'y'</code> character
<code>/home/*/*</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

Ranges

The pattern `[A-E]` would match any character that included ABCDE. Ranges can be used in conjunction with each other to make powerful patterns. Alphanumerical strings are matched by `[A-Za-z0-9]`. This would match the following:

- `[A-Z]` All uppercase letters from A to Z
- `[a-z]` All lowercase letters from a to z
- `[0-9]` All numbers from 0 to 9

Complementation

Globs can be used in complement with special characters that can change how the pattern works. The two complement characters are exclamation marks `(!)` and backslashes `(\\)`.

The exclamation mark can negate a pattern that it is put in front of. As `[CBR]at` matches Cat, Bat, or Rat the negated pattern `[!CBR]at` matches anything like Kat, Pat, or Vat.

Backslashes are used to remove the special meaning of single characters `'?'`, `'*'`, and `'['`, so that they can be used in patterns.

```
(io/file-matches-glob? "*.log" "file.log")
```

```
=> true
```

```
(io/file-matches-glob? "**/*.log" "x/y/file.log")
```

```
=> true
```

```
(io/file-matches-glob? "**/*.log" "file.log") ; take care, doesn't match!
```

```
=> false
```

```
(io/file-matches-glob? (io/glob-path-matcher "*.log") (io/file "file.log"))
```

```
=> true
```

```
(io/file-matches-glob? (io/glob-path-matcher "**/*.log") (io/file "x/y/file.log"))
```

```
=> true
```

SEE ALSO

[io/glob-path-matcher](#)

Returns a file matcher for glob file patterns.

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

[section](#)

[top](#)

io/file-name

```
(io/file-name f)
```

Returns the name of the file `f` as a string. `f` must be a file or a string (file path).

```
(io/file-name (io/file "/tmp/test/x.txt"))  
=> "x.txt"
```

SEE ALSO

[io/file-basename](#)

Returns the base name (file name without file extension) of the file `f` as a string. `f` must be a file or a string (file path).

[io/file-parent](#)

Returns the parent file of the file `f`. `f` must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[str/normalize-utf](#)

Normalizes an UTF string.

[section](#)

[top](#)

io/file-normalize-utf

```
(io/file-normalize-utf file)  
(io/file-normalize-utf file form)
```

Normalizes the UTF string of a file path.

On MacOS file names with umlauts like `ä` are just encoded as `'a'` plus the combining dieresis character. Therefore an `'ä'` (`\u00FC`) and an `'ä'` (`a + \u0308`) from a MacOS file name are different! Under normal circumstances this not problem. But as soon as some file name processing is in place (comparing, matching, ...) this can result in strange behaviour due of the two different technical representations of umlaut characters.

The *form* argument defaults to `:NFC` and is one of:

- `:NFD` Canonical decomposition
- `:NFC` Canonical decomposition, followed by canonical composition
- `:NFKD` Compatibility decomposition
- `:NFKC` Compatibility decomposition, followed by canonical composition

Returns an UTF normalized `java.io.File` from a file path

See the function `str/normalize-utf` for details on UTF normalization.

```
(io/file-normalize-utf "/tmp/test_u\u0308.txt")
```

```
(io/file-normalize-utf (io/file "/tmp/test_u\u0308.txt"))
```

SEE ALSO

[str/normalize-utf](#)

Normalizes an UTF string.

[section](#)

[top](#)

io/file-out-stream

```
(io/file-out-stream f options)
```

Returns a `java.io.OutputStream` for the file `f`.

`f` may be a:

- string file path, e.g. `"/temp/foo.json"`
- `java.io.File`, e.g. `(io/file "/temp/foo.json")`

Options:

:append true/false e.g.: `:append true`, defaults to false

:encoding enc e.g.: `:encoding :utf-8`, defaults to `:utf-8`

`io/file-out-stream` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

Note: The caller is responsible for closing the stream!

SEE ALSO

[io/slurp](#)

Reads the content of file `f` as text (string) or binary (bytebuf).

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` `is`. Supports the option `:binary` to either slurp binary or string data. For ...

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string `s`.

[io/bytebuf-in-stream](#)

Returns a `java.io.InputStream` from a bytebuf.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)

[top](#)

io/file-parent

```
(io/file-parent f)
```

Returns the parent file of the file `f`. `f` must be a file or a string (file path).

```
(io/file-path (io/file-parent (io/file "/tmp/test/x.txt")))
=> "/tmp/test"
```

SEE ALSO

[io/file-name](#)

Returns the name of the file *f* as a string. *f* must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[section](#)[top](#)

io/file-path

```
(io/file-path f)
```

Returns the path of the file *f* as a string. *f* must be a file or a string (file path).

```
(io/file-path (io/file "/tmp/test/x.txt"))  
=> "/tmp/test/x.txt"
```

SEE ALSO

[io/file-absolute](#)

Returns the absolute path of the file *f*. *f* must be a file or a string (file path).

[io/file-canonical](#)

Returns the canonical path of the file *f*. *f* must be a file or a string (file path).

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[str/normalize-utf](#)

Normalizes an UTF string.

[section](#)[top](#)

io/file-path-slashify

```
(io/file-path-slashify f)
```

Returns the path of the file *f* as a string, turns backslashes into slashes.

f must be a file or a string (file path).

`C:\Users\foo\image.png -> C:/Users/foo/image.png`

Note: Windows only. On other OSs works identical to 'io/file-path'.

```
(io/file-path-slashify (io/file "C:" "Users" "foo" "image.png"))  
=> "C:/Users/foo/image.png"
```

SEE ALSO

[io/file-path](#)

Returns the path of the file *f* as a string. *f* must be a file or a string (file path).

[section](#)[top](#)

io/file-set-executable

```
(io/file-set-executable f executable owner-only)
```

Set the owner's execute permission to the file or directory *f*. *f* must be a file or a string (file path).

Returns true if and only if the operation succeeded. The operation will fail if the user does not have permission to change the access permissions of this abstract pathname. If 'readable' is false and the underlying file system does not implement a read permission, then the operation will fail.

If 'executable' is true sets the access permission to allow execute operations; if false to disallow execute operations.

If 'owner-only' is true the execute permission applies only to the owner's execute permission; otherwise, it applies to everybody. If the underlying file system can not distinguish the owner's execute permission from that of others, then the permission will apply to everybody, regardless of this value.

```
(io/file-set-executable "/tmp/test.txt" true true)
```

SEE ALSO

[io/file-can-execute?](#)

Returns true if the file or directory *f* exists and can be executed. *f* must be a file or a string (file path).

[io/file-set-readable](#)

Set the owner's read permission to the file or directory *f*. *f* must be a file or a string (file path).

[io/file-set-writable](#)

Set the owner's write permission to the file or directory *f*. *f* must be a file or a string (file path).

[section](#)

[top](#)

io/file-set-readable

```
(io/file-set-readable f readable owner-only)
```

Set the owner's read permission to the file or directory *f*. *f* must be a file or a string (file path).

Returns true if and only if the operation succeeded. The operation will fail if the user does not have permission to change the access permissions of this abstract pathname. If 'readable' is false and the underlying file system does not implement a read permission, then the operation will fail.

If 'readable' is true sets the access permission to allow read operations; if false to disallow read operations.

If 'owner-only' is true the read permission applies only to the owner's read permission; otherwise, it applies to everybody. If the underlying file system can not distinguish the owner's read permission from that of others, then the permission will apply to everybody, regardless of this value.

```
(io/file-set-readable "/tmp/test.txt" true true)
```

SEE ALSO

[io/file-can-read?](#)

Returns true if the file or directory *f* exists and can be read. *f* must be a file or a string (file path).

[io/file-set-writable](#)

Set the owner's write permission to the file or directory *f*. *f* must be a file or a string (file path).

[io/file-set-executable](#)

Set the owner's execute permission to the file or directory *f*. *f* must be a file or a string (file path).

[section](#)

[top](#)

io/file-set-writable

```
(io/file-set-writable f writable owner-only)
```

Set the owner's write permission to the file or directory `f`. `f` must be a file or a string (file path).

Returns true if and only if the operation succeeded. The operation will fail if the user does not have permission to change the access permissions of this abstract pathname. If 'writable' is false and the underlying file system does not implement a read permission, then the operation will fail.

If 'writable' is true sets the access permission to allow write operations; if false to disallow write operations.

If 'owner-only' is true the write permission applies only to the owner's write permission; otherwise, it applies to everybody. If the underlying file system can not distinguish the owner's write permission from that of others, then the permission will apply to everybody, regardless of this value.

```
(io/file-set-writable "/tmp/test.txt" true true)
```

SEE ALSO

[io/file-can-write?](#)

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

[io/file-set-readable](#)

Set the owner's read permission to the file or directory `f`. `f` must be a file or a string (file path).

[io/file-set-executable](#)

Set the owner's execute permission to the file or directory `f`. `f` must be a file or a string (file path).

[section](#)

[top](#)

io/file-size

```
(io/file-size f)
```

Returns the size of the file `f`. `f` must be a file or a string (file path).

```
(io/file-size "/tmp/test.txt")
```

SEE ALSO

[io/file](#)

Returns a java.io.File from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[section](#)

[top](#)

io/file-within-dir?

```
(io/file-within-dir? dir file)
```

Returns true if the file is within the dir else false.

The file and dir args must be absolute paths.

```
(io/file-within-dir? (io/file "/temp/foo")
                    (io/file "/temp/foo/img.png"))
=> true
```

```
(io/file-within-dir? (io/file "/temp/foo")
                     (io/file "/temp/foo/../bar/img.png"))
=> false
```

SEE ALSO

[io/file](#)

Returns a java.io.File from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

[section](#)

[top](#)

io/file?

```
(io/file? f)
```

Returns true if x is a :java.io.File.

```
(io/file? (io/file "/tmp/test.txt"))
=> true
```

[section](#)

[top](#)

io/filesystem-total-space

```
(io/filesystem-total-space)
(io/filesystem-total-space file)
```

Returns the total disk space in bytes. With no args returns the total disk space of the current working directory's file store. With a file argument returns the total disk space of the file store the file is located.

```
(io/filesystem-total-space)
```

SEE ALSO

[io/filesystem-usable-space](#)

Returns the usable disk space in bytes. With no args returns the usable disk space of the current working directory's file store. With ...

[section](#)

[top](#)

io/filesystem-usable-space

```
(io/filesystem-usable-space)
(io/filesystem-usable-space file)
```

Returns the usable disk space in bytes. With no args returns the usable disk space of the current working directory's file store. With a file argument returns the usable disk space of the file store the file is located.

```
(io/filesystem-usable-space)
```

SEE ALSO

[io/filesystem-total-space](#)

Returns the total disk space in bytes. With no args returns the total disk space of the current working directory's file store. With ...

[section](#)[top](#)

io/flush

```
(io/flush s)
```

Flushes a `:java.io.OutputStream` or a `:java.io.Writer`.

SEE ALSO

[io/close](#)

Closes a `:java.io.InputStream`, `:java.io.OutputStream`, `:java.io.Reader`, or a `:java.io.Writer`.

[section](#)[top](#)

io/glob-path-matcher

```
(io/glob-path-matcher pattern)
```

Returns a file matcher for glob file patterns.

Globbing patterns

<code>*.txt</code>	Matches a path that represents a file name ending in .txt
<code>*.*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with .txt or .xml
<code>foo.?[xy]</code>	Matches file names starting with foo. and a single character extension followed by a 'x' or 'y' character
<code>/home/**</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

Ranges

The pattern `[A-E]` would match any character that included ABCDE. Ranges can be used in conjunction with each other to make powerful patterns. Alphanumerical strings are matched by `[A-Za-z0-9]`. This would match the following:

- `[A-Z]` All uppercase letters from A to Z
- `[a-z]` All lowercase letters from a to z
- `[0-9]` All numbers from 0 to 9

Complementation

Globs can be used in complement with special characters that can change how the pattern works. The two complement characters are exclamation marks `(!)` and backslashes `(\)`.

The exclamation mark can negate a pattern that it is put in front of. As `[CBR]at` matches Cat, Bat, or Rat the negated pattern `[!CBR]at` matches anything like Kat, Pat, or Vat.

Backslashes are used to remove the special meaning of single characters `'?'`, `'*'`, and `'['`, so that they can be used in patterns.

```
(io/glob-path-matcher "*.log")
```

```
(io/glob-path-matcher "**/*.log")
```


SEE ALSO

[io/file-matches-glob?](#)

Returns true if the file `f` matches the glob pattern. `f` must be a file or a string (file path).

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

[section](#)

[top](#)

io/gzip

```
(io/gzip f)
```

gzips `f`. `f` may be a file, a string (file path), a bytebuf or an `InputStream`. Returns a bytebuf.

```
(->> (io/gzip "a.txt")
      (io/spit "a.gz"))
```

```
(io/gzip (bytebuf-from-string "abcdef" :utf-8))
```

SEE ALSO

[io/gzip?](#)

Returns true if `f` is a gzipped file. `f` may be a file, a string (file path), a bytebuf, or an `InputStream`

[io/ungzip](#)

ungzips `f`. `f` may be a file, a string (file path), a bytebuf, or an `InputStream`. Returns a bytebuf.

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/spit](#)

Opens file `f`, writes content, and then closes `f`. `f` may be a file or a string (file path). The content may be a string or a bytebuf.

[section](#)

[top](#)

io/gzip-to-stream

```
(io/gzip f os)
```

gzips `f` to the `OutputStream` `os`. `f` may be a file, a string (file path), a bytebuf, or an `InputStream`.

```
(do
  (import :java.io.ByteArrayOutputStream)
  (try-with [os (. :ByteArrayOutputStream :new)]
    (-> (bytebuf-from-string "abcdef" :utf-8)
        (io/gzip-to-stream os))
    (-> (. os :toByteArray)
        (io/ungzip)
        (bytebuf-to-string :utf-8))))
=> "abcdef"
```

SEE ALSO

[io/gzip](#)

gzips f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

[section](#)[top](#)

io/gzip?

```
(io/gzip? f)
```

Returns true if f is a gzipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

```
(-> (io/gzip (bytebuf-from-string "abc" :utf-8))
    (io/gzip?))
=> true
```

SEE ALSO

[io/gzip](#)

gzips f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

[section](#)[top](#)

io/in-stream?

```
(io/in-stream? is)
```

Returns true if 'is' is a `java.io.InputStream`

```
(try-with [is (io/string-in-stream "123")]
  (io/in-stream? is))
=> true
```

SEE ALSO

[io/out-stream?](#)

Returns true if 'os' is a `java.io.OutputStream`

[section](#)[top](#)

io/inflate

```
(io/inflate bytebuf)
```

inflates (decompresses) a bytebuf using ZLIB compression.

```
(-> (bytebuf-from-string "abcdef" :utf-8)
    (io/deflate)
    (io/inflate))
=> [97 98 99 100 101 102]
```

SEE ALSO

[io/deflate](#)

deflates (compresses) a bytebuf using ZLIB compression.

io/internet-avail?

```
(io/internet-avail?)  
(io/internet-avail? url)
```

Checks if an internet connection is present for a given url. Defaults to URL *http://www.google.com*.

```
(io/internet-avail?)
```

```
(io/internet-avail? "http://www.google.com")
```

io/list-file-tree

```
(io/list-file-tree dir)  
(io/list-file-tree dir filter-fn)
```

Lists all files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files found. The filter gets a `java.io.File` as argument.

Returns files as `java.io.File`

```
(io/list-file-tree "/tmp")
```

```
(io/list-file-tree "/tmp" #(io/file-ext? % ".log"))
```

SEE ALSO

[io/list-file-tree-lazy](#)

Returns a lazy sequence of all the files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional ...

[io/list-files](#)

Lists files in a directory. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files found.

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

io/list-file-tree-lazy

```
(io/list-file-tree-lazy dir)  
(io/list-file-tree-lazy dir filter-fn)
```

Returns a lazy sequence of all the files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files found. The filter gets a `java.io.File` as argument.

The lazy sequence returns files as `java.io.File`

```
(->> (io/list-file-tree-lazy "/tmp")
      (docoll println))

(->> (io/list-file-tree-lazy "/tmp" #(io/file-ext? % ".log"))
      (docoll println))
```

SEE ALSO

[io/list-file-tree](#)

Lists all files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files ...

[io/list-files](#)

Lists files in a directory. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files found.

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

[section](#)

[top](#)

io/list-files

```
(io/list-files dir)
(io/list-files dir filter-fn)
```

Lists files in a directory. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files found. The filter gets a `java.io.File` as argument.

Returns files as `java.io.File`

```
(io/list-files "/tmp")

(io/list-files "/tmp" #(io/file-ext? % ".log"))
```

SEE ALSO

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

[io/list-file-tree](#)

Lists all files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files ...

[io/list-file-tree-lazy](#)

Returns a lazy sequence of all the files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional ...

[section](#)

[top](#)

io/list-files-glob

```
(io/list-files-glob dir glob)
```

Lists all files in a directory that match the glob pattern. `dir` must be a file or a string (file path). Returns files as `java.io.File`

Globbing patterns

<code>*.txt</code>	Matches a path that represents a file name ending in <code>.txt</code>
<code>*.*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with <code>.txt</code> or <code>.xml</code>
<code>foo.?[xy]</code>	Matches file names starting with <code>foo.</code> and a single character extension followed by a <code>'x'</code> or <code>'y'</code>

	character
<code>/home/*/*</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

Ranges

The pattern `[A-E]` would match any character that included ABCDE. Ranges can be used in conjunction with each other to make powerful patterns. Alphanumeric strings are matched by `[A-Za-z0-9]`. This would match the following:

- `[A-Z]` All uppercase letters from A to Z
- `[a-z]` All lowercase letters from a to z
- `[0-9]` All numbers from 0 to 9

Complementation

Globs can be used in complement with special characters that can change how the pattern works. The two complement characters are exclamation marks `(!)` and backslashes `(\)`.

The exclamation mark can negate a pattern that it is put in front of. As `[CBR]at` matches Cat, Bat, or Rat the negated pattern `[!CBR]at` matches anything like Kat, Pat, or Vat.

Backslashes are used to remove the special meaning of single characters `'?'`, `'*'`, and `'['`, so that they can be used in patterns.

```
(io/list-files-glob "." "sample*.txt")
```

SEE ALSO

[io/list-files](#)

Lists files in a directory. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files found.

[io/list-file-tree](#)

Lists all files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional filter that filters the files ...

[io/list-file-tree-lazy](#)

Returns a lazy sequence of all the files in a directory tree. `dir` must be a file or a string (file path). `filter-fn` is an optional ...

section top

io/load-classpath-resource

```
(io/load-classpath-resource name)
```

Loads a classpath resource. Returns a bytebuf

```
(io/load-classpath-resource "com/github/jlangch/venice/images/venice.png")
=> [137 80 78 71 13 10 26 10 0 0 0 13 73 72 68 82 0 0 3 254 0 0 0 242 8 6 0 0 0 244 182 30 43 0 0 12 70 105 67
67 80 73 67 67 32 80 114 111 102 105 108 101 0 0 72 137 149 87 7 88 83 201 22 158 91 82 73 104 129 8 72 9 189
137 82 164 75 9 161 69 16 144 42 216 8 73 32 161 196 144 16 68 236 46 203 42 184 118 17 1 ...]
```

SEE ALSO

[io/classpath-resource?](#)

Returns true if the classpath resource exists otherwise false.

section top

io/log

```
(io/log logger-name level message)
(io/log logger-name level message exception)
```

Logs a message at a given level to a Java Util Logger (JUL).

```
(do
  ;; note: define the log filehandler just once at app startup!
  (io/log-filehandler "venice"
    "/var/log/myapp/venice_%g.log"
    16_000_000
    8)

  (io/log "venice" :info    "message 1")
  (io/log "venice" :warning "message 2")
  (io/log "venice" :severe  "message 3" (ex :VncException "test")))
```

SEE ALSO

[io/log-filehandler](#)

Creates a file handler for a Java Util Logger (JUL).

[section](#)

[top](#)

io/log-filehandler

```
(io/log-filehandler logger-name file-name-pattern)
(io/log-filehandler logger-name file-name-pattern file-size-limit)
(io/log-filehandler logger-name file-name-pattern file-size-limit max-file-count)
(io/log-filehandler logger-name file-name-pattern file-size-limit max-file-count, formatter)
```

Creates a file handler for a Java Util Logger (JUL).

The file name pattern determines the output file location and naming scheme:

- %t = system temp directory
- %h = user home directory
- %u = unique number to resolve conflicts
- %g = generation number for rotated logs
- %% = escapes the % character

If no "%g" field has been specified and the file count is greater than one, then the generation number will be added to the end of the generated filename, after a dot. Thus for example a pattern of "%t/java%g.log" with a count of 2 would typically cause log files to be written on Linux to /var/tmp/java0.log and /var/tmp/java1.log whereas on Windows they would be typically written to C:\TEMP\java0.log and C:\TEMP\java1.log

Generation numbers follow the sequence 0, 1, 2, etc.

Normally the "%u" unique field is set to 0. However, if the FileHandler tries to open the filename and finds the file is currently in use by another process it will increment the unique number field and try again. This will be repeated until FileHandler finds a file name that is not currently in use. If there is a conflict and no "%u" field has been specified, it will be added at the end of the filename after a dot. (This will be after any automatically added generation number.)

Thus if three processes were all trying to log to fred%.%g.txt then they might end up using fred0.0.txt, fred1.0.txt, fred2.0.txt as the first file in their rotating sequences.

Note that the use of unique ids to avoid conflicts is only guaranteed to work reliably when using a local disk file system.

Examples:

- /var/log/myapp/app.log
- /var/log/myapp/app_%g.log
- %t/app_%g.log

• %h/app_%g.log

```
(do
  ;; note: define the log filehandler just once at app startup!
  (io/log-filehandler "venice"
    "/var/log/myapp/venice_%g.log"
    16_000_000
    8)

  (io/log "venice" :info    "message 1")
  (io/log "venice" :warning "message 2")
  (io/log "venice" :severe  "message 3"))

(do
  (def tf (time/formatter "yyyy-MM-dd HH:mm:ss.SSS"))

  (defn formatter [log-record]
    (str/format "%s|%s|%s%s%n"
      (time/format (:timestamp log-record) tf)
      (:level log-record)
      (:message log-record)
      (:throwable log-record)))

  ;; note: define the log filehandler just once at app startup!
  (io/log-filehandler "venice"
    "/var/log/myapp/venice_%g.log"
    16_000_000
    8
    formatter)

  (io/log "venice" :info    "message 1")
  (io/log "venice" :warning "message 2")
  (io/log "venice" :severe  "message 3" (ex :VncException "test")))
```

SEE ALSO

[io/log](#)

Logs a message at a given level to a Java Util Logger (JUL).

[section](#)

[top](#)

io/mime-type

```
(io/mime-type file)
```

Returns the mime-type for the file if available else nil.

```
(io/mime-type "document.pdf")
=> "application/pdf"
```

```
(io/mime-type (io/file "document.pdf"))
=> "application/pdf"
```

[section](#)

[top](#)

io/mkdir

```
(io/mkdir dir)
```

Creates the directory. dir must be a file or a string (file path).

SEE ALSO

[io/mkdirs](#)

Creates the directory including any necessary but nonexistent parent directories. dir must be a file or a string (file path).

[section](#)

[top](#)

io/mkdirs

```
(io/mkdirs dir)
```

Creates the directory including any necessary but nonexistent parent directories. dir must be a file or a string (file path).

SEE ALSO

[io/mkdir](#)

Creates the directory. dir must be a file or a string (file path).

[section](#)

[top](#)

io/move-file

```
(io/move-file source target & options)
```

Moves source to target. Returns nil or throws a VncException. Source and target must be a file or a string (file path).

Options:

:replace true/false e.g.: if true replace an existing file, defaults to false

:atomic-move true/false e.g.: if true move the file as an atomic file system operation, defaults to false

SEE ALSO

[io/copy-file](#)

Copies source to dest. Returns nil or throws a VncException. Source must be a file or a string (file path), dest must be a file, a ...

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If f is a directory the directory must be empty. f ...

[io/touch-file](#)

Updates the lastModifiedTime of the file to the current time, or creates a new empty file if the file doesn't already exist. File must ...

[section](#)

[top](#)

io/move-files-glob

```
(io/move-files-glob src-dir dst-dir glob & options)
```

Move all files that match the glob pattern from a source to a destination directory. src-dir and dst-dir must be a file or a string (file path).

Options:

:replace true/false e.g.: if true replace an existing file, defaults to false
:atomic-move true/false e.g.: if true move the file as an atomic file system operation, defaults to false

Globbing patterns

<code>*.txt</code>	Matches a path that represents a file name ending in .txt
<code>*,*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with .txt or .xml
<code>foo.?[xy]</code>	Matches file names starting with foo. and a single character extension followed by a 'x' or 'y' character
<code>/home/*/*</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

Ranges

The pattern `[A-E]` would match any character that included ABCDE. Ranges can be used in conjunction with each other to make powerful patterns. Alphanumerical strings are matched by `[A-Za-z0-9]`. This would match the following:

- `[A-Z]` All uppercase letters from A to Z
- `[a-z]` All lowercase letters from a to z
- `[0-9]` All numbers from 0 to 9

Complementation

Globs can be used in complement with special characters that can change how the pattern works. The two complement characters are exclamation marks `(!)` and backslashes `(\\)`.

The exclamation mark can negate a pattern that it is put in front of. As `[CBR]at` matches Cat, Bat, or Rat the negated pattern `[!CBR]at` matches anything like Kat, Pat, or Vat.

Backslashes are used to remove the special meaning of single characters `'?'`, `'*'`, and `'['`, so that they can be used in patterns.

```
(io/move-files-glob "from" "to" "*.log")
```

SEE ALSO

[io/move-file](#)

Moves source to target. Returns nil or throws a VncException. Source and target must be a file or a string (file path).

[io/move-files-glob](#)

Move all files that match the glob pattern from a source to a destination directory. src-dir and dst-dir must be a file or a string (file path).

[io/copy-files-glob](#)

Copies all files that match the glob pattern from a source to a destination directory. src-dir and dst-dir must be a file or a string ...

[io/delete-files-glob](#)

Removes all files in a directory that match the glob pattern. dir must be a file or a string (file path).

[io/list-files-glob](#)

Lists all files in a directory that match the glob pattern. dir must be a file or a string (file path). Returns files as java.io.File

[section](#)

[top](#)

io/out-stream?

```
(io/out-stream? os)
```

Returns true if 'os' is a `java.io.OutputStream`

```
(try-with [os (io/bytebuf-out-stream)]
  (io/out-stream? os))
=> true
```

SEE ALSO

[io/in-stream?](#)

Returns true if 'is' is a `java.io.InputStream`

[section](#)

[top](#)

io/path?

```
(io/path f)
```

Returns true if f is a `java.nio.Path`.

```
(io/path? (io/->path "some.txt"))
=> true
```

SEE ALSO

[io/->path](#)

Converts to a `java.nio.Path`. f must be a file or a string (file path).

[section](#)

[top](#)

io/print

```
(io/print os s)
```

Prints a string s to an output stream. The output stream may be a `java.io.Writer` or a `java.io.PrintStream`!

[section](#)

[top](#)

io/print-line

```
(io/print-line os)
(io/print-line os s)
```

Prints a string s to an output stream. The output stream may be a `java.io.Writer` or a `java.io.PrintStream`!

[section](#)

[top](#)

io/read-char

```
(io/read-char is)
```

With arg reads the next char from the passed stream that must be a subclass of `java.io.Reader`.

Returns `nil` if the end of the stream is reached.

SEE ALSO

[io/read-line](#)

Reads the next line from the passed stream that must be a subclass of `:java.io.BufferedReader`.

[section](#)

[top](#)

io/read-line

```
(io/read-line is)
```

Reads the next line from the passed stream that must be a subclass of `:java.io.BufferedReader`.

Returns `nil` if the end of the stream is reached.

SEE ALSO

[io/read-char](#)

With `arg` reads the next char from the passed stream that must be a subclass of `:java.io.Reader`.

[section](#)

[top](#)

io/reader?

```
(io/reader? rd)
```

Returns true if `'rd'` is a `java.io.Reader`

```
(try-with [rd (io/string-reader "123")]
  (io/reader? rd))
=> true
```

SEE ALSO

[io/writer?](#)

Returns true if `'rd'` is a `java.io.Writer`

[section](#)

[top](#)

io/registered-watch-dirs

```
(io/registered-watch-dirs watcher)
```

Returns the registered watch directories of a watcher.

SEE ALSO

[io/watch-dir](#)

Watch a directory for changes, and call the function `event-fn` when it does. Calls the optional `failure-fn` if errors occur. On closing ...

[io/close-watcher](#)

Closes a watcher created from 'io/watch-dir'.

[section](#)[top](#)

io/slurp

(io/slurp f & options)

Reads the content of file f as text (string) or binary (bytebuf).

f may be a:

- string file path, e.g: "/temp/foo.json"
- bytebuffer
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.io.InputStream`
- `java.io.Reader`
- `java.nio.file.Path`
- `java.net.URL`
- `java.net.URI`

Returns a `bytebuf` or string depending on the passed `:binary` option.

Options:

`:binary true/false` e.g.: `:binary true`, defaults to false

`:encoding enc` e.g.: `:encoding :utf-8`, defaults to `:utf-8`

`io/slurp` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

Note: For HTTP and HTTPS downloads prefer to use `io/download`.

SEE ALSO

[io/slurp-lines](#)

Read all lines from f.

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` is. Supports the option `:binary` to either slurp binary or string data. For ...

[io/slurp-reader](#)

Slurps string data from a `java.io.Reader` rd.Note:

[io/spit](#)

Opens file f, writes content, and then closes f. f may be a file or a string (file path). The content may be a string or a bytebuf.

[io/download](#)

Downloads the content from the uri and reads it as text (string) or binary (bytebuf). Supports http and https protocols!

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)[top](#)

io/slurp-lines

(io/slurp-lines f & options)

Read all lines from f.

f may be a:

- string file path, e.g: `"/temp/foo.json"`
- `bytebuffer`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.io.InputStream`
- `java.io.Reader`
- `java.nio.file.Path`
- `java.net.URL`
- `java.net.URI`

Returns the a list of strings.

Options:

`:encoding enc` e.g.: `:encoding :utf-8`, defaults to `:utf-8`

`io/slurp-lines` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
(->> "1\n2\n3"
      io/string-in-stream
      io/slurp-lines)
=> ("1" "2" "3")
```

SEE ALSO

[str/split-lines](#)

Splits s into lines.

[io/slurp](#)

Reads the content of file f as text (string) or binary (bytebuf).

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` is. Supports the option `:binary` to either slurp binary or string data. For ...

[io/spit](#)

Opens file f, writes content, and then closes f. f may be a file or a string (file path). The content may be a string or a bytebuf.

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string s.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)

[top](#)

io/slurp-reader

`(io/slurp-reader rd)`

Slurps string data from a `java.io.Reader` rd.Note:

`io/slurp-reader` offers the same functionality as `io/slurp` but it opens more flexibility with sandbox configuration. `io/slurp` can be blacklisted to prevent reading data from the filesystem and still having `io/slurp-reader` for readers input available!

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (io/delete-file-on-exit file)
    (io/spit file "123456789" :append true)
    (try-with [rd (io/buffered-reader file :encoding :utf-8)]
```

```
(io/slurp-reader rd)))
)
=> "123456789"
```

SEE ALSO

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` is. Supports the option `:binary` to either slurp binary or string data. For ...

[io/slurp](#)

Reads the content of file `f` as text (string) or binary (bytebuf).

[io/slurp-lines](#)

Read all lines from `f`.

[io/spit](#)

Opens file `f`, writes content, and then closes `f`. `f` may be a file or a string (file path). The content may be a string or a bytebuf.

[io/uri-stream](#)

Returns a `java.io.InputStream` from the `uri`.

[io/file-in-stream](#)

Returns a `java.io.InputStream` for the file `f`.

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string `s`.

[io/bytebuf-in-stream](#)

Returns a `java.io.InputStream` from a bytebuf.

[section](#)[top](#)

io/slurp-stream

```
(io/slurp-stream is & options)
```

Slurps binary or string data from a `java.io.InputStream` is. Supports the option `:binary` to either slurp binary or string data. For string data an optional encoding can be specified.

Returns the result as a `bytebuf` or string depending on the passed `:binary` option.

Options:

`:binary true/false` e.g.: `:binary true`, defaults to `false`

`:encoding enc` e.g.: `:encoding :utf-8`, defaults to `:utf-8`

Note:

`io/slurp-stream` offers the same functionality as `io/slurp` but it opens more flexibility with sandbox configuration. `io/slurp` can be blacklisted to prevent reading data from the filesystem and still having `io/slurp-stream` for stream input available!

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (io/delete-file-on-exit file)
    (io/spit file "123456789" :append true)
    (try-with [is (io/file-in-stream file)]
      (io/slurp-stream is :binary false)))
  )
=> "123456789"
```

SEE ALSO

[io/slurp-reader](#)

Slurps string data from a `java.io.Reader` `rd`. Note:

[io/slurp](#)

Reads the content of file `f` as text (string) or binary (bytebuf).

[io/slurp-lines](#)

Read all lines from `f`.

[io/spit](#)

Opens file `f`, writes content, and then closes `f`. `f` may be a file or a string (file path). The content may be a string or a bytebuf.

[io/uri-stream](#)

Returns a `java.io.InputStream` from the uri.

[io/file-in-stream](#)

Returns a `java.io.InputStream` for the file `f`.

[io/string-in-stream](#)

Returns a `java.io.InputStream` for the string `s`.

[io/bytebuf-in-stream](#)

Returns a `java.io.InputStream` from a bytebuf.

[section](#)[top](#)

io/spit

```
(io/spit f content & options)
```

Opens file `f`, writes content, and then closes `f`. `f` may be a file or a string (file path). The content may be a string or a bytebuf.

Options:

:append true/false e.g.: `:append true`, defaults to false

:encoding enc e.g.: `:encoding utf-8`, defaults to `:utf-8`

`io/spit` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

SEE ALSO

[io/spit-stream](#)

Writes content (string or bytebuf) to the `java.io.OutputStream` `os`. If content is of type string an optional encoding (defaults to UTF-8) ...

[io/spit-writer](#)

Writes text to the `java.io.Writer` `wr`. The writer can optionally be flushed after the operation.

[io/slurp](#)

Reads the content of file `f` as text (string) or binary (bytebuf).

[io/slurp-lines](#)

Read all lines from `f`.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)[top](#)

io/spit-stream

```
(io/spit-stream os content & options)
```

Writes content (string or bytebuf) to the `java.io.OutputStream` `os`. If content is of type string an optional encoding (defaults to UTF-8) is supported. The stream can optionally be flushed after the operation.

Options:

:flush true/false e.g.: :flush true, defaults to false
:encoding enc e.g.: :encoding :utf-8, defaults to :utf-8

Note:

`io/spit-stream` offers the same functionality as `io/spit` but it opens more flexibility with sandbox configuration. `io/spit` can be blacklisted to prevent writing data to the filesystem and still having `io/spit-stream` for stream output available!

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (io/delete-file-on-exit file)
    (try-with [os (io/file-out-stream file)]
      (io/spit-stream os "123456789" :flush true))))
=> nil
```

SEE ALSO

[io/spit-writer](#)

Writes text to the `java.io.Writer` `wr`. The writer can optionally be flushed after the operation.

[io/spit](#)

Opens file `f`, writes content, and then closes `f`. `f` may be a file or a string (file path). The content may be a string or a bytebuf.

[section](#)

[top](#)

io/spit-writer

```
(io/spit-writer wr text)
```

Writes text to the `java.io.Writer` `wr`. The writer can optionally be flushed after the operation.

Options:

:flush true/false e.g.: :flush true, defaults to false

Note:

`io/spit-writer` offers the same functionality as `io/spit` but it opens more flexibility with sandbox configuration. `io/spit` can be blacklisted to prevent writing data to the filesystem and still having `io/spit-writer` for stream output available!

```
(do
  (let [file (io/temp-file "test-", ".txt")
        os (io/file-out-stream file)]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer os :encoding :utf-8)]
      (io/spit-writer wr "123456789" :flush true))))
=> nil
```

SEE ALSO

[io/spit-stream](#)

Writes content (string or bytebuf) to the `java.io.OutputStream` `os`. If content is of type string an optional encoding (defaults to UTF-8) ...

[io/spit](#)

Opens file `f`, writes content, and then closes `f`. `f` may be a file or a string (file path). The content may be a string or a bytebuf.

[section](#)

[top](#)

io/string-in-stream

```
(io/string-in-stream s & options)
```

Returns a `java.io.InputStream` for the string `s`.

Options:

`:encoding enc` e.g.: `:encoding :utf-8`, defaults to `:utf-8`

Note: The caller is responsible for closing the stream!

```
(let [text "The quick brown fox jumped over the lazy dog"]
  (try-with [is (io/string-in-stream text)]
    ; do something with is
  ))
```

SEE ALSO

[io/slurp-stream](#)

Slurps binary or string data from a `java.io.InputStream` `is`. Supports the option `:binary` to either slurp binary or string data. For ...

[io/file-in-stream](#)

Returns a `java.io.InputStream` for the file `f`.

[io/bytebuf-in-stream](#)

Returns a `java.io.InputStream` from a `bytebuf`.

[section](#)

[top](#)

io/string-reader

```
(io/string-reader s)
```

Creates a `java.io.StringReader` from a string.

Note: The caller is responsible for closing the reader!

```
(try-with [rd (io/string-reader "1234")]
  (println (read-char rd))
  (println (read-char rd))
  (println (read-char rd)))
1
2
3
=> nil

(let [rd (io/string-reader "1\n2\n3\n4")]
  (try-with [br (io/buffered-reader rd)]
    (println (read-line br))
    (println (read-line br))
    (println (read-line br))))
1
2
3
=> nil
```

SEE ALSO

[read-line](#)

Without arg reads the next line from the stream that is the current value of *in*. With arg reads the next line from the passed stream ...

[io/buffered-reader](#)

Create a java.io.Reader from f.

[io/string-writer](#)

Creates a java.io.StringWriter.

[section](#)[top](#)

io/string-writer

(io/string-writer)

Creates a `java.io.StringWriter` .

Dereferencing a string writer returns the captured string.

Note: The caller is responsible for closing the writer!

```
(try-with [sw (io/string-writer)]
  (print sw 100)
  (print sw "-")
  (print sw 200)
  (flush sw)
  (println @sw))
```

100-200

=> nil

SEE ALSO

[println](#)

Prints the values xs to the stream that is the current value of *out* or to the passed output stream os if given followed by a (newline).

[io/buffered-writer](#)

Creates a java.io.Writer for f.

[io/buffered-reader](#)

Create a java.io.Reader from f.

[section](#)[top](#)

io/symbolic-link?

(io/symbolic-link? f)

Returns true if the file f exists and is a symbolic link. f must be a file or a string (file path).

```
(io/symbolic-link? "/tmp/test.txt")
```

SEE ALSO

[io/file-hidden?](#)

Returns true if the file or directory f exists and is hidden. f must be a file or a string (file path).

[io/file-can-read?](#)

Returns true if the file or directory f exists and can be read. f must be a file or a string (file path).

[io/file-can-write?](#)

Returns true if the file or directory `f` exists and can be written. `f` must be a file or a string (file path).

[io/file-can-execute?](#)

Returns true if the file or directory `f` exists and can be executed. `f` must be a file or a string (file path).

[section](#)[top](#)

io/temp-dir

```
(io/temp-dir prefix)
```

Creates a new temp directory with prefix. Returns a `:java.io.File`.

```
(io/temp-dir "test-")
```

```
=> /var/folders/q0/gg9f6pqx5079cfvp9g5lqbzh0000gn/T/test-6782612499813408815
```

SEE ALSO

[with-tmp-dir](#)

Creates the sub directory 'subdir' in the tmp dir and runs the forms. After running the forms removes the created sub directory recursively.

[io/tmp-dir](#)

Returns the tmp dir as a `java.io.File`.

[io/temp-file](#)

Creates an empty temp file with the given prefix and suffix. Returns a `:java.io.File`.

[section](#)[top](#)

io/temp-file

```
(io/temp-file prefix suffix)
```

Creates an empty temp file with the given prefix and suffix. Returns a `:java.io.File`.

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (io/spit file "123456789" :append true)
    (io/slurp file :binary false :remove true))
)
```

```
=> "123456789"
```

SEE ALSO

[io/temp-dir](#)

Creates a new temp directory with prefix. Returns a `:java.io.File`.

[io/delete-file-on-exit](#)

Requests that the files or directories be deleted when the virtual machine terminates. Files (or directories) are deleted in the reverse ...

[section](#)[top](#)

io/tmp-dir

```
(io/tmp-dir)
```

Returns the tmp dir as a `java.io.File`.

```
(io/tmp-dir)
```

```
=> /var/folders/q0/gg9f6pqx5079cfvp9g5lqbzh0000gn/T
```

SEE ALSO

[io/user-dir](#)

Returns the user dir (current working dir) as a `java.io.File`.

[io/user-home-dir](#)

Returns the user's home dir as a `java.io.File`.

[io/temp-dir](#)

Creates a new temp directory with prefix. Returns a `java.io.File`.

[section](#)

[top](#)

io/touch-file

```
(io/touch-file file)
```

Updates the *lastModifiedTime* of the file to the current time, or creates a new empty file if the file doesn't already exist. File must be a file or a string (file path). Returns the file

SEE ALSO

[io/move-file](#)

Moves source to target. Returns nil or throws a `VncException`. Source and target must be a file or a string (file path).

[io/copy-file](#)

Copies source to dest. Returns nil or throws a `VncException`. Source must be a file or a string (file path), dest must be a file, a ...

[io/delete-file](#)

Deletes one or multiple files. Silently skips delete if the file does not exist. If *f* is a directory the directory must be empty. *f* ...

[section](#)

[top](#)

io/truncate-from-start-keep-lines

```
(io/truncate-from-start-keep-lines f max-size)
```

Truncates a text file to the given max size beginning at the start, honoring complete lines.

The ideal `cutoff` position is `file-size - max-size`. If there is no newline found after cutoff, the tail is a single (too long) line; to keep line integrity and size limit, keep nothing. If there is a newline found after cutoff, the effective cutoff will be the character position after the newline.

f must be a file or a string (file path).

```
(io/truncate-from-start-keep-lines "/tmp/test.txt" 1_000_000)
```

SEE ALSO

[io/file](#)

Returns a `java.io.File` from file path, or from a parent path and one or multiple children. The path and parent may be a file or a string ...

io/ungzip

```
(io/ungzip f)
```

ungzips f. f may be a file, a string (file path), a bytebuf, or an InputStream. Returns a bytebuf.

```
(-> (bytebuf-from-string "abcdef" :utf-8)
    (io/gzip)
    (io/ungzip))
=> [97 98 99 100 101 102]
```

SEE ALSO

[io/gzip](#)

gzips f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

[io/gzip?](#)

Returns true if f is a gzipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

[io/ungzip-to-stream](#)

ungzips a bytebuf returning an InputStream to read the deflated data from.

io/ungzip-to-stream

```
(io/ungzip-to-stream buf)
```

ungzips a bytebuf returning an InputStream to read the deflated data from.

```
(-> (bytebuf-from-string "abcdef" :utf-8)
    (io/gzip)
    (io/ungzip-to-stream)
    (io/slurp-stream :binary false :encoding :utf-8))
=> "abcdef"
```

SEE ALSO

[io/gzip](#)

gzips f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

io/unzip

```
(io/unzip f entry-name)
```

Unzips an entry from zip f the entry's data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abcdef" :utf-8))
    (io/unzip "a.txt"))
=> [97 98 99 100 101 102]
```

SEE ALSO

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/zip?](#)

Returns true if f is a zipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

[section](#)[top](#)

io/unzip-all

```
(io/unzip-all f)
(io/unzip-all glob f)
```

Unzips all entries of the zip f returning a map with the entry names as key and the entry data as bytebuf values. f may be a bytebuf, a file, a string (file path) or an InputStream.

An optional globbing pattern can be passed to filter the files to be unzipped.

Note: globbing patterns with unzip are always relative. E.g. `static/**/*.png`

Globbing patterns:

<code>*.txt</code>	Matches a path that represents a file name ending in .txt
<code>*.*</code>	Matches file names containing a dot
<code>*.{txt,xml}</code>	Matches file names ending with .txt or .xml
<code>foo.?</code>	Matches file names starting with foo. and a single character extension
<code>/home/*/*</code>	Matches <code>/home/gus/data</code> on UNIX platforms
<code>/home/**</code>	Matches <code>/home/gus</code> and <code>/home/gus/data</code> on UNIX platforms
<code>C:*</code>	Matches <code>C:\\foo</code> and <code>C:\\bar</code> on the Windows platform

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
          "b.txt" (bytebuf-from-string "def" :utf-8)
          "c.txt" (bytebuf-from-string "ghi" :utf-8))
  (io/unzip-all))
=> {"a.txt" [97 98 99] "b.txt" [100 101 102] "c.txt" [103 104 105]}
```

```
(->> (io/zip "foo/a.txt" (bytebuf-from-string "abc" :utf-8)
          "bar/b.txt" (bytebuf-from-string "def" :utf-8)
          "bar/c.log" (bytebuf-from-string "ghi" :utf-8))
  (io/unzip-all "bar/*.txt"))
=> {"bar/b.txt" [100 101 102]}
```

SEE ALSO

[io/unzip-to-dir](#)

Unzips the zip f to a directory. f may be a file, a string (file path), a bytebuf, or an InputStream.

[io/unzip-nth](#)

Unzips the nth (zero.based) entry of the zip f returning its data as a bytebuf. f may be a bytebuf, a file, a string (file path) or ...

[io/unzip-first](#)

Unzips the first entry of the zip f returning its data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/zip?](#)

Returns true if f is a zipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

io/unzip-first

```
(io/unzip-first zip)
```

Unzips the first entry of the zip *f* returning its data as a bytebuf. *f* may be a bytebuf, a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "b.txt" (bytebuf-from-string "def" :utf-8))
    (io/unzip-first))
=> [97 98 99]
```

SEE ALSO

[io/unzip-to-dir](#)

Unzips the zip *f* to a directory. *f* may be a file, a string (file path), a bytebuf, or an InputStream.

[io/unzip-nth](#)

Unzips the *nth* (zero.based) entry of the zip *f* returning its data as a bytebuf. *f* may be a bytebuf, a file, a string (file path) or ...

[io/unzip-all](#)

Unzips all entries of the zip *f* returning a map with the entry names as key and the entry data as bytebuf values. *f* may be a bytebuf, ...

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/zip?](#)

Returns true if *f* is a zipped file. *f* may be a file, a string (file path), a bytebuf, or an InputStream

io/unzip-nth

```
(io/unzip-nth zip n)
```

Unzips the *nth* (zero.based) entry of the zip *f* returning its data as a bytebuf. *f* may be a bytebuf, a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "b.txt" (bytebuf-from-string "def" :utf-8)
           "c.txt" (bytebuf-from-string "ghi" :utf-8))
    (io/unzip-nth 1))
=> [100 101 102]
```

SEE ALSO

[io/unzip-to-dir](#)

Unzips the zip *f* to a directory. *f* may be a file, a string (file path), a bytebuf, or an InputStream.

[io/unzip-first](#)

Unzips the first entry of the zip *f* returning its data as a bytebuf. *f* may be a bytebuf, a file, a string (file path) or an InputStream.

[io/unzip-all](#)

Unzips all entries of the zip *f* returning a map with the entry names as key and the entry data as bytebuf values. *f* may be a bytebuf, ...

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/zip?](#)

Returns true if f is a zipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

[section](#)

[top](#)

io/unzip-to-dir

```
(io/unzip-to-dir f dir)
```

Unzips the zip f to a directory. f may be a file, a string (file path), a bytebuf, or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
          "b.txt" (bytebuf-from-string "def" :utf-8)
          "c.txt" (bytebuf-from-string "ghi" :utf-8))
  (io/unzip-to-dir "."))
```

SEE ALSO

[io/unzip](#)

Unzips an entry from zip f the entry's data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

[io/unzip-nth](#)

Unzips the nth (zero.based) entry of the zip f returning its data as a bytebuf. f may be a bytebuf, a file, a string (file path) or ...

[io/unzip-first](#)

Unzips the first entry of the zip f returning its data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

[io/unzip-all](#)

Unzips all entries of the zip f returning a map with the entry names as key and the entry data as bytebuf values. f may be a bytebuf, ...

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/zip?](#)

Returns true if f is a zipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

[section](#)

[top](#)

io/uri-stream

```
(io/uri-stream uri)
```

Returns a `java.io.InputStream` from the uri.

Note: The caller is responsible for closing the stream!

```
(let [url "https://www.w3schools.com/xml/books.xml"]
  (try-with [is (io/uri-stream url)]
    (io/slurp-stream is :binary false :encoding :utf-8)))
```

SEE ALSO

[io/slurp-stream](#)

Slurps binary or string data from a java.io.InputStream is. Supports the option :binary to either slurp binary or string data. For ...

[section](#)

[top](#)

io/user-dir

(io/user-dir)

Returns the user dir (current working dir) as a java.io.File.

SEE ALSO

[io/tmp-dir](#)

Returns the tmp dir as a java.io.File.

[io/user-home-dir](#)

Returns the user's home dir as a java.io.File.

[section](#) [top](#)

io/user-home-dir

(io/user-home-dir)

Returns the user's home dir as a `java.io.File` .

SEE ALSO

[user-name](#)

Returns the logged-in's user name.

[io/user-dir](#)

Returns the user dir (current working dir) as a java.io.File.

[io/tmp-dir](#)

Returns the tmp dir as a java.io.File.

[section](#) [top](#)

io/watch-dir

(io/watch-dir dir & options)

Watch a directory for changes, and call the function `event-fn` when it does. Calls the optional `failure-fn` if errors occur. On closing the watcher `termination-fn` is called.

Options:

:include-all-subdirs true/false	If <code>true</code> includes in addition to the main dir recursively all subdirs. If <code>false</code> just watches on the main dir. Defaults to <code>false</code> .
:file-fn fn	A four argument function that receives the 'path', a dir? (true if path is a dir), a file? (true if path is a file), and an 'action' {:created, :deleted, :modified} of the changed file. Defaults to <code>nil</code> .
:error-fn fn	A two argument function called in error case that receives the watch path and the failure exception. Defaults to <code>nil</code> .
:termination-fn fn	A one argument function called when the watcher terminates. It receives the main watch dir. Defaults to <code>nil</code> .

Specific MacOS options (fswatch):

:fswatch-monitor m	An optional platform dependent monitor {:fsevents_monitor, :kqueue_monitor, :poll_monitor} to use with <code>fswatch</code> . Pass <code>nil</code> to let <code>fswatch</code> choose the optimal platform. Defaults to <code>nil</code> .

Run `fswatch --list-monitors` to get list of the available platform monitors.

`:fswatch-program p`

An optional path to the `fswatch` program, e.g.: `"fswatch"`, `"/opt/homebrew/bin/fswatch"`.

Defaults to `"/opt/homebrew/bin/fswatch"`.

Important

The *watcher* is a resource that **must be closed** with either `(io/close-watcher w)` or by using a `(try-with [w (io/watch-dir ... resource` protection statement!

Linux

On Linux the Java WatchService is used to watch dirs for file changes!

MacOS

The Java WatchService doesn't run properly on MacOS due to limitations in the Java WatchService implementation! As a workaround on MacOS the file watcher is implemented on top of the *fswatch* tool. The required *fswatch* tool can be installed from *Homebrew*.

```
$ brew install fswatch
```

Documentation:

- [fswatch Github](#)
- [fswatch Manual](#)
- [fswatch Installation](#)

```
(try-with [w (io/watch-dir "/tmp" #(println %1 %2))]  
  ;; wait 30s and terminate  
  (sleep 30 :seconds))  
  
(do  
  (def dir (io/temp-dir "watchdir-"))  
  (io/delete-file-on-exit dir)  
  
  (def lock 0)  
  
  (defn log [format & args]  
    (locking lock (println (apply str/format format args))))  
  
  (defn file-event [path dir? file? action]  
    (log "File:      %s %-9s, dir: %b, file: %b"  
      path action dir? file?))  
  
  (defn error-event [path exception]  
    (log "Failure:   %s %s" path (:message e)))  
  
  (defn termination-event [path]  
    (log "Terminated: %s" path))  
  
  (log "Watching:  %s" dir)  
  
  (try-with [w (io/watch-dir dir  
    :include-all-subdirs true  
    :file-fn          file-event  
    :error-fn         error-event  
    :termination-fn   termination-event)]  
    ;; on MacOS you might want to add the option  
    ;; :fswatch-program "/opt/homebrew/bin/fswatch"  
  
    (let [f (io/file dir "test1.txt")]  
      (io/touch-file f)           ;; created  
      (io/delete-file-on-exit f)  
      (sleep 1000)  
      (io/spit f "AAA" :append true)  ;; modified
```

```
(sleep 1000)
(io/delete-file f))                ;; deleted

;; wait that all events can be processed before the watcher is closed
(sleep 3000))

;; wait to receive the termination event
(sleep 1 :seconds))
```

SEE ALSO

[io/registered-watch-dirs](#)

Returns the registered watch directories of a watcher.

[io/close-watcher](#)

Closes a watcher created from 'io/watch-dir'.

[io/await-for](#)

Blocks the current thread until the file has been created, deleted, or modified according to the passed modes {:created, :deleted, ...

[section](#)

[top](#)

io/wrap-is-with-buffered-reader

```
(io/wrap-is-with-buffered-reader is encoding?)
```

Wraps an `java.io.InputStream` `is` with a `java.io.BufferedReader` using an optional encoding (defaults to :utf-8).

Note: The caller is responsible for closing the reader!

```
(let [data (bytebuf [108 105 110 101 32 49 10 108 105 110 101 32 50])]
  (try-with [is (io/bytebuf-in-stream data)
             rd (io/wrap-is-with-buffered-reader is :utf-8)]
    (println (read-line rd))
    (println (read-line rd))))
line 1
line 2
=> nil
```

SEE ALSO

[io/buffered-reader](#)

Create a `java.io.Reader` from `f`.

[section](#)

[top](#)

io/wrap-is-with-gzip-input-stream

```
(io/wrap-is-with-gzip-input-stream is)
```

Wraps a `:java.io.InputStream` `is` with a `:java.io.GZIPInputStream` to read compressed data in the GZIP format.

Note: The caller is responsible for closing the reader!

```
(let [text      "hello, hello, hello"
      gzip-buf  (-> (bytebuf-from-string text :utf-8)
                    (io/gzip))]
  (try-with [is (-> (io/bytebuf-in-stream gzip-buf)
```

```
        (io/wrap-is-with-gzip-input-stream))]]
(-> (io/slurp is :binary true)
    (bytebuf-to-string :utf-8))))
=> "hello, hello, hello"
```

SEE ALSO

[io/wrap-os-with-gzip-output-stream](#)

Wraps a `:java.io.OutputStream` is with a `:java.io.GZIPOutputStream` to write compressed data in the GZIP format.

[section](#)

[top](#)

io/wrap-is-with-inflater-input-stream

```
(io/wrap-is-with-inflater-input-stream is)
```

Wraps a `:java.io.InputStream` is with a `:java.io.InflaterInputStream` to read compressed data in the 'zlib' format.

Note: The caller is responsible for closing the reader!

```
(let [text      "hello, hello, hello"
      zlib-buf (-> (bytebuf-from-string text :utf-8)
                    (io/deflate))]
  (try-with [is (-> (io/bytebuf-in-stream zlib-buf)
                    (io/wrap-is-with-inflater-input-stream))]
    (-> (io/slurp is :binary true)
        (bytebuf-to-string :utf-8))))
=> "hello, hello, hello"
```

SEE ALSO

[io/wrap-os-with-deflater-output-stream](#)

Wraps a `:java.io.OutputStream` is with a `:java.io.DeflaterOutputStream` to write compressed data in the 'zlib' format.

[section](#)

[top](#)

io/wrap-os-with-buffered-writer

```
(io/wrap-os-with-buffered-writer os encoding?)
```

Wraps a `:java.io.OutputStream` os with a `:java.io.BufferedWriter` using an optional encoding (defaults to :utf-8).

Note: The caller is responsible for closing the writer!

```
(try-with [os (io/bytebuf-out-stream)
            wr (io/wrap-os-with-buffered-writer os :utf-8)]
  (println wr "100")
  (println wr "200")
  (flush wr)
  (bytebuf-to-string @os :utf-8))
=> "100\n200\n"
```

SEE ALSO

[io/wrap-os-with-print-writer](#)

Wraps an `:java.io.OutputStream` os with a `:java.io.PrintWriter` using an optional encoding (defaults to :utf-8).

io/wrap-os-with-deflater-output-stream

(io/wrap-os-with-deflater-output-stream is)

Wraps a `:java.io.OutputStream` is with a `:java.io.DeflaterOutputStream` to write compressed data in the 'zlib' format.

Note: The caller is responsible for closing the reader!

```
(let [text "hello, hello, hello"
      bos  (io/bytebuf-out-stream)]
  (try-with [gos (io/wrap-os-with-deflater-output-stream bos)]
    (io/spit gos text :encoding :utf-8)
    (io/flush gos)
    (io/close gos)
    (-> (io/inflate @bos)
        (bytebuf-to-string :utf-8))))
=> "hello, hello, hello"
```

SEE ALSO

[io/wrap-is-with-inflater-input-stream](#)

Wraps a `:java.io.InputStream` is with a `:java.io.InflaterInputStream` to read compressed data in the 'zlib' format.

io/wrap-os-with-gzip-output-stream

(io/wrap-os-with-gzip-output-stream is)

Wraps a `:java.io.OutputStream` is with a `:java.io.GZIPOutputStream` to write compressed data in the GZIP format.

Note: The caller is responsible for closing the reader!

```
(let [text "hello, hello, hello"
      bos  (io/bytebuf-out-stream)]
  (try-with [gos (io/wrap-os-with-gzip-output-stream bos)]
    (io/spit gos text :encoding :utf-8)
    (io/flush gos)
    (io/close gos)
    (-> (io/ungzip @bos)
        (bytebuf-to-string :utf-8))))
=> "hello, hello, hello"
```

SEE ALSO

[io/wrap-is-with-gzip-input-stream](#)

Wraps a `:java.io.InputStream` is with a `:java.io.GZIPInputStream` to read compressed data in the GZIP format.

io/wrap-os-with-print-writer

(io/wrap-os-with-print-writer os encoding?)

Wraps an `java.io.OutputStream` `os` with a `java.io.PrintWriter` using an optional encoding (defaults to `:utf-8`).

Note: The caller is responsible for closing the writer!

```
(let [os (io/bytebuf-out-stream)]
  (try-with [pr (io/wrap-os-with-print-writer os :utf-8)]
    (println pr "line 1")
    (println pr "line 2")
    (flush pr)
    @os))
=> [108 105 110 101 32 49 10 108 105 110 101 32 50 10]
```

SEE ALSO

[io/wrap-os-with-buffered-writer](#)

Wraps a `java.io.OutputStream` `os` with a `java.io.BufferedWriter` using an optional encoding (defaults to `:utf-8`).

[section](#)

[top](#)

io/writer?

```
(io/writer? rd)
```

Returns true if 'rd' is a `java.io.Writer`

```
(try-with [wr (io/string-writer)]
  (io/writer? wr))
=> true
```

SEE ALSO

[io/reader?](#)

Returns true if 'rd' is a `java.io.Reader`

[section](#)

[top](#)

io/zip

```
(io/zip & entries)
```

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string (file path), or an `InputStream`.

An entry name with a trailing '/' creates a directory. Returns the zip as bytebuf.

```
; single entry
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8))
  (io/spit "test.zip"))

; multiple entries
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
  "b.txt" (bytebuf-from-string "def" :utf-8)
  "c.txt" (bytebuf-from-string "ghi" :utf-8))
  (io/spit "test.zip"))
```

```

; multiple entries with subdirectories
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "x/b.txt" (bytebuf-from-string "def" :utf-8)
           "x/y/c.txt" (bytebuf-from-string "ghi" :utf-8))
      (io/spit "test.zip"))

; empty directory z/
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "z/" nil)
      (io/spit "test.zip"))

```

SEE ALSO

[io/zip-file](#)

Zips files and directories recursively. Does not zip hidden files and does not follow symbolic links. The zip-file may be a file, a ...

[io/unzip](#)

Unzips an entry from zip f the entry's data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

[io/gzip](#)

gzips f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

[io/spit](#)

Opens file f, writes content, and then closes f. f may be a file or a string (file path). The content may be a string or a bytebuf.

[io/zip-list](#)

List the content of a the zip f and prints it to the current value of out. f may be a bytebuf, a file, a string (file path), or an ...

[io/zip-list-entry-names](#)

Returns a list of the zip's entry names.

[io/zip-append](#)

Appends entries to an existing zip file f. Overwrites existing entries. An entry is given by a name and data. The entry data may be ...

[io/zip-remove](#)

Remove entries from a zip file f.

[section](#)

[top](#)

io/zip-append

```
(io/zip-append f & entries)
```

Appends entries to an existing zip file f. Overwrites existing entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string (file path), or an InputStream.

An entry name with a trailing '/' creates a directory.

```

(let [data (bytebuf-from-string "abc" :utf-8)]
  ; create the zip with a first file
  (->> (io/zip "a.txt" data)
        (io/spit "test.zip"))
  ; add text files
  (io/zip-append "test.zip" "b.txt" data "x/c.txt" data)
  ; add an empty directory
  (io/zip-append "test.zip" "x/y/" nil))

```

SEE ALSO

[io/zip-file](#)

Zips files and directories recursively. Does not zip hidden files and does not follow symbolic links. The zip-file may be a file, a ...

io/zip-remove

Remove entries from a zip file f.

[section](#)[top](#)

io/zip-file

(io/zip-file options* zip-file & files)

Zips files and directories recursively. Does not zip hidden files and does not follow symbolic links. The zip-file may be a file, a string (file path) or an OutputStream.

Options:

- :filter-fn fn a predicate function that filters the files to be added to the zip.
- :mapper-fn fn a mapper function that can map the file content of a file before it gets zipped. Returns nil or a :java.io.InputStream. The real file is used when nil is returned.
- :silent b if false prints the added entries to *out*, defaults to false

Example:

```
venice> (io/zip-file :silent false "test.zip" "dirA" "dirB")
```

Output:

```
adding: dirA/
adding: dirA/a1.png
adding: dirA/a2.png
adding: dirB/
adding: dirB/b1.png
```

```
; zip files
(io/zip-file "test.zip" "a.txt" "x/b.txt")

; zip all files from a directory
(io/zip-file "test.zip" "dir")

; zip all files in from two directories
(io/zip-file "test.zip" "dirA" "dirB")

; zip all files in from two directories and print the added entries
(io/zip-file :silent false "test.zip" "dirA" "dirB")

; zip all *.txt files from a directory
(io/zip-file :filter-fn (fn [dir name] (str/ends-with? name ".txt"))
  "test.zip"
  "dir")
```

SEE ALSO

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/zip-list](#)

List the content of a the zip f and prints it to the current value of out. f may be a bytebuf, a file, a string (file path), or an ...

[section](#)[top](#)

io/zip-list


```
(io/zip-list options* f)
```

List the content of a the zip *f* and prints it to the current value of *out*. *f* may be a bytebuf, a file, a string (file path), or an InputStream. Returns nil in print mode otherwise returns a list with attributes for each zip file entry.

Options:

:verbose b if true print verbose output, defaults to false
:print b if true print the entries to *out*, defaults to true

Example:

```
venice> (io/zip-list "test.zip")
  Length      Date/Time  Name
-----
    0   2021-01-05 10:32  dirA/
 309977  2021-01-05 10:32  dirA/a1.png
 309977  2021-01-05 10:32  dirA/a2.png
    0   2021-01-05 10:32  dirB/
 309977  2021-01-05 10:32  dirB/b1.png
-----
 929931                                5 files
=> nil

venice> (io/zip-list :verbose true "test.zip")
  Length  Method      Size  Cmpr      Date/Time    CRC-32  Name
-----
    0   Stored         0    0%   2021-01-05 10:32  00000000  dirA/
 309977  Defl:N    297691   4%   2021-01-05 10:32  C7F24B5C  dirA/a1.png
 309977  Defl:N    297691   4%   2021-01-05 10:32  C7F24B5C  dirA/a2.png
    0   Stored         0    0%   2021-01-05 10:32  00000000  dirB/
 309977  Defl:N    297691   4%   2021-01-05 10:32  C7F24B5C  dirB/b1.png
-----
 929931    null    893073   4%                                5 files
=> nil

venice> (io/zip-list :print false "test.zip")
=> ({:size 0 :method "Stored" :name "dirA/" ...} ...)
```

```
(io/zip-list "test-file.zip")
```

```
(io/zip-list :verbose true "test-file.zip")
```

SEE ALSO

[io/zip-list-entry-names](#)

Returns a list of the zip's entry names.

[io/zip-file](#)

Zips files and directories recursively. Does not zip hidden files and does not follow symbolic links. The zip-file my be a file, a ...

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/unzip](#)

Unzips an entry from zip *f* the entry's data as a bytebuf. *f* may be a bytebuf, a file, a string (file path) or an InputStream.

[section](#)

[top](#)

io/zip-list-entry-names

```
(io/zip-list-entry-names)
```

Returns a list of the zip's entry names.

```
(io/zip-list-entry-names "test-file.zip")
```

SEE ALSO

[io/zip-list](#)

List the content of a the zip `f` and prints it to the current value of `out`. `f` may be a bytebuf, a file, a string (file path), or an ...

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

[io/unzip](#)

Unzips an entry from zip `f` the entry's data as a bytebuf. `f` may be a bytebuf, a file, a string (file path) or an `InputStream`.

[section](#)

[top](#)

io/zip-remove

```
(io/zip-remove f & entry-names)
```

Remove entries from a zip file `f`.

```
; remove files from zip
(io/zip-remove "test.zip" "x/a.txt" "x/b.txt")
```

```
; remove directory from zip
(io/zip-remove "test.zip" "x/y/")
```

SEE ALSO

[io/zip-file](#)

Zips files and directories recursively. Does not zip hidden files and does not follow symbolic links. The zip-file may be a file, a ...

[io/zip-append](#)

Appends entries to an existing zip file `f`. Overwrites existing entries. An entry is given by a name and data. The entry data may be ...

[section](#)

[top](#)

io/zip?

```
(io/zip? f)
```

Returns true if `f` is a zipped file. `f` may be a file, a string (file path), a bytebuf, or an `InputStream`

```
(> (io/zip "a" (bytebuf-from-string "abc" :utf-8))
  (io/zip?))
=> true
```

SEE ALSO

[io/zip-file](#)

Zips files and directories recursively. Does not zip hidden files and does not follow symbolic links. The zip-file may be a file, a ...

[io/zip](#)

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string ...

ip-private?

```
(ip-private? addr)
```

Returns true if the IP address is private.

IPv4 addresses reserved for private networks:

- 192.168.0.0 - 192.168.255.255
- 172.16.0.0 - 172.31.255.255
- 10.0.0.0 - 10.255.255.255

```
(ip-private? "192.168.170.181")  
=> true
```

ipc/add-acl

```
(ipc/add-acl authenticator dest-type dest-name access principal)
```

Add ACL items for a destination and principal

Arguments:

authenticator	An authenticator
dest-type	A destination type { :queue , :topic , :function }
dest-name	A destination name
access	An access type { :offer , :poll , :offer-poll , :publish , :subscribe , :publish-subscribe , :execute , :deny }

ACL configurations:

queue offer	:queue -> one of{ :offer , :offer-poll , :deny }
queue poll	:queue -> one of{ :poll , :offer-poll , :deny }
topic subscribe	:topic -> one of{ :subscribe , :publish-subscribe , :deny }
topic publish	:topic -> one of{ :publish , :publish-subscribe , :deny }
function execute	:function -> one of{ :execute , :deny }

```
(let [auth (ipc/authenticator)]  
  (ipc/add-credentials auth "tom" "123")  
  (ipc/add-credentials auth "max" "456" :admin)  
  
  (ipc/add-acl auth :queue :queue/1 :offer "tom")  
  (ipc/add-acl auth :queue :queue/2 :poll "tom")  
  (ipc/add-acl auth :topic :topic/1 :publish "tom")  
  (ipc/add-acl auth :queue :topic/2 :subscribe "tom")  
  (ipc/add-acl auth :function :echo :execute "tom"))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/remove-acl](#)

Remove an ACL item for a destination and principal.

[ipc/default-acl](#)

Add default ACL items to override the system defaults. ACL default items are set per destination type.

[section](#)

[top](#)

ipc/add-cidr

```
(ipc/add-cidr authenticator cidr)
```

Adds a CIDR (Classless Inter-Domain Routing) block ACL.

```
(let [auth (ipc/authenticator)]
  (ipc/add-cidr auth "192.168.0.1/16"))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/remove-cidr](#)

Remove a CIDR (Classless Inter-Domain Routing) block ACL from an authenticator.

[ipc/clear-cidrs](#)

Removes all CIDR (Classless Inter-Domain Routing) block ACLs from an authenticator.

[section](#)

[top](#)

ipc/add-credentials

```
(ipc/add-credentials authenticator user-name password)
(ipc/add-credentials authenticator user-name password role)
```

Adds user credentials to an authenticator.

The only role currently support is the `:admin` role. The `:admin` is mandatory for clients to manage queues!

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "max" "456" :admin))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/remove-credentials](#)

Remove user credentials from an authenticator.

[ipc/clear-credentials](#)

Clears all user credentials from an authenticator.

[section](#)

[top](#)

ipc/authenticator

(ipc/authenticator)

Creates a new authenticator the manages the credential on behalf of the server.

The authenticator stores passwords as salted PBKDF2 hashes! It does not keep the clear text passwords.

(ipc/authenticator)

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "max" "456" :admin))
```

SEE ALSO

[ipc/load-authenticator](#)

Loads an authenticator from a file or an input stream.

[ipc/store-authenticator](#)

Stores an authenticator to a file or an output stream.

[ipc/add-credentials](#)

Adds user credentials to an authenticator.

[ipc/remove-credentials](#)

Remove user credentials from an authenticator.

[ipc/clear-credentials](#)

Clears all user credentials from an authenticator.

[ipc/add-acl](#)

Add ACL items for a destination and principal

[ipc/remove-acl](#)

Remove an ACL item for a destination and principal.

[section](#)

[top](#)

ipc/benchmark

(ipc/benchmark conn-uri msg-size duration & options)

Runs a benchmark.

Arguments:

conn-uri u	A connection URI - TCP/IP sockets<pre>af-inet://localhost:33333</pre> - Unix domain sockets (requires <code>unixsocket</code> libraries!)<pre>af-unix:///data/ipc/test.sock</pre>
msg-size n	The message payload size (payload is random data). The size can be specified as a number like <code>16384</code> or a number with a unit like <code>:16KB</code> or <code>:1MB</code> .
duration n	The duration in seconds.

Options:

:mode m	Operation mode <code>:client</code> , <code>:server</code> , or <code>:client-server</code> . Defaults to <code>:client-server</code>
:print b	If <code>true</code> print the result to stdout. Defaults to <code>false</code>
:oneway b	If <code>true</code> send oneway messages. Defaults to <code>false</code>
:encrypt b	If <code>true</code> encrypt the messages. Defaults to <code>false</code>
:connections n	The number of parallel connctions. Defaults to 1

`:socket-snd-buf-size n` The server socket's send buffer size.
Defaults to `-1` (use the sockets default buf size).
The size can be specified as a number like `64536` or a number with a unit like `:64KB` or `:1MB`.

`:socket-rcv-buf-size n` The server socket's receive buffer size.
Defaults to `-1` (use the sockets default buf size).
The size can be specified as a number like `64536` or a number with a unit like `:64KB` or `:1MB`.

`:ramp-up n` The ramp-up duration in seconds. Defaults to 0s.

Prints this statistics if the *print* option is enabled:

```
Messages:      79370
Payload size:   50 KB
Encryption:     off
-----
Duration:       5.0 s
Total bytes:    3875.5 MB
Throughput msgs: 15871 msg/s
Throughput bytes: 775 MB/s
```

With deactivated *print* option a map with detailed statistics values will be returned.

```
(ipc/benchmark "af-inet://localhost:33333"
  :5KB
  5
  :print true)

(ipc/benchmark "af-inet://localhost:33333"
  :5KB
  5
  :print true
  :ramp-up 1)

(ipc/benchmark "af-unix:///Users/foo/Desktop/venice/tmp/test.sock"
  :5KB
  5
  :socket-snd-buf-size :128KB
  :socket-rcv-buf-size :128KB
  :print true)
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[section](#) [top](#)

ipc/binary-message

```
(ipc/binary-message request-id topic mimetype data)
(ipc/binary-message request-id topic mimetype data durable)
(ipc/binary-message request-id topic mimetype data durable expires-at)
(ipc/binary-message request-id topic mimetype data durable expires-val expires-unit)
```

Creates a binary message.

Arguments:

`request-id r` A request ID (string, may be `nil`). May be used for idempotency checks by the receiver

topic t	A topic (string or keyword)
mimetype m	The mimetype of the payload data. A string like 'application/octet-stream', 'image/png'
data d	The message payload binary data (a bytebuf)
durable b	If <code>true</code> create a durable message if the server supports durable queues. Defaults to <code>false</code> .
expires-at t	Message expiration time in millis since epoch (may be <code>nil</code>)
expires-val v	Message expiration duration. E.g.: 2 (may be <code>nil</code>)
expires-unit u	Message expiration time unit. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(->> (ipc/binary-message "100" :test
                        "application/octet-stream"
                        (bytebuf [0 1 2 3 4 5 6 7])))

(ipc/message->map)
(println))

(->> (ipc/binary-message "100" "test"
                        "application/octet-stream"
                        (bytebuf [0 1 2 3 4 5 6 7])))

(ipc/message->map)
(println))

(->> (ipc/binary-message "100" "test"
                        "application/octet-stream"
                        (bytebuf [0 1 2 3 4 5 6 7])
                        false
                        (-> (time/local-date-time)
                           (time/plus :hours 2)
                           (time/to-millis))))

(ipc/message->map)
(println))

(->> (ipc/binary-message "100" "test"
                        "application/octet-stream"
                        (bytebuf [0 1 2 3 4 5 6 7])
                        false
                        2 :hours))

(ipc/message->map)
(println))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[ipc/oneway?](#)

Returns true if the message is one-way else false.

[ipc/response-ok?](#)

Returns true if the message response status is :OK else false.

[ipc/response-err?](#)

Returns true if the message has a response error status else false.

[section](#)[top](#)

ipc/clear-cidrs

```
(ipc/clear-cidrs authenticator)
```

Removes all CIDR (Classless Inter-Domain Routing) block ACLs from an authenticator.

```
(let [auth (ipc/authenticator)]
      (ipc/add-cidr auth "192.168.0.1/16")
      (ipc/clear-cidrs auth ))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/add-cidr](#)

Adds a CIDR (Classless Inter-Domain Routing) block ACL.

[ipc/remove-cidr](#)

Remove a CIDR (Classless Inter-Domain Routing) block ACL from an authenticator.

[section](#)[top](#)

ipc/clear-credentials

```
(ipc/clear-credentials authenticator)
```

Clears all user credentials from an authenticator.

```
(let [auth (ipc/authenticator)]
      (ipc/add-credentials auth "tom" "123")
      (ipc/add-credentials auth "max" "456" :admin)
      (ipc/clear-credentials auth))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/add-credentials](#)

Adds user credentials to an authenticator.

[ipc/remove-credentials](#)

Remove user credentials from an authenticator.

[section](#)[top](#)

ipc/client

```
(ipc/client port)
(ipc/client port & options)
(ipc/client host port & options)
(ipc/client conn-uri)
(ipc/client conn-uri & options)
```

Create a new client connecting to a server on the specified host and port.

Arguments:

port p	The server's TCP/IP port
host h	The server's TCP/IP host
conn-uri u	A connection URI • TCP/IP sockets <code>af-inet://localhost:33333</code> • Unix domain sockets (requires <code>unixsocket</code> libraries!) <code>af-unix:///data/ipc/test.sock</code>

Options:

:encrypt b	If <code>true</code> encrypt the payload data of all messages exchanged between this client and its associated server. The data is AES-256-GCM encrypted using a secret that is created and exchanged using the Diffie-Hellman key exchange algorithm.
:dh-rsa-client-key-pair k	The client RSA key pair for signing the Diffie-Hellman secret. Defaults to <code>nil</code> .
:dh-rsa-server-public-key k	The server RSA public key for verifying the Diffie-Hellman secret. Defaults to <code>nil</code> .
:user-name s	A user-name if the server requires authentication
:password s	A password if the server requires authentication
:socket-snd-buf-size n	The client socket's send buffer size. Defaults to <code>-1</code> (use the sockets default buf size). The size can be specified as a number like <code>64536</code> or a number with a unit like <code>:64KB</code> or <code>:1MB</code> .
:socket-rcv-buf-size n	The client socket's receive buffer size. Defaults to <code>-1</code> (use the sockets default buf size). The size can be specified as a number like <code>64536</code> or a number with a unit like <code>:64KB</code> or <code>:1MB</code> .

The client is thread-safe!

The client must be closed after use!

[See Inter-Process-Communication](#)

```
(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m))
    m)

  (defn send [client msg]
    (->> (ipc/send client :echo msg)
      (ipc/message->map)
      (println "RESPONSE: ")))

  (try-with [server (ipc/server 33333)
            client-1 (ipc/client 33333)
            client-2 (ipc/client "localhost" 33333)
            client-3 (ipc/client :localhost 33333)
            client-4 (ipc/client "af-inet://localhost:33333")]
    (ipc/create-function server :echo echo-handler)
    (send client-1 (ipc/plain-text-message "1" "test" "hello")))
```

```

(send client-2 (ipc/plain-text-message "2" "test" "hello"))
(send client-3 (ipc/plain-text-message "3" "test" "hello"))
(send client-4 (ipc/plain-text-message "4" "test" "hello")))

(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m)
      m))

  (defn send [client msg]
    (->> (ipc/send client :echo msg)
      (ipc/message->map)
      (println "RESPONSE: ")))

  (let [auth (ipc/authenticator)]
    (ipc/add-credentials auth "tom" "3-kio")
    (try-with [server (ipc/server 3333
                               :encrypt true
                               :authenticator auth)
              client (ipc/client "localhost" 3333
                               :user-name "tom"
                               :password "3-kio")]
      (ipc/create-function server :echo echo-handler)
      (send client (ipc/plain-text-message "1" "test" "hello")))))

```

SEE ALSO

[ipc/copy](#)

Copy a client with all its configuration, except 'user-name', 'password'.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[ipc/send](#)

Sends a message to the server the client is associated with.

[ipc/send-oneway](#)

Sends a one-way message to the server the client is associated with.

[ipc/publish](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/subscribe](#)

Subscribe to a topic.

[ipc/offer](#)

Offers a message to the named queue.

[ipc/poll](#)

Polls a message from the named queue.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a json string with optional pretty printing.

[ipc/benchmark](#)

Runs a benchmark.

[section](#)[top](#)

ipc/close

```
(ipc/close server)
(ipc/close client)
```

Closes a server or client

```
;; prefer try-with-resources to safely close server and client
(do
  (try-with [server (ipc/server 33333)
             client (ipc/client "localhost" 33333)]
    (println "Server running:" (ipc/running? server))
    (println "Client running:" (ipc/running? client))))

;; explicitly closing server and client
(do
  (let [server (ipc/server 33333)
        client (ipc/client "localhost" 33333)]
    (println "Server running:" (ipc/running? server))
    (println "Client running:" (ipc/running? client))
    (ipc/close client)
    (ipc/close server)))
```

SEE ALSO

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/running?](#)

Return true if the server or client is running else false

[section](#)[top](#)

ipc/copy

```
(ipc/copy client)
(ipc/copy client user-name password)
```

Copy a client with all its configuration, except 'user-name', 'password'.

```
(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m))
    m)

  (defn send [client msg])
```

```

(->> (ipc/send client :echo msg)
      (ipc/message->map)
      (println "RESPONSE: ")))

(try-with [server (ipc/server 3333)
          client-1 (ipc/client "localhost" 3333 :encrypted true)
          client-2 (ipc/copy client-1)
          client-3 (ipc/copy client-1)]
  (ipc/create-function server :echo echo-handler)
  (send client-1 (ipc/plain-text-message "1" "test" "hello 1"))
  (send client-2 (ipc/plain-text-message "2" "test" "hello 2"))
  (send client-3 (ipc/plain-text-message "3" "test" "hello 3"))))

(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m))
    m)

  (defn send [client msg]
    (->> (ipc/send client :echo msg)
          (ipc/message->map)
          (println "RESPONSE: ")))

  (let [auth (ipc/authenticator)]
    (ipc/add-credentials auth "tom" "3-kio")
    (ipc/add-credentials auth "max" "67-po")
    (ipc/add-credentials auth "jak" "99-af")
    (try-with [server (ipc/server 3333
                                :encrypt true
                                :authenticator auth)
              client-1 (ipc/client "localhost" 3333
                                :user-name "tom"
                                :password "3-kio")
              client-2 (ipc/copy client-1 "max" "67-po")
              client-3 (ipc/copy client-1 "jak" "99-af")]
      (ipc/create-function server :echo echo-handler)
      (send client-1 (ipc/plain-text-message "1" "test" "hello 1"))
      (send client-2 (ipc/plain-text-message "2" "test" "hello 2"))
      (send client-3 (ipc/plain-text-message "3" "test" "hello 3")))))

```

SEE ALSO

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[section](#)

[top](#)

ipc/create-function

```
(ipc/create-function server name func)
```

Creates a named function on the server.

A function name must only contain the characters 'a-z', 'A-Z', '0-9', '!', '_', '-', or '/'. Up to 100 characters are allowed.

Returns always `nil`.

```
(try-with [server (ipc/server 3333)
            client1 (ipc/client 3333)]
  (ipc/create-function server :echo (fn [m] m))
  ;;
  (ipc/exists-topic? server :echo))
```

SEE ALSO

[ipc/remove-function](#)

Removes a named topic.

[ipc/exists-function?](#)

Returns true if the named topic exists else false.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[section](#)

[top](#)

ipc/create-queue

```
(ipc/create-queue node name capacity)
(ipc/create-queue node name capacity type)
(ipc/create-queue node name capacity type persistence)
```

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered to exactly one client. 1 to N clients can *offer / poll* messages *from / to* the queue.

Queues live as long as the server lives if they are not durable.

A queue can be bounded or circular. Bounded queues block when offering new messages and the queue is full.

Circular queues never block but just keep the last 'capacity' messages. The oldest messages get discarded if the buffer is full and new messages are offered to the queue.

A queue name must only contain the characters 'a-z', 'A-Z', '0-9', '.', '_', '-', or '/'. Up to 100 characters are allowed.

Use `ipc/offer` to offer a new message to a queue.

Use `ipc/poll` to poll a message from a queue.

Returns always `nil` or throws an exception if the named queue already exists.

Arguments:

node s	A server or a client
name s	A queue name (string or keyword)
capacity n	The queue's capacity (max number of messages)
type t	Optional queue type <code>:bounded</code> or <code>:circular</code> . Defaults to <code>:bounded</code> .
persistence p	Optional queue persistence <code>:transient</code> or <code>:durable</code> . Defaults to <code>:transient</code> .

```
(try-with [server (ipc/server 3333)
            client1 (ipc/client 3333)
            client2 (ipc/client 3333)]
  (let [order (ipc/venice-message
                "1"
                :order
                {:item "espresso", :count 2})]
    (ipc/create-queue server :orders 100)
    (->> (ipc/message->json true order)
      (println "ORDER:"))
    (->> (ipc/offer client1 :orders 300 order)
      (ipc/message->json true)
      (println "OFFERED:"))
```

```
(->> (ipc/poll client2 :orders 300)
      (ipc/message->json true)
      (println "POLLED:"))))

(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123" :admin)
  (try-with [server (ipc/server 33333
                              :encrypt true
                              :authenticator auth)

            client (ipc/client 33333
                              :user-name "tom"
                              :password "123")]

    (ipc/create-queue client :orders 100)
    ;; ...
  ))
```

SEE ALSO

[ipc/create-temporary-queue](#)

Creates a named temporary queue on the server. Temporary queues live as long as the client, that created it, lives.

[ipc/remove-queue](#)

Removes a named queue.

[ipc/exists-queue?](#)

Returns true if the named queue exists else false.

[ipc/queue-status](#)

Returns a map with the queue status key - values.

[ipc/offer](#)

Offers a message to the named queue.

[ipc/poll](#)

Polls a message from the named queue.

[ipc/offer-async](#)

Offers a message to the named queue.

[ipc/poll-async](#)

Polls a message from the named queue.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[section](#)
[top](#)

ipc/create-temporary-queue

```
(ipc/create-temporary-queue client capacity)
```

Creates a named temporary queue on the server. Temporary queues live as long as the client, that created it, lives.

Venice can create a temporary queue dynamically for use as a dedicated reply queue for a client. You can use this to ensure that a reply message can be sent to the appropriate queue and reaches the desired client.

Returns the name of the created temporary queue.

Use `ipc/offer` to offer a new message to a temporary queue.

Use `ipc/poll` to poll a message from a temporary queue.

Arguments:

client c A client. Can only be called on behalf of a client!

capacity n The queue's capacity (max number of messages)

```
(do
  (try-with [server (ipc/server 33333)
             client1 (ipc/client 33333)
             client2 (ipc/client 33333)]

    (ipc/create-queue server :orders 100)

    (let [confirm-queue (ipc/create-temporary-queue client1 100)]
      ;; client1 sends an order to the order queue
      (ipc/offer client1 :orders confirm-queue 300
        (ipc/venice-message
          "1"
          :order
          {:item "espresso", :count 2}))

      ;; client2 receives order from order queue and replies to the reply-to queue
      (let [order (ipc/poll client2 :orders 300)
            request-id (ipc/message-field order :request-id)
            reply-to-queue (ipc/message-field order :reply-to-queue-name)
            order-data (ipc/message-field order :payload-venice)
            reply-message (ipc/venice-message request-id "confirmed" order-data)]
        (ipc/offer client2 reply-to-queue 1_000 reply-message))

      ;; client1 receives confirmation
      (println (ipc/poll client2 confirm-queue 300))))))
```

SEE ALSO

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/remove-queue](#)

Removes a named queue.

[ipc/exists-queue?](#)

Returns true if the named queue exists else false.

[ipc/queue-status](#)

Returns a map with the queue status key - values.

[ipc/offer](#)

Offers a message to the named queue.

[ipc/poll](#)

Polls a message from the named queue.

[ipc/offer-async](#)

Offers a message to the named queue.

[ipc/poll-async](#)

Polls a message from the named queue.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

ipc/create-topic

```
(ipc/create-topic node name)
```

Creates a named topic on the server.

A topic name must only contain the characters 'a-z', 'A-Z', '0-9', '.', '_', '-', or '/'. Up to 100 characters are allowed.

Returns always `nil` or throws an exception if the named topic already exists.

Arguments:

node s A server or a client

name s A topic name (string or keyword)

```
(try-with [server (ipc/server 33333)
            client1 (ipc/client 33333)]
  (ipc/create-topic server :orders-closed)
  ;;
  (ipc/exists-topic? server :orders-closed))
```

SEE ALSO

[ipc/remove-topic](#)

Removes a named topic.

[ipc/exists-topic?](#)

Returns true if the named topic exists else false.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

section

top

ipc/default-acl

```
(ipc/default-acl authenticator dest-type access)
```

Add default ACL items to override the system defaults. ACL default items are set per destination type.

Arguments:

authenticator An authenticator

dest-type A destination type { `:queue`, `:topic`, `:function` }

| access | An access type { `:offer`, `:poll`, `:offer-poll`, `:publish`, `:subscribe`, `:publish-subscribe`, `:execute`, `:deny` } | *System default ACLs:*

queues `:offer-poll`

topics `:publish-subscribe`

functions `:execute`

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "max" "456" :admin)
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "jak" "123"))
```



```
(ipc/add-default-acl auth :queue :poll)
(ipc/add-default-acl auth :topic :deny)
(ipc/add-default-acl auth :function :deny)

(ipc/add-acl auth :queue :queue/1 :offer-poll "tom")
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/add-acl](#)

Add ACL items for a destination and principal

[ipc/remove-acl](#)

Remove an ACL item for a destination and principal.

[section](#)

[top](#)

ipc/exists-function?

```
(ipc/exists-function? node name)
```

Returns `true` if the named topic exists else `false` .

```
(try-with [server (ipc/server 33333)]
  (ipc/create-function server :echo (fn [m] m))
  ;; ...
  (ipc/exists-function? server :echo))
```

SEE ALSO

[ipc/create-function](#)

Creates a named function on the server.

[ipc/remove-function](#)

Removes a named topic.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[section](#)

[top](#)

ipc/exists-queue?

```
(ipc/exists-queue? node name)
```

Returns `true` if the named queue exists else `false` .

Arguments:

node n A server or client

name n A queue name (string or keyword)

```
(try-with [server (ipc/server 33333)]
  (ipc/create-queue server :orders 100)
  ;; ...
  (ipc/exists-queue? server :orders))
```

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123" :admin)
  (try-with [server (ipc/server 33333
                             :encrypt true
                             :authenticator auth)
             client (ipc/client 33333
                               :user-name "tom"
                               :password "123")]
    (ipc/create-queue client :orders 100)
    ;; ...
    (ipc/exists-queue? client :orders)))
```

SEE ALSO

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/create-temporary-queue](#)

Creates a named temporary queue on the server. Temporary queues live as long as the client, that created it, lives.

[ipc/remove-queue](#)

Removes a named queue.

[ipc/queue-status](#)

Returns a map with the queue status key - values.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[section](#)

[top](#)

ipc/exists-topic?

```
(ipc/exists-topic? node name)
```

Returns `true` if the named topic exists else `false`.

Arguments:

node n A server or client

name n A topic name (string or keyword)

```
(try-with [server (ipc/server 33333)]
  (ipc/create-topic server :orders-closed)
  ;; ...
  (ipc/exists-topic? server :orders-closed))
```

SEE ALSO

[ipc/create-topic](#)

Creates a named topic on the server.

[ipc/remove-topic](#)

Removes a named topic.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[section](#)

[top](#)

ipc/load-authenticator

(ipc/load-authenticator source)

Loads an authenticator from a file or an input stream.

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "max" "456")
  (ipc/store-authenticator auth (io/file "./ipc.cred")))
  (let [b (ipc/load-authenticator (io/file "./ipc.cred"))]
    ;; ...
  ))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/store-authenticator](#)

Stores an authenticator to a file or an output stream.

[section](#)

[top](#)

ipc/message->json

(ipc/message->json message)
(ipc/message->json pretty message)

Converts message to a Json string with optional pretty printing.

Returns a Json string.

```
(->> (ipc/text-message "1" "test"
  "text/plain" :UTF-8 "hello")
  (ipc/message->json true))

(->> (ipc/venice-message "1" "test"
  {:a 100, :b 200})
  (ipc/message->json true))

(->> (ipc/binary-message "1" "test"
  "application/octet-stream"
  (bytebuf [0 1 2 3 4]))
  (ipc/message->json true))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message-field](#)

Returns a specific field from the message.

[ipc/message-expired?](#)

Returns true the message has expired else false.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/message->map

```
(ipc/message->map message)
```

Converts a message to a Venice map.

Returns a Venice map with the keys:

- `:type`
- `:status`
- `:timestamp`
- `:expires-at`
- `:oneway?`
- `:durable?`
- `:request-id`
- `:topic`
- `:mimetype`
- `:charset`
- `:text` (only set if there is a message charset defined)
- `:data`

```
(->> (ipc/text-message "1" "test"
  "text/plain" :UTF-8 "hello")
  (ipc/message->map))
```

```
(->> (ipc/venice-message "1" "test"
  {:a 100, :b 200})
  (ipc/message->map))
```

```
(->> (ipc/binary-message "1" "test"
  "application/octet-stream"
  (bytebuf [0 1 2 3 4]))
  (ipc/message->map))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message-field](#)

Returns a specific field from the message.

[ipc/message-expired?](#)

Returns true the message has expired else false.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[section](#)

[top](#)

ipc/message-expired?

```
(ipc/message-expired? message)
```

Returns `true` the message has expired else `false` .

```
(let [m (ipc/text-message "100"
                          "test"
                          "text/plain"
                          :UTF-8
                          "Hello!")]
      (println (ipc/message-expired? m)))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/message-field](#)

Returns a specific field from the message.

[ipc/message->map](#)

Converts a message to a Venice map.

ipc/message-field

(ipc/message-field message field)

Returns a specific field from the message.

Message	Originator
ID	send, publish/subscribe method
Message Type	send, publish/subscribe method
Oneway	client or framework method
Durable	client or framework method
Response Status	server response processor
Timestamp	message creator
ExpiresAt	client (may be null)
Request ID	client (may be used for idempotency checks the receiver)
Subject	client
Destination Name	client
ReplyTo Queue Name	client (offer/poll, may be null)
Payload Mimetype	client
Payload Charset	client if payload data is a string else null
Payload data	client

Supported field names:

- `:id` - the message's technical ID
- `:type` - the message type (request, response, ..)
- `:oneway?` - `true` if one-way message else `false`
- `:durable?` - `true` if durable message else `false`
- `:response-status` - the response status (ok, bad request, ...)
- `:timestamp` - the message's creation timestamp in milliseconds since epoch
- `:expires-at` - the message's expiry timestamp in milliseconds since epoch (may be nil)
- `:request-id` - the request ID (may be nil)
- `:subject` - the subject
- `:destination-name` - the destination name (queue/topic/function name)
- `:reply-to-queue-name` - the reply to queue name
- `:payload-mimetype` - the payload data mimetype
- `:payload-charset` - the payload data charset (if payload is a text form)
- `:payload-text` - the payload converted to text data if payload is textual data else error

- `:payload-binary` - the payload binary data (the raw message binary data)
- `:payload-venice` - the payload converted venice data if mimetype is 'application/json' else error

Message type:

- `:REQUEST` - a request message
- `:PUBLISH` - a publish message
- `:SUBSCRIBE` - a subscribe message
- `:UNSUBSCRIBE` - an unsubscribe message
- `:OFFER` - an offer message for a queue
- `:POLL` - a poll message from a queue
- `:CREATE_QUEUE` - a queue create request message
- `:CREATE_TEMP_QUEUE` - a temporary queue create request message
- `:REMOVE_QUEUE` - a queue remove request message
- `:STATUS_QUEUE` - a queue status request message
- `:RESPONSE` - a response to a request message
- `:NULL` - a message with yet undefined type

Response status:

- `:OK` - a response message for a successfully processed request
- `:SERVER_ERROR` - a response indicating a server side error while processing the request
- `:BAD_REQUEST` - invalid request
- `:HANDLER_ERROR` - a server handler error in the server's request processing
- `:QUEUE_NOT_FOUND` - the required queue does not exist
- `:QUEUE_EMPTY` - the addressed queue in a poll request is empty
- `:QUEUE_FULL` - the addressed queue in offer request is full
- `:NULL` - a message with yet undefined status

```
(let [m (ipc/text-message "100"
  "test"
  "text/plain"
  :UTF-8
  "Hello!")]
  (println (ipc/message-field m :id))
  (println (ipc/message-field m :type))
  (println (ipc/message-field m :oneway?))
  (println (ipc/message-field m :durable?))
  (println (ipc/message-field m :timestamp))
  (println (ipc/message-field m :expires-at))
  (println (ipc/message-field m :response-status))
  (println (ipc/message-field m :request-id))
  (println (ipc/message-field m :subject))
  (println (ipc/message-field m :payload-mimetype))
  (println (ipc/message-field m :payload-charset))
  (println (ipc/message-field m :payload-text))
  (println (ipc/message-field m :payload-binary)))

(let [m (ipc/binary-message "100"
  "test"
  "application/octet-stream"
  (bytebuf [0 1 2 3 4 5 6 7]))]
  (println (ipc/message-field m :id))
  (println (ipc/message-field m :type))
  (println (ipc/message-field m :oneway?))
  (println (ipc/message-field m :durable?))
  (println (ipc/message-field m :timestamp)))
```

```

(println (ipc/message-field m :expires-at))
(println (ipc/message-field m :response-status))
(println (ipc/message-field m :request-id))
(println (ipc/message-field m :subject))
(println (ipc/message-field m :payload-mimetype))
(println (ipc/message-field m :payload-charset))
(println (ipc/message-field m :payload-binary))

(let [m (ipc/venice-message "100" "test"
                           {:a 100, :b 200})]
  (println (ipc/message-field m :id))
  (println (ipc/message-field m :type))
  (println (ipc/message-field m :oneway?))
  (println (ipc/message-field m :durable?))
  (println (ipc/message-field m :timestamp))
  (println (ipc/message-field m :expires-at))
  (println (ipc/message-field m :response-status))
  (println (ipc/message-field m :request-id))
  (println (ipc/message-field m :subject))
  (println (ipc/message-field m :payload-mimetype))
  (println (ipc/message-field m :payload-charset))
  (println (ipc/message-field m :payload-venice)))

```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/message-expired?](#)

Returns true the message has expired else false.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[section](#)

[top](#)

ipc/message-size

```
(ipc/message-size message)
```

Calculates the effective message size (not compressed, not encrypted).

Returns a map with the keys:

- `header` - header size in bytes
- `payload-meta` - payload meta data size in bytes
- `payload-data` - payload data size in bytes
- `total` - total size in bytes


```
(->> (ipc/plain-text-message "1" "test" "hello")
      (ipc/message-size))
```

SEE ALSO

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/offer

```
(ipc/offer client queue-name queue-offer-timeout message)
(ipc/offer client queue-name reply-to-queue-name queue-offer-timeout message)
```

Offers a message to the named queue.

Returns the server's response message.

Arguments:

client c	A client to send the offer message from
queue-name q	A queue name (string or keyword) to offer the message to
reply-to-queue-name q	An optional reply-to queue name (string or keyword) where replies are sent to
queue-offer-timeout t	The maximum time in milliseconds the server waits offering the message to the queue. A timeout of -1 means wait as long as it takes.
message m	The offer request message

The server returns a response message with one of these status:

- `:OK` - message added to the queue
- `:SERVER_ERROR` - indicates a server while offering the message to the queue
- `:BAD_REQUEST` - invalid request, details in the payload
- `:QUEUE_NOT_FOUND` - the queue does not exist
- `:QUEUE_FULL` - the queue is full, offer rejected

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (try-with [server (ipc/server 33333)
             client1 (ipc/client 33333)
             client2 (ipc/client 33333)]
    ;; create a queue to allow client1 and client2 to exchange messages
    (ipc/create-queue server :orders 100)

    ;; client1 offers order Venice data message to the queue
    ;; requestId="1" and "2", subject="order", payload={:item "espresso", :count 2}
    (let [order (ipc/venice-message "1" "order" {:item "espresso", :count 2})])
```

```
(println "ORDER:" (ipc/message->json true order))

;; publish the order
(->> (ipc/offer client1 :orders 300 order)
  (ipc/message->json true)
  (println "OFFERED:"))

;; client2 pulls next order from the queue
(->> (ipc/poll client2 :orders 300)
  (ipc/message->json true)
  (println "POLLED:"))))
```

SEE ALSO

[ipc/offer-async](#)

Offers a message to the named queue.

[ipc/poll](#)

Polls a message from the named queue.

[ipc/poll-async](#)

Polls a message from the named queue.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/remove-queue](#)

Removes a named queue.

[section](#)

[top](#)

ipc/offer-async

```
(ipc/offer client queue-name queue-offer-timeout message)
(ipc/offer client queue-name reply-to-queue-name queue-offer-timeout message)
```

Offers a message to the named queue.

Returns a future with the server's response message.

Arguments:

client c	A client to send the offer message from
queue-name q	A queue name (string or keyword) to offer the message to
reply-to-queue-name q	An optional reply-to queue name (string or keyword) where replies are sent to
queue-offer-timeout t	The maximum time in milliseconds the server waits offering the message to the queue.

A timeout of -1 means wait as long as it takes.

message m The offer request message

The server returns a response message with one of these status:

- `:OK` - message added to the queue
- `:SERVER_ERROR` - indicates a server while offering the message to the queue
- `:BAD_REQUEST` - invalid request, details in the payload
- `:QUEUE_NOT_FOUND` - the queue does not exist
- `:QUEUE_FULL` - the queue is full, offer rejected

```
(do
;; thread-safe printing
(defn println [& msg] (locking println (apply core/println msg)))

(try-with [server (ipc/server 33333)
           client1 (ipc/client 33333)
           client2 (ipc/client 33333)]
;; create a queue to allow client1 and client2 to exchange messages
(ipc/create-queue server :orders 100)

;; client1 offers order Venice data message to the queue
;; requestId="1" and "2", subject="order", payload={:item "espresso", :count 2}
(let [order (ipc/venice-message "1" "order" {:item "espresso", :count 2})]
  (println "ORDER:" (ipc/message->json true order))

  ;; publish the order
  (-<> (ipc/offer-async client1 :orders 300 order)
    (deref <> 1_000 :timeout)
    (ipc/message->json true <>)
    (println "OFFERED:" <>)))

;; client2 pulls next order from the queue
(-<> (ipc/poll-async client2 :orders 300)
  (deref <> 1_000 :timeout)
  (ipc/message->json true <>)
  (println "POLLED:" <>)))
```

SEE ALSO

[ipc/offer](#)

Offers a message to the named queue.

[ipc/poll](#)

Polls a message from the named queue.

[ipc/poll-async](#)

Polls a message from the named queue.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/remove-queue](#)

Removes a named queue.

[section](#)

[top](#)

ipc/oneway?

```
(ipc/oneway? message)
```

Returns `true` if the message is one-way else `false`.

Note: the oneway flag on the message is delayed until a message is sent from the client to the server or vice versa.

```
(do
  (defn echo-handler [m]
    (if (ipc/oneway? m) nil m))

  (try-with [server (ipc/server 33333)
            client (ipc/client "localhost" 33333)]
    (ipc/create-function server :echo echo-handler)
    (->> (ipc/plain-text-message "1" "test" "hello")
      (ipc/send client :echo)
      (ipc/message->map)
      (println))))
```

SEE ALSO

[ipc/response-ok?](#)

Returns true if the message response status is :OK else false.

[ipc/response-err?](#)

Returns true if the message has a response error status else false.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[section](#)

[top](#)

ipc/plain-text-message

```
(ipc/plain-text-message request-id topic text)
(ipc/plain-text-message request-id topic text durable)
(ipc/plain-text-message request-id topic text durable expires-at)
(ipc/plain-text-message request-id topic text durable expires-val expires-unit)
```

Creates a plain text message with mimetype `text/plain` and charset `:UTF-8`.

Arguments:

request-id r	A request ID (string, may be <code>nil</code>). May be used for idempotency checks by the receiver
topic t	A topic (string or keyword)
text t	The message payload text (a string)
durable b	If <code>true</code> create a durable message if the server supports durable queues. Defaults to <code>false</code> .
expires-at t	Message expiration time in millis since epoch (may be <code>nil</code>)
expires-val v	Message expiration duration. E.g.: 2 (may be <code>nil</code>)
expires-unit u	Message expiration time unit. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(->> (ipc/plain-text-message "1" :test "hello")
      (ipc/message->map)
      (println))

(->> (ipc/plain-text-message "1" "test" "hello")
      (ipc/message->map)
      (println))

(->> (ipc/plain-text-message "100" "test" "hello"
                              false
                              2 :hours)
      (ipc/message->map)
      (println))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[ipc/oneway?](#)

Returns true if the message is one-way else false.

[ipc/response-ok?](#)

Returns true if the message response status is :OK else false.

[ipc/response-err?](#)

Returns true if the message has a response error status else false.

ipc/poll

```
(ipc/poll client queue-name queue-poll-timeout)
```

Polls a message from the named queue.

Returns the server's response message.

Arguments:

client c	A client to send the poll message from
queue-name q	A queue name (string or keyword) to poll the message to
queue-poll-timeout t	The maximum time in milliseconds the server waits to poll a the message from the queue. A timeout of -1 means wait as long as it takes.
message m	The poll request message

The server returns a response message with one of these status:

- `:OK` - message successfully polled from the queue, response holds the data
- `:SERVER_ERROR` - indicates a server while polling a message from the queue
- `:BAD_REQUEST` - invalid request, details in the payload
- `:QUEUE_NOT_FOUND` - the queue does not exist
- `:QUEUE_EMPTY` - the queue is empty

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (try-with [server (ipc/server 3333)
             client1 (ipc/client 3333)
             client2 (ipc/client 3333)]
    ;; create a queue to allow client1 and client2 to exchange messages
    (ipc/create-queue server :orders 100)

    ;; client1 offers order Venice data message to the queue
    ;; requestId="1" and "2", subject="order", payload={:item "espresso", :count 2}
    (let [order (ipc/venice-message "1" "order" {:item "espresso", :count 2})]
      (println "ORDER:" (ipc/message->json true order))

      ;; publish the order
      (->> (ipc/offer client1 :orders 300 order)
        (ipc/message->json true)
        (println "OFFERED:"))

      ;; client2 pulls next order from the queue
      (->> (ipc/poll client2 :orders 300)
        (ipc/message->json true)
        (println "POLLED:"))))
```

SEE ALSO

[ipc/poll-async](#)

Polls a message from the named queue.

[ipc/offer](#)

Offers a message to the named queue.

[ipc/offer-async](#)

Offers a message to the named queue.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/remove-queue](#)

Removes a named queue.

[section](#)

[top](#)

ipc/poll-async

```
(ipc/poll-async client queue-name, queue-poll-timeout)
```

Polls a message from the named queue.

Returns a future with the server's response message.

Arguments:

client c	A client to send the poll message from
queue-name q	A queue name (string or keyword) to poll the message to
queue-poll-timeout t	The maximum time in milliseconds the server waits to poll a the message from the queue. A timeout of -1 means wait as long as it takes.
message m	The poll request message

The server returns a response message with one of these status:

- `:OK` - message successfully polled from the queue, response holds the data
- `:SERVER_ERROR` - indicates a server while polling a message from the queue
- `:BAD_REQUEST` - invalid request, details in the payload
- `:QUEUE_NOT_FOUND` - the queue does not exist
- `:QUEUE_EMPTY` - the queue is empty

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (try-with [server (ipc/server 3333)
             client1 (ipc/client 3333)
             client2 (ipc/client 3333)]
    ;; create a queue to allow client1 and client2 to exchange messages
    (ipc/create-queue server :orders 100)

    ;; client1 offers order Venice data message to the queue
```

```
;; requestId="1" and "2", subject="order", payload={:item "espresso", :count 2}
(let [order (ipc/venice-message "1" "order" {:item "espresso", :count 2})]
  (println "ORDER:" (ipc/message->json true order))

  ;; publish the order
  (-<> (ipc/offer-async client1 :orders 300 order)
    (deref <> 1_000 :timeout)
    (ipc/message->json true <>)
    (println "OFFERED:" <>)))

;; client2 pulls next order from the queue
(-<> (ipc/poll-async client2 :orders 300)
  (deref <> 1_000 :timeout)
  (ipc/message->json true <>)
  (println "POLLED:" <>)))
```

SEE ALSO

[ipc/poll](#)

Polls a message from the named queue.

[ipc/offer-async](#)

Offers a message to the named queue.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/remove-queue](#)

Removes a named queue.

[section](#)

[top](#)

ipc/publish

```
(ipc/publish client topic-name message)
```

Publishes a messages to all clients that have subscribed to the message's topic.

Returns the server's response message.

The response message has one of these status:

- `:OK` - message successfully published
- `:SERVER_ERROR` - indicates a server side error while processing the request
- `:BAD_REQUEST` - invalid request, details in the payload

Note: a client in subscription mode can not send or publish messages!

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (defn client-subscribe-handler [m]
    (println "SUBSCRIBED:" (ipc/message->json true m)))

  (try-with [server (ipc/server 33333)
            client1 (ipc/client "localhost" 33333)
            client2 (ipc/client "localhost" 33333)]

    ;; create topic :test
    (ipc/create-topic server :test)

    ;; client1 subscribes to messages with topic :test
    (ipc/subscribe client1 :test client-subscribe-handler)

    ;; client2 publishes a plain text message:
    ;;  requestId="1", subject=:testing, payload="hello"
    (let [m (ipc/plain-text-message "1" :testing "hello")]
      (println "PUBLISHING:" (ipc/message->json true m))
      (->> (ipc/publish client2 :test m)
            (ipc/message->json true)
            (println "PUBLISHED:"))

      (sleep 300)))
```

SEE ALSO

[ipc/publish-async](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/subscribe](#)

Subscribe to a topic.

[ipc/unsubscribe](#)

Unsubscribe from a topic.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/publish-async

(ipc/publish-async topic-name client message)

Publishes a messages to all clients that have subscribed to the message's topic.

Returns a future with the server's response message.

The response message has one of these status:

- `:OK` - message successfully published
- `:SERVER_ERROR` - indicates a server side error while processing the request
- `:BAD_REQUEST` - invalid request, details in the payload

Note: a client in subscription mode can not send or publish messages!

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (defn client-subscribe-handler [m]
    (println "SUBSCRIBED:" (ipc/message->json true m)))

  (try-with [server (ipc/server 3333)
            client1 (ipc/client "localhost" 3333)
            client2 (ipc/client "localhost" 3333)]

    ;; create topic :test
    (ipc/create-topic server :test)

    ;; client1 subscribes to messages with topic :test'
    (ipc/subscribe client1 :test client-subscribe-handler)

    ;; client2 publishes a plain text message:
    ;; requestId="1", subject=:testing, payload="hello"
    (let [m (ipc/plain-text-message "1" :testing "hello")]
      (println "PUBLISHING:" (ipc/message->json true m))
      (-<> (ipc/publish-async client2 :test m)
          (deref <> 1_000 :timeout)
          (ipc/message->json true <>)
          (println "PUBLISHED:" <>)))

    (sleep 300)))
```

SEE ALSO

[ipc/publish](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/subscribe](#)

Subscribe to a topic.

[ipc/unsubscribe](#)

Unsubscribe from a topic.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/queue-status

(ipc/queue-status client name)

Returns a map with the queue status key - values.

Arguments:

client c A client

name n A queue name (string or keyword)

Queue status map keys:

:name	The queue name (string)
:exists	Queue exists: <code>true</code> or <code>false</code>
:type	Queue type "bounded" or "circular"
:temporary	Queue is temporary: <code>true</code> or <code>false</code>
:capacity	The capacity (long)
:size	The current size (long)

```
(try-with [server (ipc/server 33333)
              client (ipc/client 33333)]
  (ipc/create-queue server :orders 100)
  ;; ...
  (ipc/queue-status server :orders))

(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123" :admin)
  (try-with [server (ipc/server 33333
                              :encrypt true
                              :authenticator auth)
              client (ipc/client 33333
                              :user-name "tom"
                              :password "123")]
    (ipc/create-queue client :orders 100)
    ;; ...
    (ipc/exists-status? client :orders)))
```

SEE ALSO

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/create-temporary-queue](#)

Creates a named temporary queue on the server. Temporary queues live as long as the client, that created it, lives.

[ipc/remove-queue](#)

Removes a named queue.

[ipc/exists-queue?](#)

Returns true if the named queue exists else false.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[section](#)[top](#)

ipc/remove-acl

```
(ipc/remove-acl authenticator dest-type dest-name principal)
(ipc/remove-acl authenticator dest-type dest-name)
(ipc/remove-acl authenticator dest-type)
```

Remove an ACL item for a destination and principal.

Arguments:

authenticator	An authenticator
dest-type	A destination type { :queue, :topic, :function }
dest-name	A destination name
principal	A principal

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "max" "456" :admin))

(ipc/add-acl auth :queue :queue/1 :offer "tom")
(ipc/add-acl auth :queue :queue/2 :poll "tom")
(ipc/add-acl auth :topic :topic/1 :publish "tom")
(ipc/add-acl auth :queue :topic/2 :subscribe "tom")
(ipc/add-acl auth :function :echo :execute "tom")

(ipc/remove-acl auth :queue :queue/1 "tom") ;; remove ACLs for queue :queue/1 and user "tom"
(ipc/remove-acl auth :topic :topic/1)      ;; remove all ACLs for topic :topic/1
(ipc/remove-acl auth :function :echo)      ;; remove all ACLs for function :echo
(ipc/remove-acl auth :queue)                ;; remove all queue ACLs
(ipc/remove-acl auth :topic)                ;; remove all topic ACLs
(ipc/remove-acl auth :function)             ;; remove all function ACLs
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/add-acl](#)

Add ACL items for a destination and principal

[ipc/default-acl](#)

Add default ACL items to override the system defaults. ACL default items are set per destination type.

[section](#)[top](#)

ipc/remove-cidr

```
(ipc/remove-cidr authenticator cidr)
```

Remove a CIDR (Classless Inter-Domain Routing) block ACL from an authenticator.

```
(let [auth (ipc/authenticator)]
  (ipc/add-cidr auth "192.168.0.1/16")
  (ipc/remove-cidr auth "192.168.0.1/16"))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/add-cidr](#)

Adds a CIDR (Classless Inter-Domain Routing) block ACL.

[ipc/clear-cidrs](#)

Removes all CIDR (Classless Inter-Domain Routing) block ACLs from an authenticator.

[section](#)

[top](#)

ipc/remove-credentials

```
(ipc/remove-credentials authenticator user-name)
```

Remove user credentials from an authenticator.

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "max" "456" :admin)
  (ipc/remove-credentials auth "tom"))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/add-credentials](#)

Adds user credentials to an authenticator.

[ipc/clear-credentials](#)

Clears all user credentials from an authenticator.

[section](#)

[top](#)

ipc/remove-function

```
(ipc/remove-function server name)
```

Removes a named topic.

Returns always `nil` or throws an exception.

```
(try-with [server (ipc/server 3333)]
  (ipc/create-function server :echo (fn [m] m))
  ;; ...
  (ipc/remove-function server :echo))
```

SEE ALSO

[ipc/create-function](#)

Creates a named function on the server.

[ipc/exists-function?](#)

Returns true if the named topic exists else false.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[section](#)[top](#)

ipc/remove-queue

```
(ipc/remove-queue node name)
```

Removes a named queue.

Returns always `nil` or throws an exception.

Arguments:

node s A server or a client

name n A queue name (string or keyword)

```
(try-with [server (ipc/server 33333)]
  (ipc/create-queue server :orders 100)
  ;; ...
  (ipc/remove-queue server :orders))

(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123" :admin)
  (try-with [server (ipc/server 33333
                              :encrypt true
                              :authenticator auth)
            client (ipc/client 33333
                              :user-name "tom"
                              :password "123")]
    (ipc/create-queue client :orders 100)
    ;; ...
    (ipc/remove-queue client :orders)))
```

SEE ALSO

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/create-temporary-queue](#)

Creates a named temporary queue on the server. Temporary queues live as long as the client, that created it, lives.

[ipc/exists-queue?](#)

Returns true if the named queue exists else false.

[ipc/queue-status](#)

Returns a map with the queue status key - values.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

ipc/remove-topic

```
(ipc/remove-topic node name)
```

Removes a named topic.

Returns always `nil`.

Arguments:

node s A server or a client

name n A topic name (string or keyword)

```
(try-with [server (ipc/server 33333)]
  (ipc/create-topic server :orders-closed)
  ;; ...
  (ipc/remove-topic server :orders-closed))
```

SEE ALSO

[ipc/create-topic](#)

Creates a named topic on the server.

[ipc/exists-topic?](#)

Returns true if the named topic exists else false.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

ipc/response-err?

```
(ipc/response-err? message)
```

Returns `true` if the message has a response error status else `false`.

```
(do
  (defn echo-handler [m] m)
  (try-with [server (ipc/server 33333)
            client (ipc/client "localhost" 33333)]
    (ipc/create-function server :echo echo-handler)
    (->> (ipc/plain-text-message "1" "test" "hello")
      (ipc/send client :echo)
      (ipc/response-err?))))
```

SEE ALSO

[ipc/oneway?](#)

Returns true if the message is one-way else false.

[ipc/response-ok?](#)

Returns true if the message response status is :OK else false.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[section](#)

[top](#)

ipc/response-ok?

```
(ipc/response-ok? message)
```

Returns `true` if the message response status is `:OK` else `false`.

```
(do
  (defn echo-handler [m] m)
  (try-with [server (ipc/server 3333)
            client (ipc/client "localhost" 3333)]
    (ipc/create-function server :echo echo-handler)
    (->> (ipc/plain-text-message "1" "test" "hello")
         (ipc/send client :echo)
         (ipc/response-ok?))))
```

SEE ALSO

[ipc/oneway?](#)

Returns true if the message is one-way else false.

[ipc/response-err?](#)

Returns true if the message has a response error status else false.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[section](#)

[top](#)

ipc/running?

```
(ipc/running? server)
(ipc/running? client)
```

Return `true` if the server or client is running else `false`

```
(do
  (try-with [server (ipc/server 33333)
             client (ipc/client "localhost" 33333)]
    (println "Server running:" (ipc/running? server))
    (println "Client running:" (ipc/running? client))))
```

SEE ALSO

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/close](#)

Closes a server or client

[section](#)

[top](#)

ipc/send

```
(ipc/send client fn-name message)
```

Sends a message to the server the client is associated with.

Returns the server's response message or `nil` if the message is declared as one-way message. Throws a timeout exception if the response is not received within the timeout time.

The response message has one of these status:

- `:OK` - request handled successfully and response holds the data
- `:SERVER_ERROR` - indicates a server side error while processing the request
- `:BAD_REQUEST` - invalid request, details in the payload
- `:HANDLER_ERROR` - an error in the server's request processing handler

Arguments:

client c	A client to send the message from
fn-name n	The server function the message is sent to
message m	The message to send

Note

If the server's handler function takes more than a couple of 10 milliseconds to process the request, consider to use *offer/poll* with a reply queue to improve the system throughput!

```
;; echo handler
;; request: "hello" => echo => response: "hello"
(do
  (try-with [server (ipc/server 33333)
             client (ipc/client "localhost" 33333)]
```

```

(ipc/create-function server :echo (fn [m] m))
(->> (ipc/plain-text-message "1" "test" "hello")
      (ipc/send client :echo)
      (ipc/message->map)
      (println)))

;; handler processing JSON message data
;; request: {"x": 100, "y": 200} => add => response: {"z": 300}
(do
  (defn handler [m]
    (let [data (json/read-str (. m :getText))
          result (json/write-str { "z" (+ (get data "x") (get data "y"))})]
      (ipc/text-message (. m :getRequestId)
                        (. m :getTopic)
                        "application/json" :UTF-8
                        result)))
    (try-with [server (ipc/server 33333)
               client (ipc/client "localhost" 33333)]
      (ipc/create-function server :handler handler)
      (->> (ipc/text-message "1"
                              "test"
                              "application/json" :UTF-8
                              (json/write-str {"x" 100 "y" 200}))
            (ipc/send client :handler)
            (ipc/message->map)
            (println))))

;; handler with remote code execution
;; request: "(+ 1 2)" => exec => response: "3"
(do
  (defn exec [m]
    (let [cmd (. m :getText)
          result (str (eval (read-string cmd)))]
      (ipc/plain-text-message (. m :getRequestId)
                              (. m :getTopic)
                              result)))
    (try-with [server (ipc/server 33333)
               client (ipc/client "localhost" 33333)]
      (ipc/create-function server :exec exec)
      (->> (ipc/plain-text-message "1" "exec" "(+ 1 2)")
            (ipc/send client :exec)
            (ipc/message->map)
            (println))))

```

SEE ALSO

[ipc/send-async](#)

Sends a message to the server the client is associated with.

[ipc/send-oneway](#)

Sends a one-way message to the server the client is associated with.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/send-async

```
(ipc/send-async client fn-name message)
```

Sends a message to the server the client is associated with.

Returns a future with the server's response message. The response message is `nil` if the message is declared as one-way message.

The response message has one of these status:

- `:OK` - request handled successfully and response holds the data
- `:SERVER_ERROR` - indicates a server side error while processing the request
- `:BAD_REQUEST` - invalid request, details in the payload
- `:HANDLER_ERROR` - an error in the server's request processing handler

Arguments:

client c A client to send the message from

fn-name n The server function the message is sent to

message m The message to send

Note

If the server's handler function takes more than a couple of 10 milliseconds to process the request, consider to use *offer/poll* with a reply queue to improve the system throughput!

```
;; echo handler
;; request: "hello" => echo => response: "hello"
(do
  (try-with [server (ipc/server 3333)
                client (ipc/client "localhost" 3333)]
    (ipc/create-function server :echo (fn [m] m))
    (-<> (ipc/plain-text-message "1" "test" "hello")
        (ipc/send-async client :echo <>)
        (deref <> 1_000 :timeout)
        (ipc/message->map <>)
        (println <>))))

;; handler processing JSON message data
;; request: {"x": 100, "y": 200} => add => response: {"z": 300}
(do
  (defn handler [m]
    (let [data (json/read-str (. m :getText))
          result (json/write-str { "z" (+ (get data "x") (get data "y"))})]
      (ipc/text-message (. m :getRequestId)
                        (. m :getTopic))))
```

```

        "application/json" :UTF-8
        result)))
(try-with [server (ipc/server 33333)
            client (ipc/client "localhost" 33333)]
  (ipc/create-function server :handler handler)
  (-<> (ipc/text-message "1"
                        "test"
                        "application/json" :UTF-8
                        (json/write-str {"x" 100 "y" 200})))
      (ipc/send-async client :handler <>)
      (deref <> 1_000 :timeout)
      (ipc/message->map <>)
      (println <>))))

;; handler with remote code execution
;; request: "(+ 1 2)" => exec => response: "3"
(do
  (defn exec [m]
    (let [cmd (. m :getText)
          result (str (eval (read-string cmd)))]
      (ipc/plain-text-message (. m :getRequestId)
                             (. m :getTopic)
                             result)))
  (try-with [server (ipc/server 33333)
              client (ipc/client "localhost" 33333)]
    (ipc/create-function server :exec exec)
    (-<> (ipc/plain-text-message "1" "exec" "(+ 1 2)")
        (ipc/send-async client :exec <>)
        (deref <> 1_000 :timeout)
        (ipc/message->map <>)
        (println <>))))

```

SEE ALSO

[ipc/send](#)

Sends a message to the server the client is associated with.

[ipc/send-oneway](#)

Sends a one-way message to the server the client is associated with.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

ipc/send-oneway

```
(ipc/send-oneway client fn-name message)
```

Sends a one-way message to the server the client is associated with.

Does not wait for a response and returns always `nil`.

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (defn handler [m]
    (println "REQUEST:" (ipc/message->json true m))
    nil)

  (try-with [server (ipc/server 3333)
            client (ipc/client "localhost" 3333)]

    (ipc/create-function server :handler handler)

    ;; send a plain text messages:
    ;;   requestId="1" and "2", subject="test", payload="hello"
    (ipc/send-oneway client :handler (ipc/plain-text-message "1" "test" "hello"))
    (ipc/send-oneway client :handler (ipc/plain-text-message "2" "test" "hello"))))
```

SEE ALSO

[ipc/send](#)

Sends a message to the server the client is associated with.

[ipc/send-async](#)

Sends a message to the server the client is associated with.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

ipc/server

(ipc/server port & options)
(ipc/server conn-uri & options)

Create a new server on the specified port or connection URI.

Arguments:

port p	A TCP/IP port. E.g.: 33333
conn-uri u	A connection URI • TCP/IP sockets af-inet://localhost:33333 • Unix domain sockets (requires unixsocket libraries!) af-unix:///data/ipc/test.sock

Options:

:max-connections n	The number of the max connections the server can handle in parallel. Defaults to 20.
:max-message-size n	The max size of the message payload. Defaults to 50MB . The max size can be specified as a number like 20000 or a number with a unit like :20KB or :20MB . The value must be in the range 2KB ... 250MB
:max-queues n	The number of the max queues the server can handle. Defaults to 20.
:max-topics n	The number of the max topics the server can handle. Defaults to 20.
:max-functions n	The number of the max functions the server can handle. Defaults to 20.
:compress-cutoff-size n	The compression cutoff size for payload messages. With a negative cutoff size payload messages will not be compressed. If the payload message size is greater than the cutoff size it will be compressed. The cutoff size can be specified as a number like 1000 or a number with a unit like :1KB or :2MB . Defaults to -1 (no compression)
:encrypt b	If true encrypt the payload data of all messages exchanged with this server. The data is AES-256-GCM encrypted using a secret that is created and exchanged using the Diffie-Hellman key exchange algorithm.
:dh-rsa-sign	If true sign and verify the secrets exchanged with Diffie-Hellman key exchange with RSA private/public keys. Defaults to false .
:dh-rsa-server-key-pair k	The server RSA key pair for signing the Diffie-Hellman secret. Defaults to nil .
:dh-rsa-client-public-key k	The client RSA public key for verifying the Diffie-Hellman secret. May be a vector of client RSA public keys. Defaults to nil .
:server-log-dir f	If the server-log-dir is specified writes a server log to this directory. Defaults to nil .
:write-ahead-log-dir f	Provide a write-ahead-log directory to support durable queues. Defaults to nil .
:write-ahead-log-compress b	If true compresses the write-ahead-log records. Defaults to false .
:write-ahead-log-compact b	If true compacts the write-ahead-logs at server start. Defaults to false .
:authenticator a	An authenticator. If an authenticator is used encryption must be enabled to safely transmit users credentials! Defaults to nil .
:dead-letter-queue-size n	The dead letter queue size. A size of 0 deactivates the dead letter queue. Note that the dead letter queue is not durable.

	Defaults to a size of 100
:socket-snd-buf-size n	The server socket's send buffer size. Defaults to <code>-1</code> (use the sockets default buf size). The size can be specified as a number like <code>64536</code> or a number with a unit like <code>:64KB</code> or <code>:1MB</code>.
:socket-rcv-buf-size n	The server socket's receive buffer size. Defaults to <code>-1</code> (use the sockets default buf size). The size can be specified as a number like <code>64536</code> or a number with a unit like <code>:64KB</code> or <code>:1MB</code>.
:heartbeat-interval n	Connection heartbeat interval (in seconds). When the heartbeat mechanism is enabled (interval > 0), connected clients send a heartbeat message to the server at the configured interval. If the server fails to receive three consecutive heartbeat messages from a client, it considers the client dead and closes the connection. Defaults to 0 (heartbeat turned off).

The server must be closed after use!

[See Inter-Process-Communication](#)

```
(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m))
    m)

  (try-with [server (ipc/server 33333 )
             client (ipc/client "localhost" 33333)]
    (ipc/create-function server :echo echo-handler)
    (->> (ipc/plain-text-message "1" "test" "hello")
         (ipc/send client :echo)
         (ipc/message->map)
         (println "RESPONSE: "))))

(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m))
    m)

  (try-with [server (ipc/server "af-inet://localhost:33333")
             client (ipc/client "af-inet://localhost:33333")]
    (ipc/create-function server :echo echo-handler)
    (->> (ipc/plain-text-message "1" "test" "hello")
         (ipc/send client :echo)
         (ipc/message->map)
         (println "RESPONSE: "))))

(do
  (defn echo-handler [m]
    (println "REQUEST: " (ipc/message->map m))
    m)

  (let [a (ipc/authenticator)]
    (ipc/add-credentials a "tom" "3-kio")
    (try-with [server (ipc/server 33333
                             :encrypt true
                             :authenticator a)
               client (ipc/client "localhost" 33333
                             :user-name "tom"
                             :password "3-kio")]
      (ipc/create-function server :echo echo-handler)
      (->> (ipc/plain-text-message "1" "test" "hello")
           (ipc/send client :echo)
           (ipc/message->map)
           (println "RESPONSE: ")))))
```

SEE ALSO

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/send](#)

Sends a message to the server the client is associated with.

[ipc/send-oneway](#)

Sends a one-way message to the server the client is associated with.

[ipc/publish](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/subscribe](#)

Subscribe to a topic.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[ipc/create-queue](#)

Creates a named queue on the server. Messages can be exchanged asynchronously between two clients using a queue. Each message is delivered ...

[ipc/remove-queue](#)

Removes a named queue.

[ipc/benchmark](#)

Runs a benchmark.

[ipc/server-status](#)

Returns the status and statistics of the server the client is connected to.

[ipc/server-thread-pool-statistics](#)

Returns the server's thread pool statistics the client is connected to.

[section](#)

[top](#)

ipc/server-status

(ipc/server-status client)

Returns the status and statistics of the server the client is connected to.


```
(do
  (let [auth (ipc/authenticator)]
    (ipc/add-credentials auth "admin" "3-kio" :admin)

    (try-with [server (ipc/server 33333
                                :encrypt true
                                :authenticator auth)
              client (ipc/client "localhost" 33333
                                :user-name "admin"
                                :password "3-kio")]
              (ipc/create-function server :echo (fn [m] m))

              (-> (ipc/plain-text-message "1" "test" "hello")
                  (ipc/send client :echo)))

    ;; requires a user with admin authorization
    (println "STATUS:" (ipc/server-status client))))))
```

SEE ALSO

[ipc/server-thread-pool-statistics](#)

Returns the server's thread pool statistics the client is connected to.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[section](#)

[top](#)

ipc/server-thread-pool-statistics

```
(ipc/server-thread-pool-statistics client)
```

Returns the server's thread pool statistics the client is connected to.

```
(do
  (let [auth (ipc/authenticator)]
    (ipc/add-credentials auth "admin" "3-kio" :admin)

    (try-with [server (ipc/server 33333
                                :encrypt true
                                :authenticator auth)
              client (ipc/client "localhost" 33333
                                :user-name "admin"
                                :password "3-kio")]
              (ipc/create-function server :echo (fn [m] m))

              (-> (ipc/plain-text-message "1" "test" "hello")
                  (ipc/send client :echo)))

    ;; requires a user with admin authorization
    (println "STATS:"
             (ipc/server-thread-pool-statistics client))))))
```

SEE ALSO

[ipc/server-status](#)

Returns the status and statistics of the server the client is connected to.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/close](#)

Closes a server or client

[ipc/running?](#)

Return true if the server or client is running else false

[section](#)[top](#)

ipc/store-authenticator

```
(ipc/store-authenticator authenticator dest)
```

Stores an authenticator to a file or an output stream.

Passwords are stored as salted PBKDF2 hashes!

```
(let [auth (ipc/authenticator)]
  (ipc/add-credentials auth "tom" "123")
  (ipc/add-credentials auth "max" "456")
  (ipc/store-authenticator auth (io/file "./ipc.cred")))
```

SEE ALSO

[ipc/authenticator](#)

Creates a new authenticator the manages the credential on behalf of the server.

[ipc/load-authenticator](#)

Loads an authenticator from a file or an input stream.

[section](#)[top](#)

ipc/subscribe

```
(ipc/subscribe client topic msg-handler)
```

Subscribe to a topic.

Multiple subscriptions with different handlers are supported.

Returns the server's response message.

The response message has one of these status:

- `:OK` - subscription added. Subscribed messages will be delivered through the 'msg-handler'
- `:SERVER_ERROR` - indicates a server side error while processing the request
- `:BAD_REQUEST` - invalid request, details in the payload

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg))))
```

```
(defn client-subscribe-handler [m]
  (println "SUBSCRIBED:" (ipc/message->json true m)))

(try-with [server (ipc/server 3333)
          client1 (ipc/client "localhost" 3333)
          client2 (ipc/client "localhost" 3333)]

  ;; create topic :test
  (ipc/create-topic server :test)

  ;; client1 subscribes to messages with topic 'test'
  (ipc/subscribe client1 :test client-subscribe-handler)

  ;; client2 publishes a plain text message:
  ;; requestId="1", subject="test", payload="hello"
  (let [m (ipc/plain-text-message "1" "test" "hello")]
    (println "PUBLISHED:" (ipc/message->json true m))
    (ipc/publish client2 :test m))

  (sleep 300)))
```

SEE ALSO

[ipc/unsubscribe](#)

Unsubscribe from a topic.

[ipc/publish](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/publish-async](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/text-message

```
(ipc/text-message request-id topic mimetype charset text)
(ipc/text-message request-id topic mimetype charset text durable)
(ipc/text-message request-id topic mimetype charset text durable expires-at)
(ipc/text-message request-id topic mimetype charset text durable expires-val expires-unit)
```

Creates a text message

Arguments:

request-id r	A request ID (string, may be <code>nil</code>). May be used for idempotency checks by the receiver
topic t	A topic (string or keyword)
mimetype m	The mimetype of the payload text. A string like 'text/plain'
charset c	The charset of the payload text. A keyword like <code>:UTF-8</code>
text t	The message payload text (a string)
durable b	If <code>true</code> create a durable message if the server supports durable queues. Defaults to <code>false</code> .
expires-at t	Message expiration time in millis since epoch (may be <code>nil</code>)
expires-val v	Message expiration duration. E.g.: 2 (may be <code>nil</code>)
expires-unit u	Message expiration time unit. Units: {:years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(->> (ipc/text-message "1" :test
                        "text/plain" :UTF-8 "hello")
      (ipc/message->map)
      (println))

(->> (ipc/text-message "1" "test"
                        "text/plain" :UTF-8 "hello")
      (ipc/message->map)
      (println))

(->> (ipc/text-message "1" "test"
                        "text/plain" :UTF-8 "hello"
                        false
                        2 :hours)
      (ipc/message->map)
      (println))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[ipc/oneway?](#)

Returns true if the message is one-way else false.

[ipc/response-ok?](#)

Returns true if the message response status is :OK else false.

[ipc/response-err?](#)

Returns true if the message has a response error status else false.

ipc/unsubscribe

(ipc/unsubscribe client topic)

Unsubscribe from a topic.

Returns the server's response message.

The response message has one of these status:

- `:OK` - subscription added. Subscribed messages will be delivered through the 'msg-handler'
- `:SERVER_ERROR` - indicates a server side error while processing the request
- `:BAD_REQUEST` - invalid request, details in the payload

```
(do
  ;; thread-safe printing
  (defn println [& msg] (locking println (apply core/println msg)))

  (defn client-subscribe-handler [m]
    (println "SUBSCRIBED:" (ipc/message->json true m)))

  (try-with [server (ipc/server 3333)
            client1 (ipc/client "localhost" 3333)
            client2 (ipc/client "localhost" 3333)]

    ;; create topic :test
    (ipc/create-topic server :test)

    ;; client1 subscribes to messages with topic 'test'
    (ipc/subscribe client1 "test" client-subscribe-handler)

    ;; client1 unsubscribes from messages with topic 'test'
    (ipc/unsubscribe client1 "test")

    ;; client2 publishes a plain text message:
    ;; requestId="1", subject="test", payload="hello"
    (let [m (ipc/plain-text-message "1" "test" "hello")]
      (println "PUBLISHED:" (ipc/message->json true m))
      (ipc/publish client2 m))

    (sleep 300)))
```

SEE ALSO

[ipc/subscribe](#)

Subscribe to a topic.

[ipc/publish](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/publish-async](#)

Publishes a messages to all clients that have subscribed to the message's topic.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/venice-message](#)

Creates a venice message.

[ipc/binary-message](#)

Creates a binary message.

[ipc/message->map](#)

Converts a message to a Venice map.

[section](#)

[top](#)

ipc/venice-message

```
(ipc/venice-message request-id topic data)
(ipc/venice-message request-id topic data durable)
(ipc/venice-message request-id topic data durable expires-at)
(ipc/venice-message request-id topic data durable expires-val expires-unit)
```

Creates a venice message.

The Venice data is serialized as JSON (mimetype: 'application/json') for transport within the message.

Arguments:

request-id r	A request ID (string, may be <code>nil</code>). May be used for idempotency checks by the receiver
topic t	A topic (string or keyword)
data d	The message payload Venice data (e.g.: a map, list, ...)
durable b	If <code>true</code> create a durable message if the server supports durable queues. Defaults to <code>false</code> .
expires-at t	Message expiration time in millis since epoch (may be <code>nil</code>)
expires-val v	Message expiration duration. E.g.: 2 (may be <code>nil</code>)
expires-unit u	Message expiration time unit. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(->> (ipc/venice-message "100" :test {:a 100, :b 200})
      (ipc/message->map)
      (println))

(->> (ipc/venice-message "100" "test" {:a 100, :b 200})
      (ipc/message->map)
      (println))

(->> (ipc/venice-message "100" "test"
                          {:a 100, :b 200}
                          false
                          2 :hours)
      (ipc/message->map)
      (println))
```

SEE ALSO

[ipc/server](#)

Create a new server on the specified port or connection URI.

[ipc/client](#)

Create a new client connecting to a server on the specified host and port.

[ipc/text-message](#)

Creates a text message

[ipc/plain-text-message](#)

Creates a plain text message with mimetype text/plain and charset :UTF-8.

[ipc/message->map](#)

Converts a message to a Venice map.

[ipc/message->json](#)

Converts message to a Json string with optional pretty printing.

[ipc/oneway?](#)

Returns true if the message is one-way else false.

[ipc/response-ok?](#)

Returns true if the message response status is :OK else false.

[ipc/response-err?](#)

Returns true if the message has a response error status else false.

[section](#)

[top](#)

jansi-version

```
(jansi-version)
```

Returns the Jansi version or nil if not available.

[section](#)

[top](#)

jar-maven-manifest-version

```
(jar-maven-manifest-version group-id artefact-id)
```

Returns the Maven version for a loaded JAR's manifest or nil if there is no Maven manifest.

Reads the version from the JAR's Maven 'pom.properties' file at:
/META-INF/maven/{group-id}/{artefact-id}/pom.properties

A 'pom.properties' may look like:

- artifactId=xchart
- groupId=org.knowm.xchart
- version=3.8.0

```
(jar-maven-manifest-version :com.github.librepdf :openpdf)  
=> "1.3.35"
```

SEE ALSO

[java-package-version](#)

Returns version information for a Java package or nil if the package does not exist or is not visible.

[section](#)

[top](#)

java->venice

```
(java->venice obj)
```

Converts a Java object to Venice data.

Converts scalar values and collections.

[section](#)

[top](#)

java-double-list

```
(java-double-list l)
```

Converts a Venice list/vector to a Java `Double` list

```
(java-double-list '(1.0 2.0 3.0))  
=> [1.0, 2.0, 3.0]
```

```
(java-double-list '(1I 2 3.2 3.56M))  
=> [1.0, 2.0, 3.2, 3.56]
```

SEE ALSO

[java-float-list](#)

Converts a Venice list/vector to a Java Float list

[double-array](#)

Returns an array of Java primitive doubles containing the contents of coll or returns an array with the given length and optional init value.

[section](#)

[top](#)

java-enumeration-to-list

```
(java-enumeration-to-list e)
```

Converts a Java enumeration to a list

[section](#)

[top](#)

java-float-list

```
(java-float-list l)
```

Converts a Venice list/vector to a Java `Float` list

```
(java-float-list '(1.0F 2.0F 3.0F))  
=> [1.0, 2.0, 3.0]
```

```
(java-float-list '(1I 2 3.2 3.56M))  
=> [1.0, 2.0, 3.2, 3.56]
```

SEE ALSO

[java-double-list](#)

Converts a Venice list/vector to a Java Double list

[float-array](#)

Returns an array of Java primitive floats containing the contents of coll or returns an array with the given length and optional init value.

[section](#)

[top](#)

java-home

(java-home)

Returns the installation directory for Java Runtime Environment (JRE)

(java-home)

=> "/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/jre"

SEE ALSO

[java-version](#)

Returns the Java VM version (1.8.0_252, 11.0.7, ...)

[java-vm-args](#)

Returns the Java JVM arguments passed.

[section](#)

[top](#)

java-int-list

(java-int-list l)

Converts a Venice list/vector to a Java `Integer` list

(java-int-list '(1I 2I 3I))

=> [1, 2, 3]

(java-int-list '(1I 2 3.2 3.56M))

=> [1, 2, 3, 3]

SEE ALSO

[int-array](#)

Returns an array of Java primitive ints containing the contents of coll or returns an array with the given length and optional init value.

[section](#)

[top](#)

java-iterator-to-list

(java-iterator-to-list e)

Converts a Java iterator to a list

[section](#)

[top](#)

java-long-list

```
(java-long-list l)
```

Converts a Venice list/vector to a Java `Long` list

```
(java-long-list '(1 2 3))  
=> [1, 2, 3]
```

```
(java-long-list '(1I 2 3.2 3.56M))  
=> [1, 2, 3, 3]
```

SEE ALSO

[long-array](#)

Returns an array of Java primitive longs containing the contents of coll or returns an array with the given length and optional init value.

[section](#)

[top](#)

java-major-version

```
(java-major-version)
```

Returns the Java major version (8, 17, 21, ...).

```
(java-major-version)  
=> 8
```

SEE ALSO

[java-version](#)

Returns the Java VM version (1.8.0_252, 11.0.7, ...)

[java-version-info](#)

Returns the Java VM version info.

[section](#)

[top](#)

java-obj?

```
(java-obj? obj)
```

Returns true if obj is a Java object

```
(java-obj? (. :java.math.BigInteger :new "0"))  
=> true
```

[section](#)

[top](#)

java-optional?

```
(java-optional? val)
```

Returns `true` if 'val' is a Java `:java.util.Optional` else `false`

SEE ALSO

[java-unwrap-optional](#)

Unwraps a Java `:java.util.Optional` to its contained value.

[section](#)

[top](#)

java-package-version

```
(java-package-version class)
```

Returns version information for a Java package or nil if the package does not exist or is not visible.

```
(java-package-version :java.lang.String)
```

```
=> {:implementation-title "Java Runtime Environment" :implementation-vendor "Azul Systems, Inc." :  
implementation-version "1.8.0_462" :specification-title "Java Platform API Specification" :specification-vendor  
"Oracle Corporation" :specification-version "1.8"}
```

```
(java-package-version (class :java.lang.String))
```

```
=> {:implementation-title "Java Runtime Environment" :implementation-vendor "Azul Systems, Inc." :  
implementation-version "1.8.0_462" :specification-title "Java Platform API Specification" :specification-vendor  
"Oracle Corporation" :specification-version "1.8"}
```

SEE ALSO

[jar-maven-manifest-version](#)

Returns the Maven version for a loaded JAR's manifest or nil if there is no Maven manifest.

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[section](#)

[top](#)

java-source-location

```
(java-source-location class)
```

Returns the path of the source location of a class (fully qualified class name).

E.g.:

```
(java-source-location :com.github.jlangch.venice.Venice)
```

```
=> /Users/foo/venice/libs/venice-1.12.57.jar
```

```
(java-source-location :com.github.jlangch.venice.Venice)
```

[section](#)

[top](#)

java-string-list

```
(java-string-list l)
```

Converts a Venice list/vector to a Java `String` list

```
(java-string-list '("ab" "cd" "ef"))  
=> [ab, cd, ef]
```

```
(java-string-list '("ab" 1i 2 3.2 3.56M))  
=> [ab, 1, 2, 3.2, 3.56]
```

SEE ALSO

[string-array](#)

Returns an array of Java strings containing the contents of coll or returns an array with the given length and optional init value

[section](#)

[top](#)

java-unwrap-optional

```
(java-unwrap-optional val)  
(java-unwrap-optional val elseVal)
```

Unwraps a Java `:java.util.Optional` to its contained value.

Returns `nil` in the one-argument version if the value is not present.

Returns the 'elseVal' in the two-argument version if the value is not present.

SEE ALSO

[java-optional?](#)

Returns true if 'val' is a Java `:java.util.Optional` else false

[section](#)

[top](#)

java-version

```
(java-version)
```

Returns the Java VM version (1.8.0_252, 11.0.7, ...)

```
(java-version)  
=> "1.8.0_462"
```

SEE ALSO

[java-major-version](#)

Returns the Java major version (8, 17, 21, ...).

[java-version-info](#)

Returns the Java VM version info.

[java-home](#)

Returns the installation directory for Java Runtime Environment (JRE)

[java-vm-args](#)

Returns the Java JVM arguments passed.

java-version-info

(java-version-info)

Returns the Java VM version info.

(java-version-info)

```
=> {:version "1.8.0_462" :vendor "Azul Systems, Inc." :vm-version "25.462-b08" :vm-name "OpenJDK 64-Bit Server VM" :vm-vendor "Azul Systems, Inc."}
```

SEE ALSO

[java-version](#)

Returns the Java VM version (1.8.0_252, 11.0.7, ...)

[java-major-version](#)

Returns the Java major version (8, 17, 21, ...).

[java-vm-args](#)

Returns the Java JVM arguments passed.

java-vm-args

(java-vm-args)

Returns the Java JVM arguments passed.

(java-vm-args)

```
=> ("Dfile.encoding=UTF-8" "-Duser.country=CH" "-Duser.language=en" "-Duser.variant")
```

SEE ALSO

[java-home](#)

Returns the installation directory for Java Runtime Environment (JRE)

[java-version](#)

Returns the Java VM version (1.8.0_252, 11.0.7, ...)

[java-version-info](#)

Returns the Java VM version info.

java/as-biconsumer

(as-biconsumer f)

Wraps the function f in a [java.util.function.BiConsumer](#)

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static void testBiConsumer(BiConsumer<Long,Long> f, Long t, Long u) {
  ;;   f.accept(t,u);
  ;; }

  (defn op [t u] (println "consumed" t u))
  (. :FunctionalInterfaces :testBiConsumer (j/as-biconsumer op) 1 2))
consumed 1 2
=> nil
```

SEE ALSO

[java/as-bipredicate](#)

Wraps the function f in a [java.util.function.BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html>)

[java/as-bifunction](#)

Wraps the function f in a [java.util.function.BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html>)

[java/as-unaryoperator](#)

Wraps the function f in a [java.util.function.UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html>)

[java/as-binaryoperator](#)

Wraps the function f in a [java.util.function.BinaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html>)

[section](#)
[top](#)

java/as-bifunction

```
(as-bifunction f)
```

Wraps the function f in a [java.util.function.BiFunction](#)

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static Long testBiFunction(BiFunction<Long,Long,Long> f, Long t, Long u) {
  ;;   return f.apply(t,u);
  ;; }

  (defn op [t u] (+ t u))
  (. :FunctionalInterfaces :testBiFunction (j/as-bifunction op) 1 2))
=> 3
```

SEE ALSO

[java/as-bipredicate](#)

Wraps the function f in a [java.util.function.BiPredicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html>)

[java/as-biconsumer](#)

Wraps the function f in a [java.util.function.BiConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html>)

[java/as-unaryoperator](#)

Wraps the function f in a [java.util.function.UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html>)

[java/as-binaryoperator](#)

Wraps the function f in a [java.util.function.BinaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html>)

[section](#)
[top](#)

java/as-binaryoperator

(as-binaryoperator f)

Wraps the function f in a [java.util.function.BinaryOperator](#)

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static Long testBinaryOperator(BinaryOperator<Long> f, Long t, Long u) {
  ;;   return f.apply(t,u);
  ;; }

  (defn op [t u] (+ t u))
  (. :FunctionalInterfaces :testBinaryOperator (j/as-binaryoperator op) 1 2))
=> 3
```

SEE ALSO

[java/as-bipredicate](#)

Wraps the function f in a [java.util.function.BiPredicate](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html>)

[java/as-bifunction](#)

Wraps the function f in a [java.util.function.BiFunction](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html>)

[java/as-biconsumer](#)

Wraps the function f in a [java.util.function.BiConsumer](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html>)

[java/as-unaryoperator](#)

Wraps the function f in a [java.util.function.UnnaryOperator](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html>)

[section](#)

[top](#)

java/as-bipredicate

(as-bipredicate f)

Wraps the function f in a [java.util.function.BiPredicate](#)

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static boolean testBiPredicate(BiPredicate<Long,Long> f, Long t, Long u) {
  ;;   return f.test(t,u);
  ;; }

  (defn op [t u] (> t u))
  (. :FunctionalInterfaces :testBiPredicate (j/as-bipredicate op) 1 2))
=> false
```

SEE ALSO

[java/as-bifunction](#)

Wraps the function f in a [java.util.function.BiFunction](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html>)

[java/as-biconsumer](#)

Wraps the function `f` in a `java.util.function.BiConsumer` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html>)

[java/as-unaryoperator](#)

Wraps the function `f` in a `java.util.function.UnaryOperator` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html>)

[java/as-binaryoperator](#)

Wraps the function `f` in a `java.util.function.BinaryOperator` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html>)

[section](#)

[top](#)

java/as-callable

`(as-callable f)`

Wraps the function `f` in a `java.util.concurrent.Callable`

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static Long testCallable(Callable<Long> c) throws Exception {
  ;;   return c.call();
  ;; }

  (defn op [] 4)
  (. :FunctionalInterfaces :testCallable (j/as-callable op)))
=> 4
```

SEE ALSO

[java/as-runnable](#)

Wraps the function `f` in a `java.lang.Runnable` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-predicate](#)

Wraps the function `f` in a `java.util.function.Predicate` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

[java/as-function](#)

Wraps the function `f` in a `java.util.function.Function` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

[java/as-consumer](#)

Wraps the function `f` in a `java.util.function.Consumer` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

[java/as-supplier](#)

Wraps the function `f` in a `java.util.function.Supplier` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html>)

[section](#)

[top](#)

java/as-consumer

`(as-consumer f)`

Wraps the function `f` in a `java.util.function.Consumer`

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static void testConsumer(Consumer<Long> f, Long t) {
  ;;   f.accept(t);
  ;; }
```



```
;; }

(defn op [t] (println "consumed" t))
(. :FunctionalInterfaces :testConsumer (j/as-consumer op) 4))
consumed 4
=> nil
```

SEE ALSO

[java/as-runnable](#)

Wraps the function f in a `java.lang.Runnable` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-callable](#)

Wraps the function f in a `java.util.concurrent.Callable` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

[java/as-predicate](#)

Wraps the function f in a `java.util.function.Predicate` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

[java/as-function](#)

Wraps the function f in a `java.util.function.Function` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

[java/as-supplier](#)

Wraps the function f in a `java.util.function.Supplier` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html>)

[section](#)

[top](#)

java/as-function

(as-function f)

Wraps the function f in a `java.util.function.Function`

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static Long testFunction(Function<Long,Long> f, Long t) {
  ;;   return f.apply(t);
  ;; }

  (defn op [t] (+ t 1))
  (. :FunctionalInterfaces :testFunction (j/as-function op) 4))
=> 5
```

SEE ALSO

[java/as-runnable](#)

Wraps the function f in a `java.lang.Runnable` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-callable](#)

Wraps the function f in a `java.util.concurrent.Callable` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

[java/as-predicate](#)

Wraps the function f in a `java.util.function.Predicate` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

[java/as-consumer](#)

Wraps the function f in a `java.util.function.Consumer` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

[java/as-supplier](#)

Wraps the function f in a `java.util.function.Supplier` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html>)

[section](#)

[top](#)

java/as-predicate

```
(as-predicate f)
```

Wraps the function `f` in a [java.util.function.Predicate](#)

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static boolean testPredicate(Predicate<Long> p, Long t) {
  ;;   return p.test(t);
  ;; }

  (defn op [t] (pos? t))
  (. :FunctionalInterfaces :testPredicate (j/as-predicate op) 4))
=> true
```

SEE ALSO

[java/as-runnable](#)

Wraps the function `f` in a [java.lang.Runnable](#) (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-callable](#)

Wraps the function `f` in a [java.util.concurrent.Callable](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

[java/as-function](#)

Wraps the function `f` in a [java.util.function.Function](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

[java/as-consumer](#)

Wraps the function `f` in a [java.util.function.Consumer](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

[java/as-supplier](#)

Wraps the function `f` in a [java.util.function.Supplier](#) (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html>)

[section](#)[top](#)

java/as-runnable

```
(as-runnable f)
```

Wraps the function `f` in a [java.lang.Runnable](#)

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static void testRunnable(final Runnable r) {
  ;;   r.run();
  ;; }

  (defn op [] (println "running"))
  (. :FunctionalInterfaces :testRunnable (j/as-runnable op)))
running
=> nil
```

SEE ALSO

[java/as-callable](#)

Wraps the function f in a `java.util.concurrent.Callable` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

[java/as-predicate](#)

Wraps the function f in a `java.util.function.Predicate` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

[java/as-function](#)

Wraps the function f in a `java.util.function.Function` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

[java/as-consumer](#)

Wraps the function f in a `java.util.function.Consumer` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

[java/as-supplier](#)

Wraps the function f in a `java.util.function.Supplier` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html>)

[section](#)[top](#)

java/as-supplier

`(as-supplier f)`

Wraps the function f in a `java.util.function.Supplier`

```
(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static Long testSupplier(Supplier<Long> f) {
  ;;   return f.get();
  ;; }

  (defn op [] 5)
  (. :FunctionalInterfaces :testSupplier (j/as-supplier op)))
=> 5
```

SEE ALSO

[java/as-runnable](#)

Wraps the function f in a `java.lang.Runnable` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-callable](#)

Wraps the function f in a `java.util.concurrent.Callable` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

[java/as-predicate](#)

Wraps the function f in a `java.util.function.Predicate` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

[java/as-function](#)

Wraps the function f in a `java.util.function.Function` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

[java/as-consumer](#)

Wraps the function f in a `java.util.function.Consumer` (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

[section](#)[top](#)

java/as-unaryoperator

`(as-unaryoperator f)`

Wraps the function f in a `java.util.function.UnnaryOperator`

```

(do
  (load-module :java ['java :as 'j])
  (import :com.github.jlangch.venice.demo.FunctionalInterfaces)

  ;; public static Long testUnaryOperator(UnaryOperator<Long> f, Long t) {
  ;;   return f.apply(t);
  ;; }

  (defn op [t] (+ t 1))
  (. :FunctionalInterfaces :testUnaryOperator (j/as-unaryoperator op) 1))
=> 2

```

SEE ALSO

[java/as-bipredicate](#)

Wraps the function f in a java.util.function.BiPredicate (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html>)

[java/as-bifunction](#)

Wraps the function f in a java.util.function.BiFunction (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html>)

[java/as-biconsumer](#)

Wraps the function f in a java.util.function.BiConsumer (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html>)

[java/as-binaryoperator](#)

Wraps the function f in a java.util.function.BinaryOperator (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html>)

[section](#)
[top](#)

java/javadoc

```
(javadoc class-or-object)
```

Opens a browser window displaying the javadoc for argument.

```
(java/javadoc :java.lang.String)
```

[section](#)
[top](#)

jdbc-core/auto-commit!

```
(auto-commit! conn on)
```

Activate/Deactivate auto commit on a connection

```

(jdbc-core/auto-commit! conn true)
(jdbc-core/auto-commit! conn false)
(jdbc-core/auto-commit! conn :on)
(jdbc-core/auto-commit! conn :off)

```

SEE ALSO

[jdbc-core/tx-isolation](#)

Returns transaction isolation level of the connection

[jdbc-core/tx-isolation!](#)

Set the transaction isolation level for the connection

[jdbc-core/auto-commit?](#)

Returns true if auto commit is enabled on the connection else false

[jdbc-core/commit!](#)

Commit the current transaction on the connection

[jdbc-core/rollback!](#)

Rollback the current transaction on the connection

[section](#)

[top](#)

jdbc-core/auto-commit?

(auto-commit? conn)

Returns true if auto commit is enabled on the connection else false

([jdbc-core/auto-commit?](#) conn)

SEE ALSO

[jdbc-core/tx-isolation](#)

Returns transaction isolation level of the connection

[jdbc-core/tx-isolation!](#)

Set the transaction isolation level for the connection

[jdbc-core/auto-commit!](#)

Activate/Deactivate auto commit on a connection

[jdbc-core/commit!](#)

Commit the current transaction on the connection

[jdbc-core/rollback!](#)

Rollback the current transaction on the connection

[section](#)

[top](#)

jdbc-core/blob-bytebuf

(blob-bytebuf blob)

Returns the blob data as a bytebuf

([jdbc-core/blob-bytebuf](#) b)

SEE ALSO

[jdbc-core/blob?](#)

Returns true if val is a:java.sql.Blob

[jdbc-core/blob-length](#)

Returns the length of a blob

[jdbc-core/blob-in-stream](#)

Returns a :java.io.InputStream to read the blob data

[jdbc-core/blob-free](#)

Frees the Blob object and releases the resources the resources that it holds.

jdbc-core/blob-free

(blob-free blob)

Frees the Blob object and releases the resources the resources that it holds.

([jdbc-core/blob-free](#) b)

SEE ALSO

[jdbc-core/blob?](#)

Returns true if val is a:java.sql.Blob

[jdbc-core/blob-length](#)

Returns the length of a blob

[jdbc-core/blob-in-stream](#)

Returns a :java.io.InputStream to read the blob data

[jdbc-core/blob-bytebuf](#)

Returns the blob data as a bytebuf

jdbc-core/blob-in-stream

(blob-in-stream blob)

Returns a :java.io.InputStream to read the blob data

([jdbc-core/blob-in-stream](#) b)

SEE ALSO

[jdbc-core/blob?](#)

Returns true if val is a:java.sql.Blob

[jdbc-core/blob-length](#)

Returns the length of a blob

[jdbc-core/blob-bytebuf](#)

Returns the blob data as a bytebuf

[jdbc-core/blob-free](#)

Frees the Blob object and releases the resources the resources that it holds.

jdbc-core/blob-length

(blob-length blob)

Returns the length of a blob

([jdbc-core/blob-length](#) b)

SEE ALSO

[jdbc-core/blob?](#)
Returns true if val is a:java.sql.Blob

[jdbc-core/blob-in-stream](#)
Returns a :java.io.InputStream to read the blob data

[jdbc-core/blob-bytebuf](#)
Returns the blob data as a bytebuf

[jdbc-core/blob-free](#)
Frees the Blob object and releases the resources the resources that it holds.

[section](#) [top](#)

jdbc-core/blob?

(blob? val)

Returns true if val is a:java.sql.Blob

([jdbc-core/blob?](#) v)

SEE ALSO

[jdbc-core/blob-length](#)
Returns the length of a blob

[jdbc-core/blob-in-stream](#)
Returns a :java.io.InputStream to read the blob data

[jdbc-core/blob-bytebuf](#)
Returns the blob data as a bytebuf

[jdbc-core/blob-free](#)
Frees the Blob object and releases the resources the resources that it holds.

[section](#) [top](#)

jdbc-core/clob-free

(clob-free clob)

Frees the Clob object and releases the resources the resources that it holds.

([jdbc-core/clob-free](#) c)

SEE ALSO

[jdbc-core/clob?](#)
Returns true if val is a:java.sql.Clob

[jdbc-core/clob-length](#)
Returns the length of a clob

[jdbc-core/clob-reader](#)

Returns a `:java.io.Reader` to read the clob data

[section](#)

[top](#)

jdbc-core/clob-length

`(clob-length clob)`

Returns the length of a clob

[\(jdbc-core/clob-length](#) `vc`

SEE ALSO

[jdbc-core/clob?](#)

Returns true if `val` is a `java.sql.Clob`

[jdbc-core/clob-reader](#)

Returns a `:java.io.Reader` to read the clob data

[jdbc-core/clob-free](#)

Frees the Clob object and releases the resources the resources that it holds.

[section](#)

[top](#)

jdbc-core/clob-reader

`(clob-reader clob)`

Returns a `:java.io.Reader` to read the clob data

[\(jdbc-core/clob-reader](#) `c`)

SEE ALSO

[jdbc-core/clob?](#)

Returns true if `val` is a `java.sql.Clob`

[jdbc-core/clob-length](#)

Returns the length of a clob

[jdbc-core/clob-free](#)

Frees the Clob object and releases the resources the resources that it holds.

[section](#)

[top](#)

jdbc-core/clob?

`(clob? val)`

Returns true if `val` is a `java.sql.Clob`

[\(jdbc-core/clob?](#) `v`)

SEE ALSO

[jdbc-core/clob-length](#)

Returns the length of a clob

[jdbc-core/clob-reader](#)

Returns a `java.io.Reader` to read the clob data

[jdbc-core/clob-free](#)

Frees the Clob object and releases the resources the resources that it holds.

[section](#)

[top](#)

jdbc-core/closed?

`(closed? conn)`

Returns true the connections is closed else false.

[section](#)

[top](#)

jdbc-core/collect-result-set

`(collect-result-set rs)`

Collects data form a JDBC `java.sql.ResultSet` returns it as map with the column names and a vector of row values.

Row values may be of type:

- string
- boolean
- int
- long
- double
- decimal
- `java.sql.Clob`
- `java.sql.Blob`

```
{ :col-names ["name" "age"]  
  :rows [ ["john" 29]  
          ["mary" 32] ] }
```

```
(do  
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])  
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])  
  
  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")  
                stmt (jdbc/create-statement conn)  
                rs (jdbc/execute-query* stmt "SELECT * FROM foo")]  
    (jdbc/collect-result-set rs)))
```

SEE ALSO

[jdbc-core/render-query-result](#)

Renders the result from an execute-query in an ascii table format. Returns an ascii table formatted string.

[jdbc-core/print-query-result](#)

Prints the result from a execute-query in an ascii table format. Returns nil.

[jdbc-core/execute](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[jdbc-core/execute-query](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement or connection.

[jdbc-core/execute-update](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[section](#)[top](#)

jdbc-core/columns

```
(columns conn table)
```

List the columns of a database table

Example PostgreSQL Chinook database "genre" table:

```
[ { :name "genre_id"
    :type :INTEGER
    :size "10"
    :nullable? "NO"
    :auto-inc? "YES" }
  { :name "name"
    :type :VARCHAR
    :size "120"
    :nullable? "YES"
    :auto-inc? "NO" } ]
```

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "localhost" 5432 "chinook" "pg" "pg")
    (jdbc/columns conn "genre")]))
```

[section](#)[top](#)

jdbc-core/commit!

```
(commit! conn)
```

Commit the current transaction on the connection

```
(jdbc-core/auto-commit! conn false)
```

SEE ALSO

[jdbc-core/tx-isolation](#)

Returns transaction isolation level of the connection

[jdbc-core/tx-isolation!](#)

Set the transaction isolation level for the connection

[jdbc-core/auto-commit?](#)

Returns true if auto commit is enabled on the connection else false

[jdbc-core/auto-commit!](#)

Activate/Deactivate auto commit on a connection

[jdbc-core/rollback!](#)

Rollback the current transaction on the connection

[section](#)

[top](#)

jdbc-core/count-rows

```
(count-rows conn table)
```

Returns the row count of a table.

```
;; using a prepared statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")]
    (jdbc/count-rows conn "Albums")))
```

SEE ALSO

[jdbc-core/execute-query](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement or connection.

[section](#)

[top](#)

jdbc-core/create-database

```
(create-database conn database)
(create-database conn database force)
```

Creates a new database. If force flag is true drops the database first if it exists.

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "pg" "pg")]
    (jdbc/create-database conn "test"))

  (do
    (load-module :jdbc-core ['jdbc-core :as 'jdbc])
    (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
    (try-with [conn (jdbp/create-connection "pg" "pg")]
      (jdbc/create-database conn "test" true))))
```

SEE ALSO

[jdbc-core/drop-database](#)

Drops a database if it exists.

[section](#)

[top](#)

jdbc-core/create-statement

```
(create-statement conn)
```

Create a statement

```
(jdbc-core/create-statement conn)
```

SEE ALSO

[jdbc-core/prepare-statement](#)

Create a prepared statement

[section](#)

[top](#)

jdbc-core/drop-database

```
(drop-database conn database)
(drop-database conn database force)
```

Drops a database if it exists.

The force option will attempt to terminate all existing connections to the database.

Note: The force option is supported for PostgreSQL only and disconnects all connections prior to dropping the database!

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "pg" "pg")]
    (jdbc/drop-database conn "test")))
```

SEE ALSO

[jdbc-core/create-database](#)

Creates a new database. If force flag is true drops the database first if it exists.

[section](#)

[top](#)

jdbc-core/execute

```
(execute pstmt)
(execute stmt sql)
```

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

```
;; using a prepared statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
    pstmt (jdbc/prepare-statement conn "INSERT INTO foo VALUES(?,?)")]
    (jdbc/ps-int pstmt 1 1I))
```

```
(jdbc/ps-string pstmt 2 "Harry Potter")
(jdbc/execute pstmt)))

;; running an SQL statement on a JDBC statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
               stmt (jdbc/create-statement conn)]
    (jdbc/execute stmt "INSERT INTO foo VALUES(1, 'Harry Potter')")))
```

SEE ALSO

[jdbc-core/execute-query](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement or connection.

[jdbc-core/execute-query*](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[jdbc-core/execute-update](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[section](#)

[top](#)

jdbc-core/execute-query

```
(execute-query pstmt)
(execute-query stmt sql)
```

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement or connection.

Returns the JDBC `:java.sql.ResultSet` parsed as map with the column names and a vector of row values.

Row values may be of type:

- string
- boolean
- int
- long
- double
- decimal
- `:java.sql.Clob`
- `:java.sql.Blob`

```
{ :col-names ["name" "age"]
  :rows [ ["john" 29]
          ["mary" 32] ] }
```

```
;; using a prepared statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
               pstmt (jdbc/prepare-statement conn "SELECT * FROM foo")]
    (jdbc/execute-query pstmt)))
```

```
;; running an SQL statement on a JDBC statement
(do
```

```
(load-module :jdbc-core ['jdbc-core :as 'jdbc])
(load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

(try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
               stmt (jdbc/create-statement conn)]
  (-> (jdbc/execute-query stmt "SELECT * FROM foo")
      (jdbc-core/print-query-result)))
```

SEE ALSO

[jdbc-core/render-query-result](#)

Renders the result from an execute-query in an ascii table format. Returns an ascii table formatted string.

[jdbc-core/print-query-result](#)

Prints the result from a execute-query in an ascii table format. Returns nil.

[jdbc-core/execute](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[jdbc-core/execute-update](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[section](#)

[top](#)

jdbc-core/execute-query*

```
(execute-query* pstmt)
(execute-query* stmt sql)
```

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

Returns a JDBC `:java.sql.ResultSet`.

Note: The returned ResultSet has to be closed after use!

```
;; using a prepared statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
                pstmt (jdbc/prepare-statement conn "SELECT * FROM foo")
                rs (jdbc/execute-query* pstmt)]
    (while (jdbc/rs-next! rs)
      (println (jdbc/rs-string rs 1)))))

;; running an SQL statement on a JDBC statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
                stmt (jdbc/create-statement conn)
                rs (jdbc/execute-query* stmt "SELECT * FROM foo")]
    (while (jdbc/rs-next! rs)
      (println (jdbc/rs-string rs 1)))))

;; running an SQL statement on a JDBC statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
```

```
(try-with [conn (jdbc/create-connection "localhost" 5432 "test" "pg" "pg")
            stmt (jdbc/create-statement conn)
            rs (jdbc/execute-query* stmt "SELECT * FROM foo")]
  (jdbc/print-query-result rs)))
```

SEE ALSO

[jdbc-core/collect-result-set](#)

Collects data from a JDBC `java.sql.ResultSet` returns it as map with the column names and a vector of row values.

[jdbc-core/render-query-result](#)

Renders the result from an execute-query in an ascii table format. Returns an ascii table formatted string.

[jdbc-core/print-query-result](#)

Prints the result from a execute-query in an ascii table format. Returns nil.

[jdbc-core/execute](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[jdbc-core/execute-query](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement or connection.

[jdbc-core/execute-update](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[section](#)

[top](#)

jdbc-core/execute-update

```
(execute-update pstmt)
(execute-update stmt sql & options)
```

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

Returns an integer value that reports the number of rows affected by the SQL statement.

Options:

:gen-key Provide access to the generated keys.
The generated keys can be retrieved by a call to `(jdbc-core/generated-keys stmt)`.

Values:

`true` Return all generated keys
`["id"]` Return only the generated keys in the specified list

```
;; using a prepared statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbcp])

  (try-with [conn (jdbc/create-connection "localhost" 5432 "test" "pg" "pg")
              pstmt (jdbc/prepare-statement conn "INSERT INTO foo VALUES(?,?)"]
    (jdbc/ps-int pstmt 1 1I)
    (jdbc/ps-string pstmt 2 "Harry Potter")
    (jdbc/execute-update pstmt)))
```

```
;; running an SQL statement on a JDBC statement
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbcp])

  (try-with [conn (jdbc/create-connection "localhost" 5432 "test" "pg" "pg")
```

```
stmt (jdbc/create-statement conn)]
(jdbc/execute-update stmt "INSERT INTO foo VALUES(1, "Harry Potter")"))))
```

SEE ALSO

[jdbc-core/execute](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[jdbc-core/execute-query](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement or connection.

[jdbc-core/execute-query*](#)

Executes the SQL statement in prepared statement or executes an SQL statement on a JDBC statement.

[section](#)

[top](#)

jdbc-core/features

```
(features conn)
```

List the database' features

Example PostgreSQL features:

```
{ :supports-stored-procedures true
  :supports-full-outer-joins true
  :supports-savepoints true
  :supports-batch-updates true
  :supports-transactions true }
```

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "localhost" 5432 "pg" "pg")]
    (jdbc/features conn)))
```

[section](#)

[top](#)

jdbc-core/generated-keys

```
(generated-keys stmt)
```

Return a vector with generated keys.

```
(jdbc-core/generated-keys stmt)
```

SEE ALSO

[jdbc-core/create-statement](#)

Create a statement

[jdbc-core/prepare-statement](#)

Create a prepared statement

[section](#)

[top](#)

jdbc-core/meta-data

(meta-data conn)

List the meta data of a database

Example PostgreSQL meta data:

```
{ :product-name "PostgreSQL"
  :driver-name "PostgreSQL JDBC Driver"
  :driver-version "42.7.3"
  :product-version "16.2 (Debian 16.2-1.pgdg120+2)" }
```

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "localhost" 5432 "pg" "pg")]
    (jdbc/meta-data conn)))
```

[section](#)

[top](#)

jdbc-core/postgresql?

(postgresql? conn)

Returns true if 'conn' is a PostgreSQL connection else false.

[section](#)

[top](#)

jdbc-core/prepare-statement

(prepare-statement conn sql)
(prepare-statement conn sql & options)

Create a prepared statement

Options:

:gen-key Provide access to the generated keys.
The generated keys can be retrieved by a call to `(jdbc-core/generated-keys stmt)` .

Values:

`true` Return all generated keys
`["id"]` Return only the generated keys in the specified list

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
              pstmt (jdbc/prepare-statement conn "INSERT INTO foo VALUES(?,?)" ]
    (jdbc/ps-int pstmt 1 11)
    (jdbc/ps-string pstmt 2 "Harry Potter")
    (jdbc/execute pstmt)))
```

SEE ALSO

[jdbc-core/create-statement](#)

Create a statement

[section](#)

[top](#)

jdbc-core/print-query-result

```
(print-query-result data)
(print-query-result data max-col-width)
```

Prints the result from a `execute-query` in an ascii table format. Returns `nil`.

The functions accepts a JDBC `'java.sql.ResultSet'` or a collected result set as returned from [jdbc-core/collect-result-set](#).

max-col-width is limited to the range [5..200]

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
              stmt (jdbc/create-statement conn)
              rs   (jdbc/execute-query* stmt "SELECT * FROM foo")]
    (jdbc/print-query-result rs)))
```

SEE ALSO

[jdbc-core/collect-result-set](#)

Collects data form a JDBC `'java.sql.ResultSet'` returns it as map with the column names and a vector of row values.

[jdbc-core/render-query-result](#)

Renders the result from an `execute-query` in an ascii table format. Returns an ascii table formatted string.

[section](#)

[top](#)

jdbc-core/ps-blob

```
(ps-blob ps idx val)
```

Sets the prepared statment parameter to the given blob value.

The value may be a `bytebuf` or a `:java.io.InputStream`.

```
(jdbc-core/ps-decimal ps 1 (bytebuf [1 2 3]))

(jdbc-core/ps-decimal ps 1 (io/bytebuf-in-stream (bytebuf [1 2 3])))
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-boolean

```
(ps-boolean ps idx val)
```

Sets the prepared statment parameter to the given boolean value.

```
(jdbc-core/ps-boolean ps 1 true)
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-clear-parameters

```
(ps-clear-parameters ps)
```

Clears the prepared statment parameter.

```
(jdbc-core/ps-clear-parameters ps)
```

SEE ALSO

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-clob

```
(ps-clob ps idx val)
```

Sets the prepared statment parameter to the given clob value.

The value may be a `string` or a `:java.io.Reader` .

```
(jdbc-core/ps-decimal ps 1 "123456")
```

```
(jdbc-core/ps-decimal ps 1 (io/string-reader "123456"))
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-date

```
(ps-date ps idx val)
```

Sets the prepared statment parameter to the given date value.

```
(jdbc-core/ps-date ps 1 (time/date))
```

```
(jdbc-core/ps-date ps 1 (time/local-date 2020 1 1))
```

```
(jdbc-core/ps-date ps 1 (time/local-date-time 2020 1 1 14 0 0))
```

```
(jdbc-core/ps-date ps 1 (time/zoned-date-time "UTC" 2020 1 1 14 0 0))
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-decimal

```
(ps-decimal ps idx val)
```

Sets the prepared statment parameter to the given decimal value.

```
(jdbc-core/ps-decimal ps 1 3.1415M)
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-double

```
(ps-double ps idx val)
```

Sets the prepared statment parameter to the given double value.

```
(jdbc-core/ps-double ps 1 1.24)
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-float

```
(ps-float ps idx val)
```

Sets the prepared statment parameter to the given double value.

```
(jdbc-core/ps-float ps 1 1.24)
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-int

```
(ps-int ps idx val)
```

Sets the prepared statment parameter to the given int value.

```
(jdbc-core/ps-int ps 1 10I)
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-long

```
(ps-long ps idx val)
```

Sets the prepared statment parameter to the given long value.

```
(jdbc-core/ps-long ps 1 10)
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-string

```
(ps-string ps idx val)
```

Sets the prepared statment parameter to the given string value.

```
(jdbc-core/ps-string ps 1 "abcdef")
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[jdbc-core/ps-timestamp](#)

Sets the prepared statment parameter to the given timestamp value.

[section](#)

[top](#)

jdbc-core/ps-timestamp

```
(ps-timestamp ps idx val)
```

Sets the prepared statment parameter to the given timestamp value.

```
(jdbc-core/ps-timestamp ps 1 (time/date))
```

```
(jdbc-core/ps-timestamp ps 1 (time/local-date 2020 1 1))
```

```
(jdbc-core/ps-timestamp ps 1 (time/local-date-time 2020 1 1 14 0 0))
```

```
(jdbc-core/ps-timestamp ps 1 (time/zoned-date-time "UTC" 2020 1 1 14 0 0))
```

SEE ALSO

[jdbc-core/ps-clear-parameters](#)

Clears the prepared statment parameter.

[jdbc-core/ps-blob](#)

Sets the prepared statment parameter to the given blob value.

[jdbc-core/ps-clob](#)

Sets the prepared statment parameter to the given clob value.

[jdbc-core/ps-decimal](#)

Sets the prepared statment parameter to the given decimal value.

[jdbc-core/ps-boolean](#)

Sets the prepared statment parameter to the given boolean value.

[jdbc-core/ps-double](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-float](#)

Sets the prepared statment parameter to the given double value.

[jdbc-core/ps-int](#)

Sets the prepared statment parameter to the given int value.

[jdbc-core/ps-long](#)

Sets the prepared statment parameter to the given long value.

[jdbc-core/ps-string](#)

Sets the prepared statment parameter to the given string value.

[jdbc-core/ps-date](#)

Sets the prepared statment parameter to the given date value.

[section](#)[top](#)

jdbc-core/render-query-result

```
(render-query-result data)
```

```
(render-query-result data max-col-width)
```

Renders the result from an `execute-query` in an ascii table format. Returns an ascii table formatted string.

The functions accepts a JDBC `'java.sql.ResultSet'` or a collected result set as returned from `jdbc-core/collect-result-set`.

max-col-width is limited to the range [15..200]

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")
               stmt (jdbc/create-statement conn)
               rs (jdbc/execute-query* stmt "SELECT * FROM foo")]
    (jdbc/render-query-result rs)))
```

SEE ALSO

[jdbc-core/collect-result-set](#)

Collects data form a JDBC `java.sql.ResultSet` returns it as map with the column names and a vector of row values.

[jdbc-core/print-query-result](#)

Prints the result from a `execute-query` in an ascii table format. Returns nil.

[section](#)[top](#)

jdbc-core/rollback!

```
(rollback! conn)
```

Rollback the current transaction on the connection

```
(jdbc-core/rollback! conn)
```

SEE ALSO

[jdbc-core/tx-isolation](#)

Returns transaction isolation level of the connection

[jdbc-core/tx-isolation!](#)

Set the transaction isolation level for the connection

[jdbc-core/auto-commit?](#)

Returns true if auto commit is enabled on the connection else false

[jdbc-core/auto-commit!](#)

Activate/Deactivate auto commit on a connection

[jdbc-core/commit!](#)

Commit the current transaction on the connection

[section](#)

[top](#)

jdbc-core/rs-blob

```
(rs-blob rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.sql.Blob`.

```
(jdbc-core/rs-blob rs 1)
```

```
(jdbc-core/rs-blob rs "data")
```

SEE ALSO

[jdbc-core/blob-length](#)

Returns the length of a blob

[jdbc-core/blob-in-stream](#)

Returns a `:java.io.InputStream` to read the blob data

[jdbc-core/blob-bytebuf](#)

Returns the blob data as a bytebuf

[jdbc-core/blob-free](#)

Frees the Blob object and releases the resources the resources that it holds.

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/rs-boolean

```
(rs-boolean rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `boolean` .

```
(jdbc-core/rs-boolean rs 1)
```

```
(jdbc-core/rs-boolean rs "openBill")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/rs-clob

```
(rs-clob rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

```
(jdbc-core/rs-clob rs 1)
```

```
(jdbc-core/rs-clob rs "remarks")
```

SEE ALSO

[jdbc-core/clob-length](#)

Returns the length of a clob

[jdbc-core/clob-reader](#)

Returns a `java.io.Reader` to read the clob data

[jdbc-core/clob-free](#)

Frees the Clob object and releases the resources the resources that it holds.

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/rs-date

```
(rs-date rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalDate` .

```
(jdbc-core/rs-date rs 1)
```

```
(jdbc-core/rs-date rs "birthDate")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/rs-decimal

```
(rs-decimal rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `decimal` .

```
(jdbc-core/rs-decimal rs 1)
```

```
(jdbc-core/rs-decimal rs "billAmount")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.time.LocalDate.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.time.LocalTime.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.time.LocalDateTime.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.sql.Clob.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.sql.Blob.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)

[top](#)

jdbc-core/rs-double

```
(rs-double rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `double`.

```
(jdbc-core/rs-double rs 1)
```

```
(jdbc-core/rs-double rs "weight")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)

[top](#)

jdbc-core/rs-first!

```
(rs-first! rs)
```

Moves the cursor to the first row in this ResultSet object.

Returns true if the cursor is on a valid row; false if there are no rows in the result set

```
(jdbc-core/rs-first! rs)
```

SEE ALSO

[jdbc-core/rs-next!](#)

Moves the cursor forward one row from its current position. A ResultSet cursor is initially positioned before the first row; the first ...

[jdbc-core/rs-last!](#)

Moves the cursor to the last row in this ResultSet object.

[section](#)

[top](#)

jdbc-core/rs-float

```
(rs-float rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `double`.

```
(jdbc-core/rs-float rs 1)
```

```
(jdbc-core/rs-float rs "weight")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/rs-int

```
(rs-int rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `int`.

```
(jdbc-core/rs-int rs 1)
```

```
(jdbc-core/rs-int rs "age")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/rs-last!

```
(rs-last! rs)
```

Moves the cursor to the last row in this ResultSet object.

Returns true if the cursor is on a valid row; false if there are no rows in the result set

```
(jdbc-core/rs-last! rs)
```

SEE ALSO

[jdbc-core/rs-first!](#)

Moves the cursor to the first row in this ResultSet object.

[jdbc-core/rs-next!](#)

Moves the cursor forward one row from its current position. A ResultSet cursor is initially positioned before the first row; the first ...

[section](#)[top](#)

jdbc-core/rs-long

```
(rs-long rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `long`.

```
(jdbc-core/rs-long rs 1)
```

```
(jdbc-core/rs-long rs "age")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to string, boolean, int, long, double, ...

[section](#)

[top](#)

jdbc-core/rs-next!

```
(rs-next! rs)
```

Moves the cursor forward one row from its current position. A ResultSet cursor is initially positioned before the first row; the first call to the method `next` makes the first row the current row; the second call makes the second row the current row, and so on.

Returns true if the new current row is valid; false if there are no more rows

```
(jdbc-core/rs-next! rs)
```

SEE ALSO

[jdbc-core/rs-first!](#)

Moves the cursor to the first row in this ResultSet object.

[jdbc-core/rs-last!](#)

Moves the cursor to the last row in this ResultSet object.

[section](#)

[top](#)

jdbc-core/rs-string

```
(rs-string rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `string`.

```
(jdbc-core/rs-string rs 1)
```

```
(jdbc-core/rs-string rs "firstName")
```

SEE ALSO

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalDate`.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalTime`.

[jdbc-core/rs-timestamp](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalDateTime`.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.sql.Clob`.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.sql.Blob`.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted to `string`, `boolean`, `int`, `long`, `double`, ...

[section](#)

[top](#)

jdbc-core/rs-timestamp

```
(rs-timestamp rs name-or-index)
```

Retrieves the value of the designated column in the current row of the ResultSet object as a `:java.time.LocalDateTime`.

```
(jdbc-core/rs-timestamp rs 1)
```

```
(jdbc-core/rs-timestamp rs "createdAt")
```

SEE ALSO

[jdbc-core/rs-string](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a string.

[jdbc-core/rs-boolean](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a boolean.

[jdbc-core/rs-int](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a int.

[jdbc-core/rs-long](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a long.

[jdbc-core/rs-float](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-double](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a double.

[jdbc-core/rs-decimal](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a decimal.

[jdbc-core/rs-date](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.time.LocalDate.

[jdbc-core/rs-time](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.time.LocalTime.

[jdbc-core/rs-clob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.sql.Clob.

[jdbc-core/rs-blob](#)

Retrieves the value of the designated column in the current row of the ResultSet object as a :java.sql.Blob.

[jdbc-core/rs-object](#)

Retrieves the value of the designated column in the current row of the ResultSet object converted toString, boolean, int, long, double, ...

[section](#)[top](#)

jdbc-core/schemas

```
(schemas conn)
```

List the schemas of a database

Example PostgreSQL schemas:

```
[ { :schem "information_schema"
  :catalog nil
}
{ :schem "pg_catalog"
  :catalog nil
}
{ :schem "public"
  :catalog nil
} ]
```

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "localhost" 5432 "pg" "pg")]
    (jdbc/schemas conn)))
```

[section](#)[top](#)

jdbc-core/tables

```
(tables conn)
```

List the tables of a database

Example PostgreSQL Chinook database:

```
[ "album"
  "artist"
  "customer"
  "employee"
  "genre"
  "invoice"
  "invoice_line"
  "media_type"
  "playlist"
  "playlist_track"
  "track" ]
```

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
  (try-with [conn (jdbp/create-connection "localhost" 5432 "chinook" "pg" "pg")]
    (jdbc/tables conn)))
```

[section](#)

[top](#)

jdbc-core/tx-isolation

```
(tx-isolation conn)
```

Returns transaction isolation level of the connection

Levels:

- `:tx-none`
- `:tx-read-committed`
- `:tx-read-uncommitted`
- `:tx-repeatable-read`
- `:tx-serializable`

```
(jdbc-core/tx-isolation conn)
```

SEE ALSO

[jdbc-core/tx-isolation!](#)

Set the transaction isolation level for the connection

[jdbc-core/auto-commit?](#)

Returns true if auto commit is enabled on the connection else false

[jdbc-core/auto-commit!](#)

Activate/Deactivate auto commit on a connection

[jdbc-core/commit!](#)

Commit the current transaction on the connection

[jdbc-core/rollback!](#)

Rollback the current transaction on the connection

[section](#)

[top](#)

jdbc-core/tx-isolation!

```
(tx-isolation! conn level)
```

Set the transaction isolation level for the connection

Levels:

- `:tx-none`
- `:tx-read-committed`
- `:tx-read-uncommitted`
- `:tx-repeatable-read`
- `:tx-serializable`

```
(jdbc-core/tx-isolation! conn :tx-repeatable-read)
```

SEE ALSO

[jdbc-core/tx-isolation](#)

Returns transaction isolation level of the connection

[jdbc-core/auto-commit?](#)

Returns true if auto commit is enabled on the connection else false

[jdbc-core/auto-commit!](#)

Activate/Deactivate auto commit on a connection

[jdbc-core/commit!](#)

Commit the current transaction on the connection

[jdbc-core/rollback!](#)

Rollback the current transaction on the connection

[section](#)

[top](#)

jdbc-core/with-conn

```
(with-conn conn & forms)
```

Sets the thread local var `conn` to the passed connection, wraps the connection in a `try-with` form to close the connection automatically at the end of the template and runs the forms.

The forms have access to the connection via the `conn` thread local var.

While thread local vars may work fine (most ORMs like *Hibernate* or *JPA* rely on thread local vars), using Venice *Components* for connection management give a cleaner and more functional architecture.

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (jdbc/with-conn [conn (jdbp/create-connection "localhost" 5432
                                                "test" "pg" "pg")])
  (try-with [stmt (jdbc/create-statement *conn*)])
  (-> (jdbc/execute-update stmt "INSERT INTO foo VALUES(100,3.1456)")
      (jdbc-core/print-query-result))))
```


SEE ALSO

[jdbc-core/with-tx](#)

Runs the forms within a JDBC transaction, commits the transaction at the end or rolls the transaction back if the forms throw an exception.

[section](#)

[top](#)

jdbc-core/with-tx

```
(with-tx conn & forms)
```

Runs the forms within a JDBC transaction, commits the transaction at the end or rolls the transaction back if the forms throw an exception.

Restores the auto-commit mode on the connection at the end of the successful or failed transaction.

Throws a `:com.github.jlangch.venice.TransactionException` after rolling back.

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")]
    (jdbc/with-tx conn
      (try-with [stmt (jdbc/create-statement conn)]
        (-> (jdbc/execute-update stmt "INSERT INTO foo VALUES(100,3.1415)")
          (jdbc-core/print-query-result))))))
```

SEE ALSO

[jdbc-core/with-conn](#)

Sets the thread local var `conn` to the passed connection, wraps the connection in a try-with form to close the connection automatically ...

[section](#)

[top](#)

jdbc-postgresql/create-connection

```
(create-connection user password)
(create-connection host port user password)
(create-connection host port database user password)
(create-connection host port database user password properties)
```

Creates a PostgreSQL connection.

Arguments:

<i>user</i>	A mandatory ser name
<i>password</i>	A mandatory password
<i>host</i>	An optional host. Defaults to "localhost"
<i>port</i>	An optional port. Defaults to 5432
<i>database</i>	A mandatory database name
<i>properties</i>	Optional properties (a map). E.g.: { "ssl" "true", "options" "-c statement_timeout=90000" }

```
(do
  (load-module :jdbc-core ['jdbc-core :as 'jdbc])
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])
```

```
(try-with [conn (jdbc/create-connection "localhost" 5432 "test" "pg" "pg")]
  (-> (jdbc/execute-query conn "SELECT * FROM mytable WHERE foo = 500")
    (jdbc/print-query-result))))
```

SEE ALSO

[jdbc-core/create-database](#)

Creates a new database. If force flag is true drops the database first if it exists.

[jdbc-core/drop-database](#)

Drops a database if it exists.

[section](#)

[top](#)

jdbc-postgresql/describe-table

```
(describe-table conn table & options)
```

Describe the schema of a table.

Options:

:mode In `:print` mode prints the table description, in `:data` mode returns the description as a data structure. Defaults to `:print`.

Example PostgreSQL Chinook database "album" table:

column_name	data_type	character_maximum_length	is_nullable	column_default
album_id	integer	<null>	NO	<null>
artist_id	integer	<null>	NO	<null>
title	character varying	160	NO	<null>

```
(do
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbc])

  (try-with [conn (jdbc/create-connection "localhost" 5432 "test" "pg" "pg")]
    (jdbc/describe-table conn "album")))
```

SEE ALSO

[jdbc-postgresql/foreign-key-constraints](#)

List the foreign key constraints in a database

[section](#)

[top](#)

jdbc-postgresql/foreign-key-constraints

```
(foreign-key-constraints conn & options)
```

List the foreign key constraints in a database

Options:

:mode In `:print` mode prints the foreign key constraints, in `:data` mode returns the constraints as a data structure. Defaults to `:`

```
print.
```

Example PostgreSQL Chinook database foreign key constraints:

table_name	foreign_key	pg_get_constraintdef
album	album_artist_id_fkey	FOREIGN KEY (artist_id) REFERENCES artist(artist_id)
customer	customer_support_rep_id_fkey	FOREIGN KEY (support_rep_id) REFERENCES employee(employee_id)
employee	employee_reports_to_fkey	FOREIGN KEY (reports_to) REFERENCES employee(employee_id)
invoice	invoice_customer_id_fkey	FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
invoice_line	invoice_line_invoice_id_fkey	FOREIGN KEY (invoice_id) REFERENCES invoice(invoice_id)
invoice_line	invoice_line_track_id_fkey	FOREIGN KEY (track_id) REFERENCES track(track_id)
playlist_track	playlist_track_playlist_id_fkey	FOREIGN KEY (playlist_id) REFERENCES playlist(playlist_id)
playlist_track	playlist_track_track_id_fkey	FOREIGN KEY (track_id) REFERENCES track(track_id)
track	track_album_id_fkey	FOREIGN KEY (album_id) REFERENCES album(album_id)
track	track_genre_id_fkey	FOREIGN KEY (genre_id) REFERENCES genre(genre_id)
track	track_media_type_id_fkey	FOREIGN KEY (media_type_id) REFERENCES media_type(media_type_id)

```
(do
  (load-module :jdbc-postgresql ['jdbc-postgresql :as 'jdbp])

  (try-with [conn (jdbp/create-connection "localhost" 5432 "test" "pg" "pg")]
    (jdbp/foreign-key-constraints conn)))
```

SEE ALSO

[jdbc-postgresql/describe-table](#)

Describe the schema of a table.

[section](#)

[top](#)

json/pretty-print

```
(json/pretty-print s & options)
```

Pretty prints a JSON string

Options:

:indent s The indent for indented output. Must contain spaces or tabs only. Defaults to two spaces.

```
(-> (json/write-str {:a 100 :b 100 :c [1 2 3]}))
(json/pretty-print)
(println))

{
  "a": 100,
  "b": 100,
  "c": [1,2,3]
}
=> nil

(-> (json/write-str {:a 100 :b 100 :c [1 2 {:x 7 :y 8}] :d {:z 9}})
  (json/pretty-print :indent "  ")
  (println))

{
  "a": 100,
  "b": 100,
  "c": [1,2,{
```

```
      "x": 7,
      "y": 8
    }],
    "d": {
      "z": 9
    }
  }
=> nil
```

SEE ALSO

[json/write-str](#)

Writes the val to a JSON string.

[json/read-str](#)

Reads a JSON string and returns it as a Venice datatype.

[json/spit](#)

Spits the JSON converted val to the output.

[json/slurp](#)

Slurps a JSON data from a source and returns it as a Venice data.

[section](#)

[top](#)

json/read-str

([json/read-str](#) s & options)

Reads a JSON string and returns it as a Venice datatype.

Options:

- :key-fn fn Single argument function called on JSON property names; return value will replace the property names in the output. Default is 'identity', use 'keyword' to get keyword properties.
- :value-fn fn Function to transform values in JSON objects in the output. For each JSON property, value-fn is called with two arguments: the property name (transformed by key-fn) and the value. The return value of value-fn will replace the value in the output. The default value-fn returns the value unchanged.
- :decimal b If true use BigDecimal for decimal numbers instead of Double. Default is false.

```
(json/read-str (json/write-str {:a 100 :b 100}))
=> {"a" 100 "b" 100}
```

```
(json/read-str (json/write-str {:a 100 :b 100}) :key-fn keyword)
=> {:a 100 :b 100}
```

```
(json/read-str (json/write-str {:a 100 :b 100})
  :value-fn (fn [k v] (if (== "a" k) (inc v) v)))
=> {"a" 101 "b" 100}
```

SEE ALSO

[json/write-str](#)

Writes the val to a JSON string.

[json/spit](#)

Spits the JSON converted val to the output.

[json/slurp](#)

Slurps a JSON data from a source and returns it as a Venice data.

[json/pretty-print](#)

json/slurp

(json/slurp source & options)

Slurps a JSON data from a source and returns it as a Venice data.

The source may be a:

- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.Path`
- `java.io.InputStream`
- `java.io.Reader`

Options:

:key-fn fn Single-argument function called on JSON property names; return value will replace the property names in the output. Default is 'identity', use 'keyword' to get keyword properties.

:value-fn fn Function to transform values in JSON objects in the output. For each JSON property, value-fn is called with two arguments: the property name (transformed by key-fn) and the value. The return value of value-fn will replace the value in the output. The default value-fn returns the value unchanged.

:decimal b If true use BigDecimal for decimal numbers instead of Double. Default is false.

:encoding e e.g :encoding utf-8, defaults to :utf-8

```
(let [json (json/write-str {:a 100 :b 100 :c 1.233})]
  (try-with [in (io/string-reader json)]
    (pr-str (json/slurp in))))
=> "{ \"a\" 100 \"b\" 100 \"c\" 1.233}"

(let [json (json/write-str {:a 100 :b 100 :c 1.233})]
  (try-with [in (io/string-reader json)]
    (pr-str (json/slurp in :decimal true :key-fn keyword))))
=> "{:a 100 :b 100 :c 1.233M}"
```

SEE ALSO

[json/write-str](#)

Writes the val to a JSON string.

[json/read-str](#)

Reads a JSON string and returns it as a Venice datatype.

[json/spit](#)

Spits the JSON converted val to the output.

[json/pretty-print](#)

Pretty prints a JSON string

json/spit

(json/spit out val & options)

Spits the JSON converted val to the output.

The out may be a:

- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.Path`
- `java.io.OutputStream`
- `java.io.Writer`

Options:

:pretty b Enables/disables pretty printing. Defaults to false.
:decimal-as-double b If true emit a decimal as double else as string. Defaults to false.
:encoding e e.g :encoding :utf-8, defaults to :utf-8

```
(try-with [out (io/bytebuf-out-stream)]  
  (json/spit out {:a 100 :b 100 :c [10 20 30]}))  
(flush out)  
(bytebuf-to-string @out :utf-8))  
=> "{\"a\":100,\"b\":100,\"c\":[10,20,30]}"
```

SEE ALSO

[json/write-str](#)

Writes the val to a JSON string.

[json/read-str](#)

Reads a JSON string and returns it as a Venice datatype.

[json/slurp](#)

Slurps a JSON data from a source and returns it as a Venice data.

[json/pretty-print](#)

Pretty prints a JSON string

[section](#)

[top](#)

json/write-str

(json/write-str val & options)

Writes the val to a JSON string.

Options:

:pretty b Enables/disables pretty printing. Defaults to false.
:decimal-as-double b If true emit a decimal as double else as string. Defaults to false.

```
(json/write-str {:a 100 :b 100})  
=> "{\"a\":100,\"b\":100}"  
  
(json/write-str {:a 100 :b 100} :pretty true)  
=> "{\n  \"a\": 100,\n  \"b\": 100\n}"
```

SEE ALSO

[json/read-str](#)

Reads a JSON string and returns it as a Venice datatype.

[json/spit](#)

Spits the JSON converted val to the output.

json/slurp

Slurps a JSON data from a source and returns it as a Venice data.

json/pretty-print

Pretty prints a JSON string

[section](#)

[top](#)

json/lazy-seq-slurper

(jsonl/lazy-seq-slurper in & options)

Returns a lazy sequence of the parsed JSON line strings from the input 'in'.

'in' may be a:

- `java.io.InputStream`
- `java.io.Reader`

Note: The caller is responsible for closing the in stream/reader!

Options:

:key-fn fn	Single argument function called on JSON property names; return value will replace the property names in the output. Default is 'identity', use 'keyword' to get keyword properties.
:value-fn fn	Function to transform values in JSON objects in the output. For each JSON property, value-fn is called with two arguments: the property name (transformed by key-fn) and the value. The return value of value-fn will replace the value in the output. The default value-fn returns the value unchanged.
:decimal b	If true use BigDecimal for decimal numbers instead of Double. Default is false.
:filter-fn fn	Single argument function called on every read value from a JSON line. If it returns true the value will be kept otherwise it will be skipped
:encoding e	e.g :encoding :utf-8, defaults to :utf-8

`jsonl/lazy-seq-slurper` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
;; use a lazy sequence to read the JSON lines data
(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spit wr [{:a 100 :b 200} {:a 101 :b 201} {:a 102 :b 202}])
      (flush wr))
    (try-with [rd (io/buffered-reader file)]
      (let [slurper (jsonl/lazy-seq-slurper rd :key-fn keyword)]
        ;; realize the lazy sequence
        (doall slurper))))))
```

```
=> ({:a 100 :b 200} {:a 101 :b 201} {:a 102 :b 202})
```

```
;; use a transducer to efficiently map and filter the JSON lines data
(do
  (load-module :jsonl)

  (defn test-data []
    (try-with [sw (io/string-writer)]
      (println sw (json/write-str {:a 100 :b 200 :c 300}))
      (println sw (json/write-str {:a 101 :b 201 :c 301}))
      (println sw (json/write-str {:a 100 :b 202 :c 302}))
      (flush sw)))
```

```

@sw))

(def xform (comp (map #(dissoc % :c))
                 (map #(update % :b (fn [x] (+ x 5))))
                 (filter #(= 100 (:a %)))))

(let [json (test-data)]
  (try-with [rd (io/buffered-reader json)]
    (let [slurper (jsonl/lazy-seq-slurper rd :key-fn keyword)]
      ;; transduce the lazy sequence
      (pr-str (transduce xform conj slurper))))))
=> "[{:a 100 :b 205} {:a 100 :b 207}]"

```

SEE ALSO

[jsonl/slurp](#)

Slurps a list of JSON line strings from the input 'in' and returns it as a list of Venice data types.

[jsonl/read-str](#)

Reads a JSON line string 's' and returns it as a Venice data type.

[section](#)
[top](#)

jsonl/read-str

(jsonl/read-str s & options)

Reads a JSON line string 's' and returns it as a Venice data type.

Options:

- :key-fn fn Single argument function called on JSON property names; return value will replace the property names in the output. Default is 'identity', use 'keyword' to get keyword properties.
- :value-fn fn Function to transform values in JSON objects in the output. For each JSON property, value-fn is called with two arguments: the property name (transformed by key-fn) and the value. The return value of value-fn will replace the value in the output. The default value-fn returns the value unchanged.
- :decimal b If true use BigDecimal for decimal numbers instead of Double. Default is false.

```

(do
  (load-module :jsonl)
  (let [json (jsonl/write-str {:a 100 :b 200})]
    (jsonl/read-str json :key-fn keyword)))
=> ({:a 100 :b 200})

(do
  (load-module :jsonl)
  (let [json (jsonl/write-str [{:a 100 :b 200} {:a 100 :b 200}])]
    (jsonl/read-str json :key-fn keyword)))
=> ({:a 100 :b 200} {:a 100 :b 200})

(do
  (load-module :jsonl)
  (try-with [sw (io/string-writer)]
    (println sw (jsonl/write-str {:a 100 :b 200})))
    (println sw (jsonl/write-str {:a 101 :b 201})))
    (println sw (jsonl/write-str {:a 102 :b 202})))
    (flush sw)
    (let [json @sw]
      (jsonl/read-str json :key-fn keyword))))
=> ({:a 100 :b 200} {:a 101 :b 201} {:a 102 :b 202})

```


SEE ALSO

[jsonl/write-str](#)

Writes the value 'val' to a JSON lines string.

[section](#)

[top](#)

jsonl/slurp

(jsonl/slurp in & options)

Slurps a list of JSON line strings from the input 'in' and returns it as a list of Venice data types.

'in' may be a:

- string
- bytebuf
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.file.Path`
- `java.io.InputStream`
- `java.io.Reader`

Note: The caller is responsible for closing the in stream/reader!

Options:

- :key-fn fn Single argument function called on JSON property names; return value will replace the property names in the output. Default is 'identity', use 'keyword' to get keyword properties.
- :value-fn fn Function to transform values in JSON objects in the output. For each JSON property, value-fn is called with two arguments: the property name (transformed by key-fn) and the value. The return value of value-fn will replace the value in the output. The default value-fn returns the value unchanged.
- :decimal b If true use BigDecimal for decimal numbers instead of Double. Default is false.
- :filter-fn fn Single argument function called on every read value from a JSON line. If it returns true the value will be kept otherwise it will be skipped. The filter is applied after the 'key-fn' and the 'value-fn' have been applied to the line data value.
- :encoding e e.g :encoding :utf-8, defaults to :utf-8

`jsonl/slurp` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spit wr [{:a 100 :b 200} {:a 101 :b 201} {:a 102 :b 202}])
      (flush wr))
    (try-with [rd (io/buffered-reader file)]
      (jsonl/slurp rd :key-fn keyword))))
=> ({:a 100 :b 200} {:a 101 :b 201} {:a 102 :b 202})
```

;; slurp JSON Lines applying mapping functions and a filter on the lines

```
(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")
        now (time/local-date-time)]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spit wr [{:a 100 :b (time/plus now :days 1) :c 10.12M}
                     {:a 101 :b (time/plus now :days 2) :c 20.12M}
                     {:a 100 :b (time/plus now :days 3) :c 30.12M}]))
```

```

    (flush wr))
  (try-with [rd (io/buffered-reader file)]
    (jsonl/slurp rd :key-fn keyword
      :value-fn (fn [k v]
        (case k
          :b (time/local-date-time-parse v :iso)
          :c (decimal v)
          v))
      :filter-fn #(= 100 (:a %)))))
=> ({:a 100 :b 2026-06-05T17:21:22.276 :c 10.12M} {:a 100 :b 2026-06-07T17:21:22.276 :c 30.12M})

```

SEE ALSO

[jsonl/read-str](#)

Reads a JSON line string 's' and returns it as a Venice data type.

[jsonl/lazy-seq-slurper](#)

Returns a lazy sequence of the parsed JSON line strings from the input 'in'.

[section](#)

[top](#)

jsonl/spit

```
(jsonl/spit out val & options)
```

Spits the JSON Lines converted value 'val' to the output 'out'.

The 'out' may be a:

- `java.io.File`, e.g. `(io/file "/temp/foo.json")`
- `java.nio.Path`
- `java.io.OutputStream`
- `java.io.Writer`

Note: The caller is responsible for closing the out stream/writer!

Any reasonable Venice value like string, integer, long, double, decimal, boolean, list, vector, set, or map can be passed. Sequences like list or vector are converted to multiple JSON lines, one line for each value in the sequence. All other types are converted to a single JSON line.

Options:

:decimal-as-double b If true emit a decimal as double else as string. Defaults to false.

:append true/false e.g.: `:append true`, defaults to false

:encoding e e.g :encoding utf-8, defaults to :utf-8

`jsonl/spit` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```

(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spit wr [{:a 100 :b 200} {:a 101 :b 201} {:a 102 :b 202}])
      (flush wr))
    ;; print the json lines data
    (println (io/slurp file :encoding :utf-8))))
{"a":100,"b":200}
{"a":101,"b":201}
{"a":102,"b":202}
=> nil

```

```
;; spit a list of json lines (linefeeds are added implicitly )
(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spit wr [{"a" 100, "b" 200}
                     {"a" 101, "b" 201}
                     {"a" 102, "b" 202}]))
    (flush wr))
    ;; print the json lines data
    (println (io/slurp file :encoding :utf-8))))
{"a":100,"b":200}
{"a":101,"b":201}
{"a":102,"b":202}
=> nil

;; spit a list of json lines line by line (linefeeds must be added explicitly)
(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spit wr {"a" 100, "b" 200})
      (println wr)
      (jsonl/spit wr {"a" 101, "b" 201})
      (println wr)
      (jsonl/spit wr {"a" 102, "b" 202})
      (flush wr))
    ;; print the json lines data
    (println (io/slurp file :encoding :utf-8))))
{"a":100,"b":200}
{"a":101,"b":201}
{"a":102,"b":202}
=> nil
```

SEE ALSO

[jsonl/write-str](#)

Writes the value 'val' to a JSON lines string.

[jsonl/slurp](#)

Slurps a list of JSON line strings from the input 'in' and returns it as a list of Venice data types.

[section](#)

[top](#)

jsonl/spitln

```
(jsonl/spitln out val & options)
```

Spits the JSON Lines converted value 'val' to the output 'out' and adds a new line after the last emitted line.

This function is useful when lines are spitted to a stream/writer line by line.

The 'out' may be a:

- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.Path`
- `java.io.OutputStream`
- `java.io.Writer`

Note: The caller is responsible for closing the out stream/writer!

Any reasonable Venice value like string, integer, long, double, decimal, boolean, list, vector, set, or map can be passed. Sequences like list or vector are converted to multiple JSON lines, one line for each value in the sequence. All other types are converted to a single JSON line.

Options:

:decimal-as-double b If true emit a decimal as double else as string. Defaults to false.

:append true/false e.g.: `:append true`, defaults to false

:encoding e e.g. `:encoding :utf-8`, defaults to `:utf-8`

`jsonl/spitln` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
;; spit a list of json lines line by line
(do
  (load-module :jsonl)
  (let [file (io/temp-file "data-" ".jsonl")]
    (io/delete-file-on-exit file)
    (try-with [wr (io/buffered-writer file)]
      (jsonl/spitln wr {"a" 100, "b" 200})
      (jsonl/spitln wr {"a" 101, "b" 201})
      (jsonl/spit wr {"a" 102, "b" 202})      ;; last line no LF
      (flush wr))
    ;; print the json lines from the written file
    (println (io/slurp file :encoding :utf-8))))
{"a":100,"b":200}
{"a":101,"b":201}
{"a":102,"b":202}
=> nil
```

SEE ALSO

[jsonl/write-str](#)

Writes the value 'val' to a JSON lines string.

[jsonl/slurp](#)

Slurps a list of JSON line strings from the input 'in' and returns it as a list of Venice data types.

[section](#)

[top](#)

jsonl/write-str

`(jsonl/write-str val & options)`

Writes the value 'val' to a JSON lines string.

Any reasonable Venice value like string, integer, long, double, decimal, boolean, list, vector, set, or map can be passed. Sequences like list or vector are converted to multiple JSON lines, one line for each value in the sequence. All other types are converted to a single JSON line.

Options:

:decimal-as-double b If true emit a decimal as double else as string. Defaults to false

```
(do
  (load-module :jsonl)
  (println (jsonl/write-str {:a 100 :b 200})))
{"a":100,"b":200}
=> nil

(do
  (load-module :jsonl)
  (println (jsonl/write-str [{:a 100 :b 200}])))
```

```
      {:a 101 :b 201}
      {:a 102 :b 202}])))

{"a":100,"b":200}
{"a":101,"b":201}
{"a":102,"b":202}
=> nil
```

SEE ALSO

[jsonl/spit](#)

Spits the JSON Lines converted value 'val' to the output 'out'.

[jsonl/read-str](#)

Reads a JSON line string 's' and returns it as a Venice data type.

[section](#)

[top](#)

jtokkit/count-tokens

(count-tokens encoding text)

Encodes the given text into a list of token ids and returns the number of tokens.

The argument 'encoding' may be an encoding type like `:0200K_BASE` or a model type like `:GPT_40`

```
(do
  (load-module :jtokkit ['jtokkit :as 'jt])
  ;; e.g. model types: :GPT_4, :GPT_40, :GPT_40_MINI
  (-> (jt/encoding :GPT_40)
    (jt/count-tokens "hello world")))

(do
  (load-module :jtokkit ['jtokkit :as 'jt])
  ;; e.g. encoding types: :CL100K_BASE, :0200K_BASE
  (-> (jt/encoding :0200K_BASE)
    (jt/count-tokens "hello world")))
```

SEE ALSO

[jtokkit/encoding](#)

Returns the encoding (`:com.knuddels.jtokkit.api.Encoding`) object for the given encoding or model type.

[jtokkit/encoding-types](#)

Returns the defined encoding types. Actually from the enum type `:com.knuddels.jtokkit.api.EncodingType`.

[jtokkit/model-types](#)

Returns the defined model types. Actually from the enum type `:com.knuddels.jtokkit.api.ModelType`.

[jtokkit/encode](#)

Encodes the given text into a list of token ids.

[section](#)

[top](#)

jtokkit/encode

(encode encoding text)

Encodes the given text into a list of token ids.

The argument 'encoding' may be an encoding type like `:0200K_BASE` or a model type like `:GPT_40`

```
(do
  (load-module :jtokkit ['jtokkit :as 'jt])
  ;; e.g. model types: :GPT_4, :GPT_40, :GPT_40_MINI
  (-> (jt/encoding :GPT_40)
    (jt/encode "hello world")))

(do
  (load-module :jtokkit ['jtokkit :as 'jt])
  ;; e.g. encoding types: :CL100K_BASE, :0200K_BASE
  (-> (jt/encoding :0200K_BASE)
    (jt/encode "hello world")))
```

SEE ALSO

[jtokkit/encoding](#)

Returns the encoding (`:com.knuddels.jtokkit.api.Encoding`) object for the given encoding or model type.

[jtokkit/encoding-types](#)

Returns the defined encoding types. Actually from the enum type `:com.knuddels.jtokkit.api.EncodingType`.

[jtokkit/model-types](#)

Returns the defined model types. Actually from the enum type `:com.knuddels.jtokkit.api.ModelType`.

[jtokkit/count-tokens](#)

Encodes the given text into a list of token ids and returns the number of tokens.

[section](#)

[top](#)

jtokkit/encoding

(encoding type)

Returns the encoding (`:com.knuddels.jtokkit.api.Encoding`) object for the given encoding or model type.

Returns `nil` if it does not exist.

```
(do
  (load-module :jtokkit ['jtokkit :as 'jt])
  ;; for a list of encoding types see `(jtokkit/encoding-types)`
  (jt/encoding :CL100K_BASE))

(do
  (load-module :jtokkit ['jtokkit :as 'jt])
  ;; for a list of model types see `(jtokkit/model-types)`
  (jt/encoding :GPT_3_5_TURBO))
```

SEE ALSO

[jtokkit/encode](#)

Encodes the given text into a list of token ids.

[jtokkit/encoding-types](#)

Returns the defined encoding types. Actually from the enum type `:com.knuddels.jtokkit.api.EncodingType`.

[jtokkit/model-types](#)

Returns the defined model types. Actually from the enum type `:com.knuddels.jtokkit.api.ModelType`.

[jtokkit/count-tokens](#)

Encodes the given text into a list of token ids and returns the number of tokens.

jtokit/encoding-types

(encoding-types)

Returns the defined encoding types. Actually from the enum type :com.knuddels.jtokkit.api.EncodingType.

```
(do
  (load-module :jtokit ['jtokit :as 'jt])
  (jt/encoding-types))
```

SEE ALSO

[jtokit/encode](#)

Encodes the given text into a list of token ids.

[jtokit/model-types](#)

Returns the defined model types. Actually from the enum type :com.knuddels.jtokkit.api.ModelType.

jtokit/model-types

(model-types)

Returns the defined model types. Actually from the enum type :com.knuddels.jtokkit.api.ModelType.

```
(do
  (load-module :jtokit ['jtokit :as 'jt])
  (jt/model-types))
```

SEE ALSO

[jtokit/encode](#)

Encodes the given text into a list of token ids.

[jtokit/encoding-types](#)

Returns the defined encoding types. Actually from the enum type :com.knuddels.jtokkit.api.EncodingType.

just

(just x)

Creates a wrapped x, that is dereferenceable

```
(just 10)
=> (just 10)
```

```
(just "10")
=> (just "10")
```

```
(deref (just 10))  
=> 10
```

[section](#)[top](#)

just?

```
(just? x)
```

Returns true if x is of type just

```
(just? (just 1))  
=> true
```

[section](#)[top](#)

juxt

```
(juxt f)  
(juxt f g)  
(juxt f g h)  
(juxt f g h & fs)
```

Takes a set of functions and returns a fn that is the juxtaposition of those fns. The returned fn takes a variable number of args, and returns a vector containing the result of applying each fn to the args (left-to-right).

```
((juxt a b c) x) => [(a x) (b x) (c x)]
```

```
((juxt first last) '(1 2 3 4))  
=> [1 4]
```

```
(do  
  (defn index-by [coll key-fn]  
    (into {} (map (juxt key-fn identity) coll)))  
  
  (index-by [{:id 1 :name "foo"}  
             {:id 2 :name "bar"}  
             {:id 3 :name "baz"}]  
            :id))  
=> {1 {:name "foo" :id 1} 2 {:name "bar" :id 2} 3 {:name "baz" :id 3}}
```

[section](#)[top](#)

keep

```
(keep f coll)
```

Returns a sequence of the non-nil results of `(f item)`. Note, this means false return values will be included. `f` must be free of side-effects. Returns a transducer when no collection is provided.

```
(keep even? (range 1 4))  
=> (false true false)
```



```
(keep (fn [x] (if (odd? x) x)) (range 4))
=> (1 3)

(keep #{3 5 7} '(1 3 5 7 9))
=> (3 5 7)
```

[section](#)[top](#)

key

```
(key e)
```

Returns the key of the map entry.

```
(key (find {:a 1 :b 2} :b))
=> :b

(key (first (entries {:a 1 :b 2 :c 3})))
=> :a
```

SEE ALSO

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[entries](#)

Returns a collection of the map's entries.

[val](#)

Returns the val of the map entry.

[keys](#)

Returns a collection of the map's keys.

[section](#)[top](#)

keys

```
(keys map)
```

Returns a collection of the map's keys.

Please note that the functions 'keys' and 'vals' applied to the same map are not guaranteed not return the keys and vals in the same order!

To achieve this, keys and vals can be calculated based on the map's entry list:

```
(let [e (entries {:a 1 :b 2 :c 3})]
  (println (map key e))
  (println (map val e)))
```

```
(keys {:a 1 :b 2 :c 3})
=> (:a :b :c)
```

SEE ALSO

[vals](#)

Returns a collection of the map's values.

[entries](#)

Returns a collection of the map's entries.

[map](#)

Applies `f` to the set of first items of each coll, followed by applying `f` to the set of second items in each coll, until any one of the ...

[section](#)[top](#)

keystores/aliases

```
(aliases keystore)
```

Returns the list of aliases defined for the keystore.

```
(do
  (load-module :keystores)
  (let [ks (keystores/load (io/file "cert.p12") "12345")]
    (keystores/aliases ks)))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java `:java.security.KeyStore`.

[keystores/certificate](#)

Returns the certificate (of type `X509Certificate`) with the given alias name from the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/expiry-date](#)

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

[keystores/expired?](#)

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)[top](#)

keystores/certificate

```
(certificate keystore alias)
```

Returns the certificate (of type `X509Certificate`) with the given alias name from the keystore.

```
(do
  (load-module :keystores)
  (let [ks (keystores/load (io/file "cert.p12") "12345")
        alias (first (keystores/aliases ks))]
    (keystores/certificate ks alias)))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java `:java.security.KeyStore`.

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/expiry-date](#)

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

[keystores/expired?](#)

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)

[top](#)

keystores/expired?

```
(expired? keystore alias)
```

Returns true if the certificate with the given alias name in the keystore has expired else false.

```
(do
  (load-module :keystores)
  (let [ks      (keystores/load (io/file "cert.p12") "12345")
        alias (first (keystores/aliases ks))]
    (keystores/expired? ks alias)))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java `:java.security.KeyStore`.

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/certificate](#)

Returns the certificate (of type `X509Certificate`) with the given alias name from the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/expiry-date](#)

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

[section](#)

[top](#)

keystores/expiry-date

```
(expiry-date keystore alias)
```

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

```
(do
  (load-module :keystores)
```

```
(let [ks      (keystores/load (io/file "cert.p12") "12345")
      alias (first (keystores/aliases ks))]
      (keystores/expiry-date ks alias)))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java :java.security.KeyStore.

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/certificate](#)

Returns the certificate (of type X509Certificate) with the given alias name from the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/expired?](#)

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)

[top](#)

keystores/issuer-dn

```
(issuer-dn keystore alias)
```

Returns the issuer DN for the certificate with the given alias name in the keystore.

```
(do
  (load-module :keystores)
  (let [ks      (keystores/load (io/file "cert.p12") "12345")
        alias (first (keystores/aliases ks))]
        (keystores/issuer-dn ks alias)))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java :java.security.KeyStore.

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/certificate](#)

Returns the certificate (of type X509Certificate) with the given alias name from the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/parse-dn](#)

Parses a DN and returns a map with the DN's elements.

[keystores/expiry-date](#)

Returns the expiry date as a :java.time.LocalDateTime for the certificate with the given alias name in the keystore.

[keystores/expired?](#)

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)

[top](#)

keystores/load

(load in password)

Loads a certificate into a Java `KeyStore`. Reads it from the input 'in' and returns it as a Java `:java.security.KeyStore`.

'in' may be a:

- `bytebuf`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.nio.file.Path`
- `java.io.InputStream`

Note: The caller is responsible for closing the in stream!

```
(do
  (load-module :keystores)
  (keystores/load (io/file "cert.p12") "12345"))
```

SEE ALSO

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/certificate](#)

Returns the certificate (of type `X509Certificate`) with the given alias name from the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/expiry-date](#)

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

[keystores/expired?](#)

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)

[top](#)

keystores/parse-dn

(parse-dn dn)

Parses a DN and returns a map with the DN's elements.

Typical elements of an LDAP distinguished name are:

CN	Common name
O	Organisation
OU	Organisational unit
ST	State or province
OID.2.5.4.17	Zip code
L	Locality name (city)
C	Country

```
(do
  (load-module :keystores)
  (let [ks      (keystores/load (io/file "cert.p12") "12345")
        alias (first (keystores/aliases ks))]
    (-> (keystores/subject-dn ks alias)
        (keystores/parse-dn))))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java `:java.security.KeyStore`.

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/certificate](#)

Returns the certificate (of type `X509Certificate`) with the given alias name from the keystore.

[keystores/subject-dn](#)

Returns the subject DN for the certificate with the given alias name in the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/expiry-date](#)

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

[keystores/expired?](#)

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)

[top](#)

keystores/subject-dn

```
(subject-dn keystore alias)
```

Returns the subject DN for the certificate with the given alias name in the keystore.

```
(do
  (load-module :keystores)
  (let [ks      (keystores/load (io/file "cert.p12") "12345")
        alias (first (keystores/aliases ks))]
    (keystores/subject-dn ks alias)))
```

SEE ALSO

[keystores/load](#)

Loads a certificate into a Java KeyStore. Reads it from the input 'in' and returns it as a Java `:java.security.KeyStore`.

[keystores/aliases](#)

Returns the list of aliases defined for the keystore.

[keystores/certificate](#)

Returns the certificate (of type `X509Certificate`) with the given alias name from the keystore.

[keystores/issuer-dn](#)

Returns the issuer DN for the certificate with the given alias name in the keystore.

[keystores/parse-dn](#)

Parses a DN and returns a map with the DN's elements.

[keystores/expiry-date](#)

Returns the expiry date as a `:java.time.LocalDateTime` for the certificate with the given alias name in the keystore.

keystores/expired?

Returns true if the certificate with the given alias name in the keystore has expired else false.

[section](#)[top](#)

keyword

```
(keyword name)
(keyword ns name)
```

Returns a keyword from the given name

```
(keyword "a")
=> :a

(keyword :a)
=> :a

(keyword :foo/a)
=> :foo/a

(keyword "foo" "a")
=> :foo/a

(keyword (. :java.time.Month :JANUARY)) ;; java enum to keyword
=> :JANUARY

(name :foo/a)
=> "a"

(namespace :foo/a)
=> "foo"
```

SEE ALSO

[name](#)

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[section](#)[top](#)

keyword?

```
(keyword? x)
```

Returns true if x is a keyword

```
(keyword? (keyword "a"))
=> true

(keyword? :a)
=> true
```

```
(keyword? nil)
=> false
```

```
(keyword? 'a)
=> false
```

[section](#)[top](#)

kira/escape-html

```
(kira/escape-html val)
(kira/escape-html val f)
```

Returns a HTML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

An optional function f transforms the value before being converted to a string and HTML escaped.

```
(do
  (ns test)
  (load-module :kira)

  (println (kira/eval "<div><%= (kira/escape-html formula) %></div>"
    { :formula "x > 100" })))

  (defn format [t] (time/format t "yyyy-MM-dd"))
  (println (kira/eval "<div><%= (kira/escape-html date test/format) %></div>"
    { :date (time/local-date 2000 8 1) })))

<div>x &gt; 100</div>
<div>2000-08-01</div>
=> nil
```

SEE ALSO

[kira/escape-xml](#)

Returns an XML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

[section](#)[top](#)

kira/escape-xml

```
(kira/escape-xml val)
(kira/escape-xml val f)
```

Returns an XML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

An optional function f transforms the value before being converted to a string and XML escaped.

```
(do
  (ns test)
  (load-module :kira)

  (println (kira/eval "<formula><%= (kira/escape-xml formula) %></formula>"
    { :formula "x > 100" })))

  (defn format [t] (time/format t "yyyy-MM-dd"))
  (println (kira/eval "<date><%= (kira/escape-xml date test/format) %></date>"
    { :date (time/local-date 2000 8 1) })))
```



```
<formula>x &gt; 100</formula>
<date>2000-08-01</date>
=> nil
```

SEE ALSO

[kira/escape-html](#)

Returns a HTML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

[section](#)

[top](#)

kira/eval

```
(kira/eval source)
(kira/eval source bindings)
(kira/eval source delimiters bindings)
```

Evaluate a template using the supplied bindings. The template source may be a string, or an I/O source such as a File, Reader or InputStream.

```
(do
  (ns test)
  (load-module :kira)

  (println (kira/eval "Hello <%= name %>" { :name "Alice" })))
  (println (kira/eval "1 + 2 = <%= (+ 1 2) %>"))
  (println (kira/eval "2 + 3 = <% (print (+ 2 3)) %>"))
  (println (kira/eval "$[=x]$ + $[=y]$ = $[= (+ x y) ]$"
    ["$[" "]"$"]"
    {:x 4 :y 5})))

  (println (kira/eval "margin: <%= (if large 100 10) %>"
    { :large false })))
  (println (kira/eval "fruits: <% (doseq [f fruits] %><%= f %> <% ) %>"
    { :fruits '("apple" "peach") })))
  (println (kira/eval "fruits: <% (doseq [f fruits] %><%= f %> <% ) %>"
    { :fruits '("apple" "peach") })))

  (println (kira/eval "when: <% (when large %>is large<% ) %>"
    { :large true })))
  (println (kira/eval "if: <% (if large (do %>100<% ) (do %>1<% )) %>"
    { :large true })))

  (println (kira/eval "<div><%= (kira/escape-html formula) %></div>"
    { :formula "12 < 15" })))

Hello Alice
1 + 2 = 3
2 + 3 = 5
4 + 5 = 9
margin: 10
fruits: apple peach
fruits: apple peach
when: is large
if: 100
<div>12 &lt; 15</div>
=> nil
```

SEE ALSO

[kira/fn](#)

Compile a template into a function that takes the supplied arguments. The template source may be a string, or an I/O source such as ...

[kira/escape-xml](#)

Returns an XML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

[kira/escape-html](#)

Returns a HTML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

[section](#)[top](#)

kira/fn

```
(kira/fn args source)
(kira/fn args source delimiters)
```

Compile a template into a function that takes the supplied arguments. The template source may be a string, or an I/O source such as a File, Reader or InputStream.

```
(do
  (load-module :kira)

  (def hello (kira/fn [name] "Hello <%= name %>"))
  (println (hello "Alice"))
  (println (hello "Bob")))
Hello Alice
Hello Bob
=> nil
```

SEE ALSO

[kira/eval](#)

Evaluate a template using the supplied bindings. The template source may be a string, or an I/O source such as a File, Reader or InputStream.

[kira/escape-xml](#)

Returns an XML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

[kira/escape-html](#)

Returns a HTML escaped string. If the passed data is not of type string it will be converted first to a string using the 'str' function.

[section](#)[top](#)

last

```
(last coll)
```

Returns the last element of coll.

```
(last nil)
=> nil

(last [])
=> nil

(last [1 2 3])
=> 3

(last '())
=> nil
```

```
(last '(1 2 3))  
=> 3
```

```
(last "abc")  
=> #\c
```

[section](#)[top](#)

last-index-of

```
(last-index-of sequence val)  
(last-index-of comparefn sequence val)
```

Returns the last index of the sequence value that is equal to val or -1 if not found.

If no compare function is supplied, uses the natural compare to compare the sequence values to the supplied value. The compare function takes two arguments and returns -1, 0, or 1.

```
(last-index-of [1 2 2 3] 2)  
=> 2
```

```
(last-index-of [1 2 3] 6)  
=> -1
```

```
(last-index-of [1 2 3] nil)  
=> -1
```

```
(last-index-of [1 nil 3] nil)  
=> 1
```

```
(last-index-of nil 7)  
=> -1
```

```
(last-index-of compare [1 2 2 3] 2)  
=> 2
```

```
(last-index-of #(if (== %1 %2) 0 1) [1 2 2 3] 2)  
=> 2
```

SEE ALSO

[index-of](#)

Returns the first index of the sequence value that is equal to val or -1 if not found.

[section](#)[top](#)

latest

```
(latest)
```

Returns the latest available Venice version or `nil` if the internet is not available.

```
(latest)
```

SEE ALSO

[version](#)

Returns the Venice version.

[parse-version](#)

Parses a version to its elements: major, minor, patch and suffix.

[newer-version?](#)

Returns true if v1 is newer than v2 else false.

[section](#)[top](#)

lazy-seq

```
(lazy-seq)
(lazy-seq f)
(lazy-seq seed f)
(lazy-seq head tail-lazy-seq)
```

Creates a new lazy sequence.

```
(lazy-seq)
empty lazy sequence
```

```
(lazy-seq f)
```

(theoretically) infinitely lazy sequence using a repeatedly invoked supplier function for each next value. The supplier function `f` is a no arg function. The sequence ends if the supplier function returns `nil`.

```
(lazy-seq seed f)
```

(theoretically) infinitely lazy sequence with a seed value and a supplier function to calculate the next value based on the previous. `f` is a single arg function. The sequence ends if the supplier function returns `nil`.

```
(lazy-seq head tail-lazy-seq)
```

Constructs a lazy sequence of a head element and a lazy sequence tail supplier.

```
; empty lazy sequence
(->> (lazy-seq)
      (doall))
=> ()

; lazy sequence with a supplier function producing random longs
(->> (lazy-seq rand-long)
      (take 4)
      (doall))
=> (6807991165495902999 4602271635936598917 6593258571566415489 1012111775328977098)

; lazy sequence with a constant value
(->> (lazy-seq (constantly 5))
      (take 4)
      (doall))
=> (5 5 5 5)

; lazy sequence with a seed value and a supplier function
; producing of all positive numbers (1, 2, 3, 4, ...)
(->> (lazy-seq 1 inc)
      (take 10)
      (doall))
=> (1 2 3 4 5 6 7 8 9 10)
```

```

; producing of all positive even numbers (2, 4, 6, ...)
(->> (lazy-seq 2 #(+ % 2))
      (take 10)
      (doall))
=> (2 4 6 8 10 12 14 16 18 20)

; lazy sequence as value producing function
(interleave [:a :b :c] (lazy-seq 1 inc))
=> (:a 1 :b 2 :c 3)

; lazy sequence with a mapping
(->> (lazy-seq 1 (fn [x] (do (println "realized" x)
                             (inc x))))
      (take 10)
      (map #(* 10 %))
      (take 2)
      (doall))
realized 1
=> (10 20)

; finite lazy sequence from a vector
(->> (lazy-seq [1 2 3 4])
      (doall))
=> (1 2 3 4)

; finite lazy sequence with a supplier function that
; returns nil to terminate the sequence
(do
  (def counter (atom 5))
  (defn generate []
    (swap! counter dec)
    (if (pos? @counter) @counter nil))
  (doall (lazy-seq generate)))
=> (4 3 2 1)

; lazy sequence from a head element and a tail lazy
; sequence
(->> (cons -1 (lazy-seq 0 #(+ % 1)))
      (take 5)
      (doall))
=> (-1 0 1 2 3)

; lazy sequence from a head element and a tail lazy
; sequence
(->> (lazy-seq -1 (lazy-seq 0 #(+ % 1)))
      (take 5)
      (doall))
=> (-1 0 1 2 3)

; lazy sequence show its power to generate the Fibonacci sequence
(do
  (def fib (map first (lazy-seq [0N 1N] (fn [[a b]] [b (+ a b)]))))
  (doall (take 10 fib)))
=> (0N 1N 1N 2N 3N 5N 8N 13N 21N 34N)

```

SEE ALSO

[doall](#)

When lazy sequences are produced doall can be used to force any effects and realize the lazy sequence. Returns the realized items in a list!

[lazy-seq?](#)

Returns true if obj is a lazyseq

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[cycle](#)

Returns a lazy (infinite!) sequence of repetitions of the items in coll.

[repeat](#)

Returns a lazy sequence of x values or a collection with the value x repeated n times.

[section](#)

[top](#)

lazy-seq?

```
(lazy-seq? obj)
```

Returns true if obj is a lazyseq

```
(lazy-seq? (lazy-seq rand-long))  
=> true
```

SEE ALSO

[lazy-seq](#)

Creates a new lazy sequence.

[section](#)

[top](#)

let

```
(let [bindings*] exprs*)
```

Evaluates the expressions and binds the values to symbols in the new local context.

```
(let [x 1] x)  
=> 1
```

```
(let [x 1  
      y 2]  
  (+ x y))  
=> 3
```

```
;; Destructured list  
(let [[x y] '(1 2)]  
  (printf "x: %d, y: %d\n" x y))  
x: 1, y: 2  
=> nil
```

```
;; Destructured map  
(let [{:keys [width height title]  
       :or {width 640 height 500}  
       :as styles}  
      {:width 1000 :title "Title"}]  
  (println "width: " width)  
  (println "height: " height)  
  (println "title: " title))
```

```
(println "styles: " styles))
width: 1000
height: 500
title: Title
styles: {:width 1000 :title Title}
=> nil
```

SEE ALSO

[letfn](#)

Takes a vector of function specs and a body, and generates a set of bindings of functions to their names. All of the names are available ...

[if-let](#)

bindings is a vector with 2 elements: binding-form test.

[when-let](#)

bindings is a vector with 2 elements: binding-form test.

[binding](#)

Evaluates the expressions and binds the values to dynamic (thread-local) symbols

[section](#)[top](#)

letfn

```
(letfn [fnspec*] exprs*)
```

Takes a vector of function specs and a body, and generates a set of bindings of functions to their names. All of the names are available in all of the definitions of the functions, as well as the body.

fnspec ==> (fname [params*] exprs) or (fname ([params*] exprs)+)

```
(letfn [(foo [] "abc")] (foo))
```

is equivalent to

```
(let [foo (fn [] "abc")] (foo))
```

```
(letfn [(foo [] "abc")
        (bar [] (str (foo) "def"))]
  (bar))
=> "abcdef"
```

SEE ALSO

[let](#)

Evaluates the expressions and binds the values to symbols in the new local context.

[section](#)[top](#)

license

```
(license)
```

Returns the Venice license text.

```
(println (license))
```

[section](#)[top](#)

license-all

```
(license-all)
```

Returns the Venice license text with all 3rd party licenses.

```
(println (license-all))
```

[section](#)[top](#)

list

```
(list & items)
```

Creates a new list containing the items.

```
(list)
```

```
=> ()
```

```
(list 1 2 3)
```

```
=> (1 2 3)
```

```
(list 1 2 3 [:a :b])
```

```
=> (1 2 3 [:a :b])
```

[section](#)[top](#)

list*

```
(list* args)
```

```
(list* a args)
```

```
(list* a b args)
```

```
(list* a b c args)
```

```
(list* a b c d & more)
```

Creates a new list containing the items prepended to the rest, the last of which will be treated as a collection.

```
(list* 1 '(2 3))
```

```
=> (1 2 3)
```

```
(list* 1 2 3 [4])
```

```
=> (1 2 3 4)
```

```
(list* 1 2 3 '(4 5))
```

```
=> (1 2 3 4 5)
```

```
(list* '(1 2) 3 [4])
```

```
=> ((1 2) 3 4)
```

```
(list* nil)
```

```
=> nil
```



```
(list* nil [2 3])
=> (nil 2 3)
```

```
(list* 1 2 nil)
=> (1 2)
```

SEE ALSO

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[conj](#)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item) and (conj item) returns item.

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[vector*](#)

Creates a new vector containing the items prepended to the rest, the last of which will be treated as a collection.

[section](#)[top](#)

list-comp

```
(list-comp seq-exprs body-expr)
```

List comprehension. Takes a vector of one or more binding-form or collection-expr pairs, each followed by zero or more modifiers, and yields a collection of evaluations of expr.

Supported modifiers are: `:when` predicate

List comprehensions are used when multiple lists have to be processed down to a single list and some filtering has to be applied.

Please use the ``for`` list comprehension instead.

```
(list-comp [x (range 10)] x)
=> (0 1 2 3 4 5 6 7 8 9)
```

```
(list-comp [x (range 5)] (* x 2))
=> (0 2 4 6 8)
```

```
(list-comp [x (range 10) :when (odd? x)] x)
=> (1 3 5 7 9)
```

```
(list-comp [x (range 10) :when (odd? x)] (* x 2))
=> (2 6 10 14 18)
```

```
(list-comp [x [1 2 3 4] :when (odd? x)
           y [1 2 3 4] :when (even? y)]
           [x y])
=> ([1 2] [1 4] [3 2] [3 4])
```

SEE ALSO

[for](#)

List comprehension. Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and ...

[mapcat](#)

Returns the result of applying concat to the result of applying map to fn and colls. Thus function fn should return a collection.

[doseq](#)

Repeatedly executes body (presumably for side-effects) with bindings and filtering as provided by list-comp. Does not retain the head ...

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[section](#)[top](#)

list?

```
(list? obj)
```

Returns true if obj is a list

```
(list? (list 1 2))  
=> true
```

```
(list? '(1 2))  
=> true
```

[section](#)[top](#)

load-classpath-file

```
(load-classpath-file f)  
(load-classpath-file f force)  
(load-classpath-file f nsalias)  
(load-classpath-file f force nsalias)
```

Sequentially read and evaluate the set of forms contained in the classpath file. The function is restricted to classpath files with the extension '.venice'.

Returns a tuple with the file's name and the keyword `:loaded` if the file has been successfully loaded or `:already-loaded` if the file has been already loaded. Throws an exception on any loading error.

With 'force' set to `false` (the default) the file is only loaded once and interpreted once. Subsequent load attempts will be skipped. With 'force' set to `true` it is always loaded and interpreted.

Loaded files are cached by Venice and subsequent loads are just skipped. To enforce a reload call the file load with the force flag set to true:

```
(load-classpath-file "com/github/jlangch/venice/test.venice" true)
```

An optional namespace alias can be passed: `(load-classpath-file "com/github/jlangch/venice/test.venice" ['test :as 't])`

`load-classpath-file` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
(do  
  (load-classpath-file "com/github/jlangch/venice/test-support.venice")  
  (test-support/test-fn "hello"))  
=> "test: hello"
```

```
(do  
  (load-classpath-file "com/github/jlangch/venice/test-support.venice")  
  (test-support/test-fn "hello")  
  ; reload the classpath file  
  (ns-remove 'test-support)  
  (load-classpath-file "com/github/jlangch/venice/test-support.venice" true)  
  (test-support/test-fn "hello"))  
=> "test: hello"
```

```
;; namespace aliases
(do
  (load-classpath-file "com/github/jlangch/venice/test-support.venice" ['test-support :as 't])
  (t/test-fn "hello"))
=> "test: hello"
```

SEE ALSO

[load-file](#)

Sequentially read and evaluate the set of forms contained in the file.

[load-string](#)

Sequentially read and evaluate the set of forms contained in the string.

[load-module](#)

Loads a Venice predefined extension module.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)

[top](#)

load-file

```
(load-file f)
(load-file f force)
(load-file f nsalias)
(load-file f force nsalias)
```

Sequentially read and evaluate the set of forms contained in the file.

If the file is found on one of the defined load paths it is read and the forms it contains are evaluated. If the file is not found an exception is raised.

Returns a tuple with the file's name and the keyword `:loaded` if the file has been successfully loaded or `:already-loaded` if the file has been already loaded. Throws an exception on any loading error.

With 'force' set to `false` (the default) the file is only loaded once and interpreted once. Subsequent load attempts will be skipped. With 'force' set to `true` it is always loaded and interpreted.

The function is restricted to load files with the extension '.venice'. If the file extension is missing '.venice' will be implicitly added.

An optional namespace alias can be passed: `(load-file "coffee.venice" ['coffee :as 'c])`

`load-file` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
;; With load-paths: [/users/foo/scripts]
;;      -> loads: /users/foo/scripts/coffee.venice
(load-file "coffee")

;; With load-paths: [/users/foo/scripts]
;;      -> loads: /users/foo/scripts/coffee.venice
(load-file "coffee.venice")

;; With load-paths: [/users/foo/scripts]
;;      -> loads: /users/foo/scripts/beverages/coffee.venice
(load-file "beverages/coffee")

;; With load-paths: [/users/foo/resources.zip]
;;      -> loads: /users/foo/resources.zip!beverages/coffee.venice
(load-file "beverages/coffee")
```

SEE ALSO

[load-classpath-file](#)

Sequentially read and evaluate the set of forms contained in the classpath file. The function is restricted to classpath files with ...

[load-string](#)

Sequentially read and evaluate the set of forms contained in the string.

[load-module](#)

Loads a Venice predefined extension module.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[section](#)[top](#)

load-module

```
(load-module m)
(load-module m force)
(load-module m nsalias)
(load-module m force nsalias)
```

Loads a Venice predefined extension module.

Returns a tuple with the module's name and the keyword `:loaded` if the module has been successfully loaded or `:already-loaded` if the module has been already loaded. Throws an exception on any loading error.

With 'force' set to `false` (the default) the module is only loaded once and interpreted once. Subsequent load attempts will be skipped. With 'force' set to `true` it is always loaded and interpreted.

Loaded modules are cached by Venice and subsequent loads are just skipped. To enforce a reload call the module load with the force flag set to true: `(load-module :hexdump true)`

An optional namespace alias can be passed: `(load-module :hexdump ['hexdump :as 'h])`

`load-module` supports load paths. See the `loadpath/paths` doc for a description of the *load path* feature.

```
(load-module :trace)

;; loading the :trace modul and define a ns alias 't for namespace
;; 'trace used in the module
(load-module :trace ['trace :as 't])

;; reloading a module
(do
  (load-module :trace)
  ; reload the module
  (ns-remove 'trace)
  (load-module :trace true))

;; namespace aliases
(do
  (load-module :hexdump ['hexdump :as 'h])
  (h/dump (range 32 64)))

;; dynamically load a module
(let [mname (keyword "hexdump")]
  (load-module mname))
```

SEE ALSO

[load-file](#)

Sequentially read and evaluate the set of forms contained in the file.

[load-classpath-file](#)

Sequentially read and evaluate the set of forms contained in the classpath file. The function is restricted to classpath files with ...

[load-string](#)

Sequentially read and evaluate the set of forms contained in the string.

[loaded-modules](#)

Returns the names of the loaded modules.

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[doc](#)

Prints documentation for a var or special form given x as its name. Prints the definition of custom types.

[section](#)

[top](#)

load-string

```
(load-string s)
```

Sequentially read and evaluate the set of forms contained in the string.

```
(do
  (load-string "(def x 1)")
  (+ x 2))
=> 3
```

SEE ALSO

[load-file](#)

Sequentially read and evaluate the set of forms contained in the file.

[load-classpath-file](#)

Sequentially read and evaluate the set of forms contained in the classpath file. The function is restricted to classpath files with ...

[loaded-modules](#)

Returns the names of the loaded modules.

[section](#)

[top](#)

loaded-modules

```
(loaded-modules)
```

Returns the names of the loaded modules.

SEE ALSO

[load-module](#)

Loads a Venice predefined extension module.

[section](#)

[top](#)

loadpath/normalize

(loadpath/normalize f)

Normalize a relative file regarding the load paths.

With the load paths: `["/Users/foo/img.png", "/Users/foo/resources"]`

- `(loadpath/normalize "img.png") -> "/Users/foo/img.png"`
- `(loadpath/normalize "test.json") -> "/Users/foo/resources/test.json"`
- `(loadpath/normalize "/tmp/data.json") -> "/tmp/data.json"`

SEE ALSO

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[loadpath/unrestricted?](#)

Returns true if the load paths are unrestricted.

[section](#)

[top](#)

loadpath/paths

(loadpath/paths)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the application level. They are passed as part of the sandbox to the Venice evaluator.

The functions that support load paths try sequentially every load path to access files. If a load path is a ZIP file, files can be read from within that ZIP file.

Example:

```
/Users/foo/demo
|
+--- resources.zip
|
+--- /data
    |
    +--- config.json
    |
    +--- /scripts
        |
        +--- script1.venice
```

With a load path configuration of `["/Users/foo/demo/resources.zip", "/Users/foo/demo/data"]`

- `(io/slurp "config.json") -> slurps /Users/foo/demo/data/config.json`
- `(io/slurp "scripts/script1.venice") -> slurps /Users/foo/demo/data/scripts/script1.venice`
- `(io/slurp "img1.png") -> slurps /Users/foo/demo/resources.zip!img1.png`

I/O functions with support for load paths:

- `load-file`
- `io/slurp`
- `io/slurp-lines`
- `io/spit`

- `io/file-in-stream`
- `io/file-out-stream`
- `io/delete-file`

To enforce a Venice script to read/write files on the load paths only:

- Define a custom sandbox
- Disable all I/O functions
- Enable the I/O functions that support load paths

SEE ALSO

[loadpath/unrestricted?](#)

Returns true if the load paths are unrestricted.

[loadpath/normalize](#)

Normalize a relative file regarding the load paths.

[load-file](#)

Sequentially read and evaluate the set of forms contained in the file.

[section](#)

[top](#)

loadpath/unrestricted?

`(loadpath/unrestricted?)`

Returns true if the load paths are unrestricted.

SEE ALSO

[loadpath/paths](#)

Returns the list of the defined load paths. A load path is either a file, a ZIP file, or a directory. Load paths are defined at the ...

[loadpath/normalize](#)

Normalize a relative file regarding the load paths.

[section](#)

[top](#)

lock

`(lock)`

Creates a new lock object.

The lock object implements the Java `AutoClosable` interface thus it can be used with try-with-resources.

```
(let [l (lock)]
  (acquire l)
  ;; do something
  (release l))
=> nil
```

```
(let [l (lock)]
  (try-with [l (acquire l)]
    ;; do something
```

```
    ))  
=> nil
```

SEE ALSO

[acquire](#)

Acquires a lock, blocking until the lock is available.

[try-acquire](#)

Acquires a lock within the given timeout time. Without a timeout returns immediately if the lock is not available.

[release](#)

Releases a lock.

[locked?](#)

Returns true if the lock is in use else false.

[lock?](#)

Returns true if o is a lock object else false.

[section](#)[top](#)

lock?

```
(lock? o)
```

Returns `true` if o is a lock object else `false`.

```
(let [l (lock)]  
  (lock? l))  
=> true
```

SEE ALSO

[acquire](#)

Acquires a lock, blocking until the lock is available.

[try-acquire](#)

Acquires a lock within the given timeout time. Without a timeout returns immediately if the lock is not available.

[release](#)

Releases a lock.

[locked?](#)

Returns true if the lock is in use else false.

[section](#)[top](#)

locked?

```
(locked? lock)
```

Returns `true` if the lock is in use else `false`.

```
(let [l (lock)]  
  (acquire l)  
  (locked? l))  
=> true
```


SEE ALSO

[lock](#)

Creates a new lock object.

[acquire](#)

Acquires a lock, blocking until the lock is available.

[try-acquire](#)

Acquires a lock within the given timeout time. Without a timeout returns immediately if the lock is not available.

[release](#)

Releases a lock.

[section](#)[top](#)

locking

(locking x & exprs)

Executes 'exprs' in an implicit do, while holding the monitor of 'x'. Will release the monitor of 'x' in all circumstances. Locking operates like the synchronized keyword in Java.

```
(do
  (def x 1)
  (locking x
    (println 100)
    (println 200)))
100
200
=> nil

;; Locks are reentrant
(do
  (def x 1)
  (locking x
    (locking x
      (println "in"))
    (println "out")))
in
out
=> nil

(defn log [msg] (locking log (println msg)))
(log "message")
message
=> nil
```

[section](#)[top](#)

log

(log x)

Returns the natural logarithm (base e) of a value

```
(log 10)
=> 2.302585092994046
```

```
(log 10.23)
=> 2.325324579963535
```

```
(log 10.23M)
=> 2.325324579963535
```

SEE ALSO

[log10](#)

Returns the base 10 logarithm of a value

[log2](#)

Returns the base 2 logarithm of a value

[section](#)

[top](#)

log10

```
(log10 x)
```

Returns the base 10 logarithm of a value

```
(log10 10)
=> 1.0
```

```
(log10 10.23)
=> 1.0098756337121602
```

```
(log10 10.23M)
=> 1.0098756337121602
```

```
;; the number of digits
(long (+ (floor (log10 235)) 1))
=> 3
```

SEE ALSO

[log](#)

Returns the natural logarithm (base e) of a value

[log2](#)

Returns the base 2 logarithm of a value

[section](#)

[top](#)

log2

```
(log2 x)
```

Returns the base 2 logarithm of a value

```
(log2 8)
=> 3.0
```

```
(log2 10.23)
=> 3.354734239970604
```

```
(log2 10.23M)
=> 3.354734239970604
```

SEE ALSO

[log](#)

Returns the natural logarithm (base e) of a value

[log10](#)

Returns the base 10 logarithm of a value

[section](#)[top](#)

logger/console-logger

```
(console-logger)
(console-logger formatter)
```

Create a console logger with either the default log message formatter or a custom formatter.

The console logger has the fix id `:console`.

```
(do
  (load-module :logger)

  ;; create the console logger
  (logger/console-logger)

  ;; get the :console logger
  (def log(logger/logger :console))

  ;; log some messages
  (log :info "test message")
  (log :info "kermit" "test message")
  (log :info "kermit" "test message" (ex :VncException "test exception")))

(do
  (load-module :logger)

  ;; custom formatter
  (defn custom-formatter [level principal msg exc]
    (let [ex-msg (logger/format-ex exc)]
      (str/format "%s [%s] [%s] [%s] %s%n"
        (time/format (time/local-date-time) "yyyy-MM-dd HH:mm:ss.SSS")
        (logger/format-level level)
        (thread-name)
        (if (keyword? principal) (name principal) (str principal))
        (if (nil? ex-msg) msg (str msg logger/linefeed ex-msg)))))

  ;; create the console logger with a custom formatter
  (logger/console-logger custom-formatter)

  ;; get the :console logger
  (def log (logger/logger :console))

  ;; log some messages
```

```
(log :info "test message")
(log :info "kermit" "test message")
(log :info "kermit" "test message" (ex :VncException "test exception")))
```

SEE ALSO

[logger/logger](#)

Returns the logger with the specified ID.

[logger/file-logger](#)

Create a file logger. The logger is identified by it's ID.

[logger/format-level](#)

Formats a log level. The 'level' must be a keyword!

[logger/format-ex](#)

Formats an exception. The exception may be a Venice exception or a Java exception.

[section](#)

[top](#)

logger/enable-auto-start-rotation-scheduler

```
(enable-auto-start-rotation-scheduler activate)
```

Enable or disable the auto starting of the rotation scheduler. Default is deactivated.

If enabled, the scheduler starts when the first rotating file logger is created. As long as no rotating file logger exists, the scheduler remains inactive to conserve resources.

```
(do
  (load-module :logger)

  ;; disable auto start (auto start is enabled by default)
  (logger/enable-auto-start-rotation-scheduler false)

  (logger/file-logger :venice "/logs/venice.log"
    :rotate-mode :daily
    :rotate-dir "/logs/archive")

  (def log (logger/logger :venice))

  (log :info "test message 1")
  (log :info "test message 2"))
```

SEE ALSO

[logger/rotation-scheduler-running?](#)

Returns true if a rotation scheduler is running else false.

[logger/start-rotation-scheduler](#)

Manually start the rotation scheduler.

[logger/rotate](#)

Manually rotates the log file of the logger with the ID 'id' if the logger has file rotation enabled.

[logger/rotate-all](#)

Manually rotates the log files of all loggers that have file rotation enabled.

[logger/requires-rotation?](#)

Returns true if there is at least on logger that has log file rotation enabled.

[section](#)

[top](#)

logger/file-logger

(file-logger id file & options)

Create a file logger. The logger is identified by it's ID.

Arguments:

id The mandatory logger ID. ID's are of type keyword like `:venice`.

file The mandatory file to log to. The parent dir must exist!

Options:

`:max-size n` The max size of the file in bytes. If the size is reached, the file will be truncated at a line boundary to the given max size. Defaults to `-1` meaning there is no limit in size.
The max size can be specified as a number like `20000` or a number with a unit like `:20KB`, `:20MB`, or `:2GB`

`:formatter f` A custom formatter. Defaults to `nil` to activate the default formatter

`:rotate-mode k` A file rotation mode { `:none`, `:daily`, `:monthly` }. Defaults to `:none`
Name convention for rotated files:
daily: `venice.log` `venice-{yyyy.mm.dd}.log`
monthly: `venice.log` `venice-{yyyy.mm}.log`

`:rotate-dir f` The file rotation directory. The directory must be specified and must exist if the rotate mode is `:daily` or `:monthly`. `nil` is passed for mode `:none`

```
(do
  (load-module :logger)

  ;; create the file loggers
  (logger/file-logger :venice "/data/logs/venice.log")
  (logger/file-logger :database "/data/logs/database.log" :max-size :20MB)

  ;; get the loggers
  (def log-venice (logger/logger :venice))
  (def log-database (logger/logger :database))

  ;; log some messages
  (log-venice :info "test message")
  (log-venice :info "kermit" "test message")
  (log-venice :info "kermit" "test message" (ex :VncException "test exception"))
  (log-database :info "test message"))

(do
  (load-module :logger)

  ;; create the rotating file logger
  (logger/file-logger :venice
    "/data/logs/venice.log"
    :max-size :20MB
    :rotate-mode :daily
    :rotate-dir "/data/logs/archive")

  ;; get the logger
  (def log (logger/logger :venice))

  ;; log some messages
  (log :info "test message")
  (log :info "kermit" "test message")
  (log :info "kermit" "test message" (ex :VncException "test exception")))
```

```

(do
  (load-module :logger)

  ;; custom formatter
  (defn custom-formatter [level principal msg exc]
    (let [ex-msg (logger/format-ex exc)]
      (str/format "%s [%s] [%s] [%s] %s%n"
        (time/format (time/local-date-time) "yyyy-MM-dd HH:mm:ss.SSS")
        (logger/format-level level)
        (thread-name)
        (if (keyword? principal) (name principal) (str principal))
        (if (nil? ex-msg) msg (str msg logger/linefeed ex-msg)))))

  ;; create the file logger with a custom formatter
  (logger/file-logger :venice "/data/logs/venice.log"
    :formatter custom-formatter)

  ;; get the :venice logger
  (def log (logger/logger :venice))

  ;; log some messages
  (log :info "test message")
  (log :info "kermit" "test message")
  (log :info "kermit" "test message" (ex :VncException "test exception")))

```

SEE ALSO

[logger/logger](#)

Returns the logger with the specified ID.

[logger/console-logger](#)

Create a console logger with either the default log message formatter or a custom formatter.

[logger/format-level](#)

Formats a log level. The 'level' must be a keyword!

[logger/format-ex](#)

Formats an exception. The exception may be a Venice exception or a Java exception.

[section](#)

[top](#)

logger/format-level

```
(format-level level)
```

Formats a log level. The 'level' must be a keyword!

```

(do
  (load-module :logger)

  (logger/format-level :info)
  (logger/format-level :warn))

```

[section](#)

[top](#)

logger/level

```
(level id)
(level id new-level)
```

Returns or sets the log level of logger with the specified ID.

Level is one of `:debug`, `:info`, `:warn`, `:error`, or `:fatal`

```
(do
  (load-module :logger)

  (println (logger/level :console)) ;; get log level

  (logger/level :console :warn) ;; set log level

  (println (logger/level :console))) ;; get log level
```

SEE ALSO

[logger/level-all](#)

Sets the log level of all registered loggers.

[section](#)

[top](#)

logger/level-all

```
(level-all new-level)
```

Sets the log level of all registered loggers.

Level is one of `:debug`, `:info`, `:warn`, `:error`, or `:fatal`

```
(do
  (load-module :logger)

  (println (logger/level :console)) ;; get log level

  (logger/level-all :warn) ;; set log level

  (println (logger/level :console))) ;; get log level
```

SEE ALSO

[logger/level](#)

Returns or sets the log level of logger with the specified ID.

[section](#)

[top](#)

logger/logger

```
(logger id)
```

Returns the logger with the specified ID.

The returned logger is a function with 1, 2, 3, and 4 arguments to log messages.

1-arg log fn: `(fn [msg] ...)` (logs at `:info` level)

2-arg log fn: `(fn [level msg] ...)`

3-arg log fn: (fn [level principal msg] ...)

4-arg log fn: (fn [level principal msg exc] ...)

The logger supports the levels `:debug`, `:info`, `:warn`, `:error`, and `:fatal`

If there is no logger with the specified id returns the default `:console` logger.

```
(do
  (load-module :logger)

  ;; create a file logger
  (logger/file-logger :venice "/data/logs/venice.log")

  ;; get the logger
  (def log (logger/logger :venice))

  ;; log some messages
  (log :info "test message")
  (log :info "kermit" "test message")
  (log :info "kermit" "test message" (ex :VncException "test")))
```

SEE ALSO

[logger/console-logger](#)

Create a console logger with either the default log message formatter or a custom formatter.

[logger/file-logger](#)

Create a file logger. The logger is identified by it's ID.

[logger/format-level](#)

Formats a log level. The 'level' must be a keyword!

[logger/format-ex](#)

Formats an exception. The exception may be a Venice exception or a Java exception.

[section](#)

[top](#)

logger/requires-rotation?

(requires-rotation?)

Returns `true` if there is at least on logger that has log file rotation enabled.

```
(do
  (load-module :logger)

  ;; disable auto start (auto start is enabled by default)
  (logger/enable-auto-start-rotation-scheduler false)

  (logger/file-logger :venice "/logs/venice.log"
    :rotate-mode :daily
    :rotate-dir "/logs/archive")

  (def log (logger/logger :venice))

  (log :info "test message 1")
  (log :info "test message 2")

  (logger/requires-rotation?))
```


SEE ALSO

[logger/rotation-scheduler-running?](#)

Returns true if a rotation scheduler is running else false.

[logger/enable-auto-start-rotation-scheduler](#)

Enable or disable the auto starting of the rotation scheduler. Default is deactivated.

[logger/start-rotation-scheduler](#)

Manually start the rotation scheduler.

[logger/rotate](#)

Manually rotates the log file of the logger with the ID 'id' if the logger has file rotation enabled.

[logger/rotate-all](#)

Manually rotates the log files of all loggers that have file rotation enabled.

[section](#)

[top](#)

logger/rotate

(rotate id)

Manually rotates the log file of the logger with the ID 'id' if the logger has file rotation enabled.

```
(do
  (load-module :logger)

  ;; disable auto start (auto start is enabled by default)
  (logger/enable-auto-start-rotation-scheduler false)

  (logger/file-logger :venice "/logs/venice.log"
    :rotate-mode :daily
    :rotate-dir "/logs/archive")

  (def log (logger/logger :venice))

  (log :info "test message 1")
  (log :info "test message 2")

  ;; rotate the :venice logger manually
  (logger/rotate :venice))
```

SEE ALSO

[logger/rotation-scheduler-running?](#)

Returns true if a rotation scheduler is running else false.

[logger/enable-auto-start-rotation-scheduler](#)

Enable or disable the auto starting of the rotation scheduler. Default is deactivated.

[logger/start-rotation-scheduler](#)

Manually start the rotation scheduler.

[logger/rotate-all](#)

Manually rotates the log files of all loggers that have file rotation enabled.

[logger/requires-rotation?](#)

Returns true if there is at least on logger that has log file rotation enabled.

[section](#)

[top](#)

logger/rotate-all

```
(rotate-all)
```

Manually rotates the log files of all loggers that have file rotation enabled.

```
(do
  (load-module :logger)

  ;; disable auto start (auto start is enabled by default)
  (logger/enable-auto-start-rotation-scheduler false)

  (logger/file-logger :venice "/logs/venice.log"
    :rotate-mode :daily
    :rotate-dir "/logs/archive")

  (def log (logger/logger :venice))

  (log :info "test message 1")
  (log :info "test message 2")

  ;; rotate all loggers manually
  (logger/rotate-all))
```

SEE ALSO

[logger/rotation-scheduler-running?](#)

Returns true if a rotation scheduler is running else false.

[logger/enable-auto-start-rotation-scheduler](#)

Enable or disable the auto starting of the rotation scheduler. Default is deactivated.

[logger/start-rotation-scheduler](#)

Manually start the rotation scheduler.

[logger/rotate](#)

Manually rotates the log file of the logger with the ID 'id' if the logger has file rotation enabled.

[logger/requires-rotation?](#)

Returns true if there is at least on logger that has log file rotation enabled.

[section](#)

[top](#)

logger/rotation-scheduler-running?

```
(rotation-scheduler-running?)
```

Returns `true` if a rotation scheduler is running else `false`.

The scheduler is started when the first file logger with enabled file rotation is created!

```
(do
  (load-module :logger)

  (logger/file-logger :venice "/logs/venice.log"
    :rotate-mode :daily
    :rotate-dir "/logs/archive")
```

```
;; check after creation of the first file logger with file rotation
(logger/rotation-scheduler-running?))
```

SEE ALSO

[logger/enable-auto-start-rotation-scheduler](#)

Enable or disable the auto starting of the rotation scheduler. Default is deactivated.

[logger/start-rotation-scheduler](#)

Manually start the rotation scheduler.

[logger/rotate](#)

Manually rotates the log file of the logger with the ID 'id' if the logger has file rotation enabled.

[logger/rotate-all](#)

Manually rotates the log files of all loggers that have file rotation enabled.

[logger/requires-rotation?](#)

Returns true if there is at least on logger that has log file rotation enabled.

[section](#)

[top](#)

logger/start-rotation-scheduler

```
(start-rotation-scheduler)
```

Manually start the rotation scheduler.

Note: Consider to use `(logger/enable-auto-start-rotation-scheduler true)` instead of manually starting the scheduler.

```
(do
  (load-module :logger)

  ;; disable auto start (auto start is enabled by default)
  (logger/enable-auto-start-rotation-scheduler false)

  ;; start the scheduler manually
  (logger/start-rotation-scheduler)

  (logger/file-logger :venice "/logs/venice.log"
                      :rotate-mode :daily
                      :rotate-dir "/logs/archive")

  (def log (logger/logger :venice))

  (log :info "test message 1")
  (log :info "test message 2"))
```

SEE ALSO

[logger/rotation-scheduler-running?](#)

Returns true if a rotation scheduler is running else false.

[logger/enable-auto-start-rotation-scheduler](#)

Enable or disable the auto starting of the rotation scheduler. Default is deactivated.

[logger/rotate](#)

Manually rotates the log file of the logger with the ID 'id' if the logger has file rotation enabled.

[logger/rotate-all](#)

Manually rotates the log files of all loggers that have file rotation enabled.

logger/requires-rotation?

Returns true if there is at least on logger that has log file rotation enabled.

[section](#)[top](#)

logo

(logo)

Returns the Venice logo, a map with the keys `:name`, `:mimetype`, and `:data`

(logo)

```
(let [l (logo)]
  (io/spit (io/file (:name l)) (:data l)))
```

[section](#)[top](#)

long

(long x)

Converts to long

```
(long 1)
=> 1
```

```
(long nil)
=> 0
```

```
(long false)
=> 0
```

```
(long true)
=> 1
```

```
(long 1.2F)
=> 1
```

```
(long 1.2)
=> 1
```

```
(long 1.2M)
=> 1
```

```
(long "1")
=> 1
```

```
(long (char "A"))
=> 65
```

[section](#)[top](#)

long-array

```
(long-array coll)
(long-array len)
(long-array len init-val)
```

Returns an array of Java primitive longs containing the contents of coll or returns an array with the given length and optional init value.

To create an array of :java.lang.Long use:

```
(make-array :java.lang.Long 3)
```

```
(long-array '(1 2 3))
=> [1, 2, 3]
```

```
(long-array '(1I 2 3.2 3.56M))
=> [1, 2, 3, 3]
```

```
(long-array 10)
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
(long-array 10 42)
=> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

SEE ALSO

[java-long-list](#)

Converts a Venice list/vector to a Java Long list

[section](#)

[top](#)

long?

```
(long? n)
```

Returns true if n is a long

```
(long? 4)
=> true
```

```
(long? 4I)
=> false
```

```
(long? 3.1)
=> false
```

```
(long? true)
=> false
```

```
(long? nil)
=> false
```

```
(long? {})
=> false
```

loop

```
(loop [bindings*] exprs*)
```

Evaluates the exprs and binds the bindings. Creates a recursion point with the bindings.

```
;; tail recursion
(loop [x 1] ; <--+ loop
      (when (< x 4) ; |
            (println x) ; |
            (recur (inc x)))) ; --+
1
2
3
=> nil

;; tail recursion (nested loops)
(loop [x 1] ; <-----+ outer loop
      (when (< x 4) ; |
            (loop [y 1] ; <--+ inner loop
                  (when (< y 3) ; | |
                        (println [x y]) ; | |
                        (recur (inc y)))) ; --+ |
            (recur (inc x)))) ; -----+
[1 1]
[1 2]
[2 1]
[2 2]
[3 1]
[3 2]
=> nil

;; tail recursion
(do
  (defn sum [n]
    (loop [cnt n acc 0]
      (if (zero? cnt)
          acc
          (recur (dec cnt) (+ acc cnt)))))
  (sum 10000))
=> 50005000
```

SEE ALSO

[recur](#)

Evaluates the exprs and rebinds the bindings of the recursion point to the values of the exprs. The recur expression must be at the ...

macro?

```
(macro? x)
```

Returns true if x is a macro

```
(macro? and)
=> true
```

[section](#)[top](#)

macroexpand

```
(macroexpand form)
```

If form represents a macro form, returns its expansion, else returns form.

To recursively expand all macros in a form use `(macroexpand-all form)`.

```
(macroexpand '(> c (+ 3) (* 2)))
=> (* (+ c 3) 2)
```

SEE ALSO

[defmacro](#)

Macro definition

[macroexpand-all](#)

Recursively expands all macros in the form.

[section](#)[top](#)

macroexpand-all

```
(macroexpand-all form)
```

Recursively expands all macros in the form.

```
(macroexpand-all '(and true true))
=> (let [cond__29403__auto true] (if cond__29403__auto true cond__29403__auto))

(macroexpand-all '(and true (or true false) true))
=> (let [cond__29431__auto true] (if cond__29431__auto (let [cond__29431__auto (let [cond__29432__auto true]
(if cond__29432__auto cond__29432__auto false))] (if cond__29431__auto true cond__29431__auto))
cond__29431__auto))

(macroexpand-all '(let [n 5] (cond (< n 0) -1 (> n 0) 1 :else 0)))
=> (let [n 5] (if (< n 0) -1 (if (> n 0) 1 (if :else 0 nil))))
```

SEE ALSO

[macroexpand](#)

If form represents a macro form, returns its expansion, else returns form.

[defmacro](#)

Macro definition

[section](#)[top](#)

macroexpand-on-load?

```
(macroexpand-on-load?)
```

Returns true if `macroexpand-on-load` feature is enabled else false.

The activation of `macroexpand-on-load` (upfront macro expansion) results in 3x to 15x better performance. Upfront macro expansion can be activated through the `!macroexpand` command in the REPL.

```
(macroexpand-on-load?)
```

```
=> false
```

[section](#)[top](#)

make-array

```
(make-array type len)
```

```
(make-array type dim &more-dims)
```

Returns an array of the given type and length

```
(str (make-array :long 5))
```

```
=> "[0, 0, 0, 0, 0]"
```

```
(str (make-array :java.lang.Long 5))
```

```
=> "[nil, nil, nil, nil, nil]"
```

```
(str (make-array :long 2 3))
```

```
=> "[[0 0 0], [0 0 0]]"
```

```
(aset (make-array :java.lang.Long 5) 3 9999)
```

```
=> [nil, nil, nil, 9999, nil]
```

[section](#)[top](#)

map

```
(map f coll colls*)
```

Applies `f` to the set of first items of each `coll`, followed by applying `f` to the set of second items in each `coll`, until any one of the `colls` is exhausted. Any remaining items in other `colls` are ignored.

Returns a transducer when no collection is provided.

Note: if Java collections are used the mapper converts all mapped items back to Java data types to keep Java compatibility as much as possible! To avoid this just convert the Java collection to a Venice collection. E.g.: `(into [] ...)`.

```
(map inc [1 2 3 4])
```

```
=> (2 3 4 5)
```

```
(map + [1 2 3 4] [10 20 30 40])
```

```
=> (11 22 33 44)
```

```
(map list '(1 2 3 4) '(10 20 30 40))
```

```
=> ((1 10) (2 20) (3 30) (4 40))
```

```
(map vector (lazy-seq 1 inc) [10 20 30 40])
```

```
=> ([1 10] [2 20] [3 30] [4 40])
```



```

(map (fn [[k v]] [k v]) {:a 1 :b 2})
=> ([:a 1] [:b 2])

(map (fn [e] [(key e) (inc (val e))]) {:a 1 :b 2})
=> ([:a 2] [:b 3])

(map inc #{1 2 3})
=> (2 3 4)

;; Venice enforces Java types when using java collections instead
;; of Venice collections!
;; -> The returned element type is a 'java.util.ArrayList'
;; and not a 'core/vector'
(->> (doto (. :java.util.ArrayList :new) (. :add 1) (. :add 2))
      (map (fn [x] [(inc x)])) ;; map to a 'core/vector'
      (first)
      (type))
=> :core/vector

;; Same example with a Venice collection!
;; -> The returned element type is a 'core/vector'
(->> [1 2]
      (map (fn [x] [(inc x)])) ;; map to a 'core/vector'
      (first)
      (type))
=> :core/vector

```

SEE ALSO

[filter](#)

Returns a collection of the items in coll for which (predicate item) returns logical true.

[reduce](#)

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

[map-indexed](#)

Returns a collection of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item of coll, etc. until ...

[tee](#)

Applies f to every item of each the coll and returns the unaltered coll.

[section](#)

[top](#)

map-entry

```
(map-entry key val)
```

Creates a new map entry

```

(map-entry :a 1)
=> [:a 1]

(key (map-entry :a 1))
=> :a

(val (map-entry :a 1))
=> 1

```

```
(entries {:a 1 :b 2 :c 3})
=> ([:a 1] [:b 2] [:c 3])
```

SEE ALSO

[map-entry?](#)

Returns true if m is a map entry

[entries](#)

Returns a collection of the map's entries.

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[key](#)

Returns the key of the map entry.

[val](#)

Returns the val of the map entry.

[section](#)[top](#)

map-entry?

```
(map-entry? m)
```

Returns true if m is a map entry

```
(map-entry? (map-entry :a 1))
=> true
```

```
(map-entry? (first (entries {:a 1 :b 2})))
=> true
```

SEE ALSO

[map-entry](#)

Creates a new map entry

[entries](#)

Returns a collection of the map's entries.

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[section](#)[top](#)

map-indexed

```
(map-indexed f coll)
```

Returns a collection of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item of coll, etc. until coll is exhausted. Returns a stateful transducer when no collection is provided.

```
(map-indexed (fn [idx val] [idx val]) [:a :b :c])
=> ([0 :a] [1 :b] [2 :c])
```

```
(map-indexed vector [:a :b :c])
=> ([0 :a] [1 :b] [2 :c])

;; start at index 1 instead of zero
(map-indexed #(vector (inc %1) %2) [:a :b :c])
=> ([1 :a] [2 :b] [3 :c])

(map-indexed vector "abcdef")
=> ([0 #\a] [1 #\b] [2 #\c] [3 #\d] [4 #\e] [5 #\f])

(map-indexed hash-map [:a :b :c])
=> ({0 :a} {1 :b} {2 :c})
```

SEE ALSO

[map](#)

Applies *f* to the set of first items of each coll, followed by applying *f* to the set of second items in each coll, until any one of the ...

[filter](#)

Returns a collection of the items in coll for which (predicate item) returns logical true.

[reduce](#)

f should be a function of 2 arguments. If *val* is not supplied, returns the result of applying *f* to the first 2 items in coll, then ...

[tee](#)

Applies *f* to every item of each the coll and returns the unaltered coll.

[section](#)

[top](#)

map-invert

```
(map-invert m)
```

Returns the map with the vals mapped to the keys.

```
(map-invert {:a 1 :b 2 :c 3})
=> {1 :a 2 :b 3 :c}
```

[section](#)

[top](#)

map-keys

```
(map-keys f m)
```

Applies function *f* to the keys of the map *m*.

```
(map-keys name {:a 1 :b 2 :c 3})
=> {"a" 1 "b" 2 "c" 3}
```

SEE ALSO

[map-vals](#)

Applies function *f* to the values of the map *m*.

[map-invert](#)

Returns the map with the vals mapped to the keys.

map-vals

```
(map-vals f m)
```

Applies function `f` to the values of the map `m`.

```
(map-vals inc {:a 1 :b 2 :c 3})  
=> {:a 2 :b 3 :c 4}
```

```
(map-vals :len {:a {:col 1 :len 10} :b {:col 2 :len 20} :c {:col 3 :len 30}})  
=> {:a 10 :b 20 :c 30}
```

SEE ALSO

[map-keys](#)

Applies function `f` to the keys of the map `m`.

[map-invert](#)

Returns the map with the vals mapped to the keys.

map?

```
(map? obj)
```

Returns true if `obj` is a map

```
(map? {:a 1 :b 2})  
=> true
```

mapcat

```
(mapcat fn & colls)
```

Returns the result of applying `concat` to the result of applying `map` to `fn` and `colls`. Thus function `fn` should return a collection.

```
(mapcat identity [[1 2 3] [4 5 6] [7 8 9]])  
=> (1 2 3 4 5 6 7 8 9)
```

```
(mapcat identity [[1 2 [3 4]] [5 6 [7 8]]])  
=> (1 2 [3 4] 5 6 [7 8])
```

```
(mapcat reverse [[3 2 1] [6 5 4] [9 8 7]])  
=> (1 2 3 4 5 6 7 8 9)
```

```

(mapcat list [:a :b :c] [1 2 3])
=> (:a 1 :b 2 :c 3)

(mapcat #(remove even? %) [[1 2] [2 2] [2 3]])
=> (1 3)

(mapcat #(repeat 2 %) [1 2])
=> (1 1 2 2)

(mapcat (fn [& x] [(apply + x)]) [10 11 12] [1 2 3])
=> (11 13 15)

(mapcat (juxt inc dec) [1 2 3 4])
=> (2 0 3 1 4 2 5 3)

;; Turn a frequency map back into a coll.
(mapcat (fn [[x n]] (repeat n x)) {:a 2 :b 1 :c 3})
=> (:a :a :a :b :c :c :c)

```

SEE ALSO

[map](#)

Applies *f* to the set of first items of each coll, followed by applying *f* to the set of second items in each coll, until any one of the ...

[flatten](#)

Takes any nested combination of collections (lists, vectors, etc.) and returns their contents as a single, flat sequence. (flatten ...

[section](#)

[top](#)

mapv

```
(mapv f coll colls*)
```

Returns a vector consisting of the result of applying *f* to the set of first items of each coll, followed by applying *f* to the set of second items in each coll, until any one of the colls is exhausted. Any remaining items in other colls are ignored.

```

(mapv inc [1 2 3 4])
=> [2 3 4 5]

(mapv + [1 2 3 4] [10 20 30 40])
=> [11 22 33 44]

(mapv vector [1 2 3 4] [10 20 30 40])
=> [[1 10] [2 20] [3 30] [4 40]]

```

SEE ALSO

[docoll](#)

Applies *f* to the items of the collection presumably for side effects. Returns nil.

[section](#)

[top](#)

match?

```
(match? s regex)
```

Returns true if the string `s` matches the regular expression `regex`.

The argument 'regex' may be a string representing a regular expression or a `java.util.regex.Pattern`.

See the functions in the 'regex' namespace if more than a simple regex match is required! E.g. `regex/matches?` performs much better on matching many strings against the same pattern:

```
(let [m (regex/matcher #"[0-9]+" "")]
  (filter #(regex/matches? m %) ["100" "1a1" "200"])))
```

```
(match? "1234" "[0-9]+")
=> true
```

```
(match? "1234ss" "[0-9]+")
=> false
```

```
(match? "1234" #"[0-9]+")
=> true
```

SEE ALSO

[not-match?](#)

Returns true if the string `s` does not match the regular expression `regex`.

[regex/matches?](#)

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

[regex/matches-not?](#)

Attempts to match the entire region against the pattern. Returns false if the patterns matches the string else true.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/find](#)

Returns the next regex match or nil if there is no further match. Returns nil if there is no match.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[section](#)

[top](#)

math/acos

```
(math/acos x)
```

Returns the arc cosine of a value; the returned angle is in the range `0.0` through `pi`

```
(math/acos 0.5)
=> 1.0471975511965979
```

SEE ALSO

[math/cos](#)

Returns the trigonometric cosine of an angle given in radians

[math/asin](#)

Returns the arc sine of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$

[math/atan](#)

Returns the arc tangent of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$.

[section](#)[top](#)

math/asin

(math/asin x)

Returns the arc sine of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$

```
(math/asin 0.8660254037844386)
=> 1.0471975511965976
```

SEE ALSO

[math/sin](#)

Returns the trigonometric sine of an angle given in radians

[math/acos](#)

Returns the arc cosine of a value; the returned angle is in the range 0.0 through π

[math/atan](#)

Returns the arc tangent of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$.

[section](#)[top](#)

math/atan

(math/atan x)

Returns the arc tangent of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$.

```
(math/atan 1.7320508075688767)
=> 1.0471975511965976
```

SEE ALSO

[math/tan](#)

Returns the trigonometric tangent of an angle given in radians

[math/asin](#)

Returns the arc sine of a value; the returned angle is in the range $-\pi/2$ through $\pi/2$

[math/acos](#)

Returns the arc cosine of a value; the returned angle is in the range 0.0 through π

[section](#)[top](#)

math/cos

(math/cos x)

Returns the trigonometric cosine of an angle given in radians

```
(math/cos (/ math/PI 3.0))  
=> 0.5000000000000001
```

SEE ALSO

[math/sin](#)

Returns the trigonometric sine of an angle given in radians

[math/tan](#)

Returns the trigonometric tangent of an angle given in radians

[section](#)

[top](#)

math/mean

```
(math/mean x)  
(math/mean x y)  
(math/mean x y & more)
```

Returns the mean value of the values

```
(math/mean 10 20 30)  
=> 20.0
```

```
(math/mean 1.4 3.6)  
=> 2.5
```

```
(math/mean 2.8M 6.4M)  
=> 4.600000000000000M
```

SEE ALSO

[math/median](#)

Returns the median of the values

[math/standard-deviation](#)

Returns the standard deviation of the values for data sample type :population or :sample.

[math/quantile](#)

Returns the quantile [0.0 .. 1.0] of the values

[math/quartiles](#)

Returns the quartiles (1st, 2nd, and 3rd) of the values

[section](#)

[top](#)

math/median

```
(math/median coll)
```

Returns the median of the values

```
(math/median '(3 1 2))  
=> 2.0
```



```
(math/median '(3 2 1 4))  
=> 2.5
```

```
(math/median '(3.6 1.4 4.8))  
=> 3.6
```

```
(math/median '(3.6M 1.4M 4.8M))  
=> 3.6M
```

SEE ALSO

[math/mean](#)

Returns the mean value of the values

[math/standard-deviation](#)

Returns the standard deviation of the values for data sample type :population or :sample.

[math/quantile](#)

Returns the quantile [0.0 .. 1.0] of the values

[math/quartiles](#)

Returns the quartiles (1st, 2nd, and 3rd) of the values

[section](#)

[top](#)

math/quantile

```
(math/quantile q coll)
```

Returns the quantile [0.0 .. 1.0] of the values

```
(math/quantile 0.5 '(3, 7, 8, 5, 12, 14, 21, 13, 18))  
=> 12.0
```

```
(math/quantile 0.5 '(3, 7, 8, 5, 12, 14, 21, 15, 18, 14))  
=> 13.0
```

SEE ALSO

[math/mean](#)

Returns the mean value of the values

[math/median](#)

Returns the median of the values

[math/standard-deviation](#)

Returns the standard deviation of the values for data sample type :population or :sample.

[math/quartiles](#)

Returns the quartiles (1st, 2nd, and 3rd) of the values

[section](#)

[top](#)

math/quartiles

```
(math/quartiles coll)
```

Returns the quartiles (1st, 2nd, and 3rd) of the values

```
(math/quartiles '(3, 7, 8, 5, 12, 14, 21, 13, 18))  
=> (6.0 12.0 16.0)
```

```
(math/quartiles '(3, 7, 8, 5, 12, 14, 21, 15, 18, 14))  
=> (7.0 13.0 15.0)
```

SEE ALSO

[math/mean](#)

Returns the mean value of the values

[math/median](#)

Returns the median of the values

[math/standard-deviation](#)

Returns the standard deviation of the values for data sample type :population or :sample.

[math/quantile](#)

Returns the quantile [0.0 .. 1.0] of the values

[section](#)

[top](#)

math/sin

```
(math/sin x)
```

Returns the trigonometric sine of an angle given in radians

```
(math/sin (/ math/PI 3.0))  
=> 0.8660254037844386
```

SEE ALSO

[math/cos](#)

Returns the trigonometric cosine of an angle given in radians

[math/tan](#)

Returns the trigonometric tangent of an angle given in radians

[section](#)

[top](#)

math/softmax

```
(math/softmax coll)
```

Softmax algorithm

See [Softmax](#)

```
(math/softmax [3.2 1.3 0.2 0.8])  
=> [0.7751495482986049 0.1159380476300716 0.03859242355646149 0.07031998051486205]
```

[section](#)

[top](#)

math/standard-deviation

```
(math/standard-deviation type coll)
```

Returns the standard deviation of the values for data sample type `:population` or `:sample`.

```
(math/standard-deviation :sample '(10 8 30 22 15))  
=> 9.055385138137417
```

```
(math/standard-deviation :population '(10 8 30 22 15))  
=> 8.099382692526634
```

```
(math/standard-deviation :sample '(1.4 3.6 7.8 9.0 2.2))  
=> 3.40587727318528
```

```
(math/standard-deviation :sample '(2.8M 6.4M 2.0M 4.4M))  
=> 1.942506971244462
```

SEE ALSO

[math/mean](#)

Returns the mean value of the values

[math/median](#)

Returns the median of the values

[math/quantile](#)

Returns the quantile [0.0 .. 1.0] of the values

[math/quartiles](#)

Returns the quartiles (1st, 2nd, and 3rd) of the values

[section](#)

[top](#)

math/tan

```
(math/tan x)
```

Returns the trigonometric tangent of an angle given in radians

```
(math/tan (/ math/PI 3.0))  
=> 1.7320508075688767
```

SEE ALSO

[math/sin](#)

Returns the trigonometric sine of an angle given in radians

[math/cos](#)

Returns the trigonometric cosine of an angle given in radians

[section](#)

[top](#)

math/to-degrees

```
(math/to-degrees x)
```

Converts an angle measured in radians to an approximately equivalent angle measured in degrees. The conversion from radians to degrees is generally inexact; users should not expect `(cos (to-radians 90.0))` to exactly equal 0.0

```
(math/to-degrees 3)
=> 171.88733853924697
```

```
(math/to-degrees 3.1415926)
=> 179.99999692953102
```

```
(math/to-degrees 3.1415926M)
=> 179.99999692953102
```

SEE ALSO

[math/to-radians](#)

Converts an angle measured in degrees to an approximately equivalent angle measured in radians. The conversion from degrees to radians ...

[section](#)

[top](#)

math/to-radians

```
(math/to-radians x)
```

Converts an angle measured in degrees to an approximately equivalent angle measured in radians. The conversion from degrees to radians is generally inexact.

```
(math/to-radians 90)
=> 1.5707963267948966
```

```
(math/to-radians 90.0)
=> 1.5707963267948966
```

```
(math/to-radians 90.0M)
=> 1.5707963267948966
```

SEE ALSO

[math/to-degrees](#)

Converts an angle measured in radians to an approximately equivalent angle measured in degrees. The conversion from radians to degrees ...

[section](#)

[top](#)

matrix/add-column-at-end

```
(matrix/add-column-at-end m c)
```

Add a column to a matrix after the last column.

```
(do
  (load-module :matrix)

  ;; | 1 2 3 | + | 4 8 | => | 1 2 3 4 |
```

```
;; | 5 6 7 | | 5 6 7 8 |

(matrix/add-column-at-end [[1 2 3] [5 6 7]] [4 8]))
=> [[1 2 3 4] [5 6 7 8]]
```

[section](#) [top](#)

matrix/add-column-at-start

```
(matrix/add-column-at-start m c)
```

Add a column to a matrix before the first column.

```
(do
  (load-module :matrix)

  ;; | 2 3 4 | + | 1 5 | => | 1 2 3 4 |
  ;; | 6 7 8 | | 5 6 7 8 |

  (matrix/add-column-at-start [[2 3 4] [6 7 8]] [1 5]))
=> [[1 2 3 4] [5 6 7 8]]
```

[section](#) [top](#)

matrix/add-row-at-end

```
(matrix/add-row-at-end m r)
```

Add a row to a matrix after the last row.

```
(do
  (load-module :matrix)

  ;; | 1 2 3 | + | 7 8 9 | => | 1 2 3 |
  ;; | 4 5 6 | | 4 5 6 |
  ;; | 7 8 9 |

  (matrix/add-row-at-end [[1 2 3] [4 5 6]] [7 8 9]))
=> [[1 2 3] [4 5 6] [7 8 9]]
```

[section](#) [top](#)

matrix/add-row-at-start

```
(matrix/add-row-at-start m r)
```

Add a row to a matrix before the first row.

```
(do
  (load-module :matrix)

  ;; | 4 5 6 | + | 1 2 3 | => | 1 2 3 |
  ;; | 7 8 9 | | 4 5 6 |
```

```
;; | 7 8 9 |

(matrix/add-row-at-start [[ 4 5 6 ] [7 8 9]] [1 2 3]))
=> [[1 2 3] [4 5 6] [7 8 9]]
```

[section](#)[top](#)

matrix/assoc-element

```
(matrix/assoc-element m row col val)
```

Replaces an element in the matrix

```
(do
  (load-module :matrix)
  (matrix/assoc-element [[1 2 3] [4 5 6]] 1 2 9))
=> [[1 2 3] [4 5 9]]
```

[section](#)[top](#)

matrix/column

```
(matrix/column m n)
```

Returns the matrix column n

```
(do
  (load-module :matrix)
  (matrix/column [[1 2 3] [4 5 6]] 1))
=> [2 5]
```

[section](#)[top](#)

matrix/columns

```
(matrix/columns m)
```

Returns the number of columns in the matrix

```
(do
  (load-module :matrix)
  (matrix/columns [[1 2 3]
                   [4 5 6]]))
=> 3
```

[section](#)[top](#)

matrix/element

```
(matrix/element m row col)
```

Returns the matrix element at the row and column

```
(do
  (load-module :matrix)
  (matrix/element [[1 2 3] [4 5 6]] 1 2))
=> 6
```

[section](#)

[top](#)

matrix/empty?

```
(matrix/empty? m)
```

Returns true if the matrix is empty else false

```
(do
  (load-module :matrix)
  (matrix/empty? []))
=> true
```

[section](#)

[top](#)

matrix/format

```
(matrix/format m)
(matrix/format m fmt)
```

Formats a matrix.

```
(println (matrix/format [[1 2] [3 4] [5 6]]))

| 1 2 |
| 3 4 |
| 5 6 |

(do
  (load-module :matrix)
  (println (matrix/format [[1 2] [3 14] [10 6]]))
  (println)
  (println (matrix/format [[1.8 2.0] [3.0 4.8] [5.1 6.8]]))
  (println)
  (println (matrix/format [[1.845 2.009] [3.054 4.889] [5.132 6.878]]
    (fn [x] (str/format "%.2f" x)))))

| 1 2 |
| 3 14 |
| 10 6 |

| 1.8 2.0 |
| 3.0 4.8 |
| 5.1 6.8 |

| 1.85 2.01 |
| 3.05 4.89 |
| 5.13 6.88 |
=> nil
```

matrix/remove-column

```
(matrix/remove-column m n)
```

Remove a column from a matrix.

```
(do
  (load-module :matrix)
  (matrix/remove-column [[2 3 4] [6 7 8]] 1))
=> [[2 4] [6 8]]
```

matrix/remove-row

```
(matrix/remove-row m n)
```

Remove a row from a matrix.

```
(do
  (load-module :matrix)
  (matrix/remove-row [[1 2] [3 4] [5 6]] 1))
=> [[1 2] [5 6]]
```

matrix/row

```
(matrix/row m n)
```

Returns the matrix row n

```
(do
  (load-module :matrix)
  (matrix/row [[1 2 3] [4 5 6]] 1))
=> [4 5 6]
```

matrix/rows

```
(matrix/rows m)
```

Returns the number of rows in the matrix

```
(do
  (load-module :matrix)
  (matrix/rows [[1 2 3]
```



```
[4 5 6]]))
```

```
=> 2
```

[section](#)[top](#)

matrix/transpose

```
(matrix/transpose m)
```

Transposes a matrix. A matrix is a vector of vectors `[[1 2] [3 4]]`

```
(do
  (load-module :matrix)

  ;; | 1 2 3 |   =>  | 1 4 |
  ;; | 4 5 6 |       | 2 5 |
  ;; |         |       | 3 6 |
  (matrix/transpose [[1 2 3]
                     [4 5 6]]))
=> [[1 4] [2 5] [3 6]]
```

[section](#)[top](#)

matrix/validate

```
(matrix/validate m)
```

Validates a matrix. A matrix is a vector of vectors `[[1 2] [3 4]]`

Returns the matrix if valid else throws an exception.

- A matrix must be an empty vector or a vector of vectors!
- All rows must have the same number of columns

```
(do
  (load-module :matrix)
  (matrix/validate [[1 2 3]
                   [4 5 6]]))
=> [[1 2 3] [4 5 6]]
```

[section](#)[top](#)

matrix/vector2d

```
(matrix/vector2d m)
```

Converts a 2D sequential collection into a 2D vector

```
(do
  (load-module :matrix)
  (matrix/vector2d (list (list 1 2 3) (list 4 5 6))))
=> [[1 2 3] [4 5 6]]
```

maven/artifact-filename

```
(maven/artifact-filename artifact file-suffix)
```

Returns the artifact file name

```
"org.knowm.xchart:xchart:3.8.6" -> "xchart-3.8.6.jar"
```

```
(do
  (load-module :maven)
  (maven/artifact-filename "org.knowm.xchart:xchart:3.8.6" ".jar"))

(do
  (load-module :maven)
  (maven/artifact-filename "org.knowm.xchart:xchart:3.8.6" "-sources.jar"))
```

SEE ALSO

[maven/parse-artifact](#)

Parses a Maven artifact

[maven/artifact-uri](#)

Returns the artifact URI

[maven/download](#)

Downloads an artifact in the format 'group-id:artifact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/get](#)

Downloads artifact in the format 'group-id:artifact-id:version' from a Maven repository. The artifact type 'type' is one of {jar, ...

maven/artifact-uri

```
(maven/artifact-uri artifact file-suffix)
(maven/artifact-uri artifact file-suffix repo)
```

Returns the artifact URI

```
"org.knowm.xchart:xchart:3.8.6" -> "https://repo1.maven.org/maven2/org/knowm/xchart/xchart-3.8.6.jar"
```

```
(do
  (load-module :maven)
  (maven/artifact-uri "org.knowm.xchart:xchart:3.8.6" ".jar"))

(do
  (load-module :maven)
  (maven/artifact-uri "org.knowm.xchart:xchart:3.8.6" "-sources.jar"))

(do
  (load-module :maven)
  (maven/artifact-uri "org.knowm.xchart:xchart:3.8.6" ".pom"))

(do
  (load-module :maven)
  (maven/artifact-uri "org.knowm.xchart:xchart:3.8.6"
```

```
".jar"
"https://repo1.maven.org/maven2"))
```

SEE ALSO

[maven/parse-artifact](#)

Parses a Maven artifact

[maven/artifact-filename](#)

Returns the artifact file name

[maven/download](#)

Downloads an artifact in the format 'group-id:artifact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/get](#)

Downloads artifact in the format 'group-id:artifact-id:version' from a Maven repository. The artifact type 'type' is one of {jar, ...

[section](#)

[top](#)

maven/dependencies

(maven/dependencies artifacts & options)

Returns the dependency tree of an artifact

Relies on the environment variable `MAVEN_HOME` to access Maven.

Options:

<code>:scope s</code>	A scope. <code>:compile</code> , <code>:provided</code> , <code>:runtime</code> , <code>:test</code> . Defaults to <code>:runtime</code>
<code>:verbose v</code>	if true invokes for verbose output else standard output. Defaults to false
<code>:excludes e</code>	A list of excluded dependencies
<code>:format f</code>	An output format. <code>:raw</code> , <code>:tree</code> , or <code>:list</code> . Defaults to <code>:tree</code>
<code>:print p</code>	If true prints the data else returns the data. Defaults to true.
<code>:print-pom p</code>	If true print the generated pom for debugging purposes. Defaults to false.
<code>:managed-dependencies d</code>	An optional list of managed dependencies (artifacts). See example 2.

The scope is one of:

- `:compile` - build, test and run
- `:provided` - build and test
- `:runtime` - test and run
- `:test` - compile and test

Excludes dependencies with the group ids (except for `:test` scope):

- `org.junit.*`
- `org.opentest4j`
- `org.apiguardian`
- `junit`

Example 1:

```
(maven/dependencies [ "org.knowm.xchart:xchart:3.8.6" ] )

org.knowm.xchart:xchart:jar:3.8.6::runtime
+- de.erichseifert.vectorgraphics2d:VectorGraphics2D:jar:0.13:runtime
+- de.rototor.pdfbox:graphics2d:jar:3.0.0:runtime
|   org.apache.pdfbox:pdfbox:jar:3.0.0:runtime
|   +- org.apache.pdfbox:pdfbox-io:jar:3.0.0:runtime
|   +- org.apache.pdfbox:fontbox:jar:3.0.0:runtime
```

```
| commons-logging:commons-logging:jar:1.2:runtime
com.madgag:animated-gif-lib:jar:1.4:runtime
```

Example 2:

lock down a transitive dependency to a specific version using Maven managed dependencies

```
(maven/dependencies ["org.knowm.xchart:xchart:3.8.6"]
  :managed-dependencies ["org.apache.pdfbox:pdfbox:2.0.27"])
```

```
org.knowm.xchart:xchart:jar:3.8.6::runtime
+- de.erichseifert.vectorgraphics2d:VectorGraphics2D:jar:0.13:runtime
+- de.rototor.pdfbox:graphics2d:jar:3.0.0:runtime
|   org.apache.pdfbox:pdfbox:jar:2.0.27:runtime
|   +- org.apache.pdfbox:fontbox:jar:2.0.27:runtime
|       commons-logging:commons-logging:jar:1.2:runtime
com.madgag:animated-gif-lib:jar:1.4:runtime
```

```
(do
  (load-module :maven)
  (maven/dependencies [ "org.knowm.xchart:xchart:3.8.6" ]))
```

```
(do
  (load-module :maven)
  (maven/dependencies [ "org.knowm.xchart:xchart:3.8.6" ]
    :scope :runtime
    :verbose true))
```

```
(do
  (load-module :maven)
  (maven/dependencies [ "org.knowm.xchart:xchart:3.8.6" ]
    :scope :runtime
    :format :list))
```

;; Install artifacts with transitive dependencies

```
(do
  (load-module :maven)
  (load-module :installer)
  (-> (maven/dependencies [ "com.openai:openai-java:4.30.0"
    "org.apache.poi:poi:5.4.0"
    "org.apache.poi:poi-ooxml:5.4.0"
    "org.apache.poi:poi-ooxml-full:5.4.0" ]
    :scope :runtime
    :format :list
    :print false)
    (installer/install :dir (repl/lib-dir)
      :silent false)))
```

[section](#) [top](#)

maven/download

(maven/download artifact options*)

Downloads an artifact in the format 'group-id:artifact-id:version' from a Maven repository. Can download any combination of the jar, sources, or pom artifacts to a directory.

Accepts a sequence of artifacts as well.

Options:

:jar {true,false}	download the jar, defaults to true
:sources {true,false}	download the sources, defaults to false

:pom {true,false}	download the pom, defaults to false
:dir path	download dir, defaults to "."
:repo maven-repo	a maven repo, defaults to "https://repo1.maven.org/maven2"
:silent {true,false}	if silent is true does not show a progress bar, defaults to true
:force {true,false}	if force is true download the artifact even if it exist already on the download dir, else skip the download if it exists. Defaults to true.

```

(do
  (load-module :maven)
  (maven/download "org.knowm.xchart:xchart:3.8.6"))

(do
  (load-module :maven)
  (maven/download "org.knowm.xchart:xchart:3.8.6"
    :sources true
    :pom true))

(do
  (load-module :maven)
  (maven/download "org.knowm.xchart:xchart:3.8.6"
    :dir "."
    :jar false
    :sources true))

(do
  (load-module :maven)
  (maven/download "org.knowm.xchart:xchart:3.8.6"
    :dir "."
    :sources true))

(do
  (load-module :maven)
  (maven/download "org.knowm.xchart:xchart:3.8.6"
    :dir "."
    :sources true
    :repo "https://repo1.maven.org/maven2"))

(do
  (load-module :maven)
  (maven/download "org.knowm.xchart:xchart:3.8.6"
    :dir "."
    :silent false))

(do
  (load-module :maven)
  ;; download all langchain4j artifacts
  (maven/download (maven/dependencies
    [ "dev.langchain4j:langchain4j:0.28.0" ]
    :format :list
    :scope :runtime
    :print false)
    :dir "."
    :silent false))

```

SEE ALSO

[maven/get](#)

Downloads artifact in the format 'group-id:artifact-id:version' from a Maven repository. The artifact type 'type' is one of {jar, ...

[maven/parse-artifact](#)

maven/get

(maven/get artifact type options*)

Downloads artifact in the format 'group-id:artifact-id:version' from a Maven repository. The artifact type 'type' is one of {`:jar`, `:sources`, `:pom`}. Returns the artifact as byte buffer.

Options:

`:repo maven-repo` a maven repo, defaults to "https://repo1.maven.org/maven2"
`:silent {true,false}` if silent is true does not show a progress bar, defaults to true

```
(do
  (load-module :maven)
  (maven/get "org.knowm.xchart:xchart:3.8.6" :jar))

(do
  (load-module :maven)
  (maven/get "org.knowm.xchart:xchart:3.8.6" :jar :silent false))

(do
  (load-module :maven)
  (maven/get "org.knowm.xchart:xchart:3.8.6" :sources))

(do
  (load-module :maven)
  (maven/get "org.knowm.xchart:xchart:3.8.6"
    :jar
    :repo "https://repo1.maven.org/maven2"))
```

SEE ALSO

[maven/download](#)

Downloads an artifact in the format 'group-id:artifact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/parse-artifact](#)

Parses a Maven artifact

maven/home-dir

(maven/home-dir)

Returns the Apache Maven home directory or nil if Maven is not installed.

If a REPL is active checks first for local Apache Maven installation in the REPL, if none is available checks the environment variable 'MAVEN_HOME'.

If a REPL is not active checks the environment variable 'MAVEN_HOME'.

```
(do
  (load-module :maven)
  (maven/home-dir))
```

SEE ALSO

[maven/mvn](#)

Runs a Maven command

[maven/version](#)

Runs the Maven version command and returns the Maven version (e.g. "3.9.15") parsed from the Maven command result.

[section](#)

[top](#)

maven/install

```
(maven/install)
(maven/install version)
```

Installs Apache Maven to {repl-home-dir}/tools/apache-maven-x.y.z

Installation is possible from within a REPL only!

Manually download Maven:

```
curl -O https://archive.apache.org/dist/maven/maven-3/3.9.15/binaries/apache-maven-3.9.15-bin.zip
```

```
(do
  (load-module :maven)
  (maven/install)) ;; installs default version 3.9.15
```

```
(do
  (load-module :maven)
  (maven/install "3.9.15"))
```

SEE ALSO

[maven/home-dir](#)

Returns the Apache Maven home directory or nil if Maven is not installed.

[maven/uninstall](#)

Uninstalls Apache Maven from {repl-home-dir}/tools

[section](#)

[top](#)

maven/mvn

```
(maven/mvn proj-dir & args)
```

Runs a Maven command

Relies on the environment variable `MAVEN_HOME` to access Maven.

```
(do
  (load-module :maven)
  (->> (maven/mvn "/Users/foo/projects/my-project" "compile")
    (println)))
```

```
(do
  (load-module :maven)
  (->> (maven/mvn "/Users/foo/projects/my-project" "-X" "package")
    (println)))
```

SEE ALSO

[maven/version](#)

Runs the Maven version command and returns the Maven version (e.g. "3.9.15") parsed from the Maven command result.

[maven/home-dir](#)

Returns the Apache Maven home directory or nil if Maven is not installed.

[section](#)

[top](#)

maven/parse-artifact

```
(maven/parse-artifact artifact)
```

Parses a Maven artifact

Form 1: "org.knowm.xchart:xchart:3.8.6"

```
{ :group-id      "org.knowm.xchart"
  :artifact-id   "xchart"
  :version       "3.8.6" }
```

Form 2: "org.knowm.xchart:jar:xchart:3.8.6"

```
{ :group-id      "org.knowm.xchart"
  :artifact-id   "xchart"
  :version       "3.8.6"
  :type          :jar }
```

Form 3: "org.knowm.xchart:jar:xchart:3.8.6:compile"

```
{ :group-id      "org.knowm.xchart"
  :artifact-id   "xchart"
  :version       "3.8.6"
  :type          :jar
  :scope         :compile }
```

```
(do
  (load-module :maven)
  (maven/parse-artifact "org.knowm.xchart:xchart:3.8.6"))
```

SEE ALSO

[maven/artifact-filename](#)

Returns the artifact file name

[maven/artifact-uri](#)

Returns the artifact URI

[maven/download](#)

Downloads an artifact in the format 'group-id:artifact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/get](#)

Downloads artifact in the format 'group-id:artifact-id:version' from a Maven repository. The artifact type 'type' is one of {jar, ...

[section](#)

[top](#)

maven/uninstall

(maven/uninstall)

Uninstalls Apache Maven from {repl-home-dir}/tools

Uninstallation is possible from within a REPL only!

```
(do
  (load-module :maven)
  (maven/uninstall))
```

SEE ALSO

[maven/home-dir](#)

Returns the Apache Maven home directory or nil if Maven is not installed.

[maven/install](#)

Installs Apache Maven to {repl-home-dir}/tools/apache-maven-x.y.z

[section](#)

[top](#)

maven/version

(maven/version)

Runs the Maven version command and returns the Maven version (e.g. "3.9.15") parsed from the Maven command result.

If run in the REPL tries to locate an installed Maven in the {REPL_HOME}/tools dir. If not found tries environment variable MAVEN_HOME to access Maven.

If not run in REPL relies on the environment variable MAVEN_HOME to access Maven.

```
(do
  (load-module :maven)
  (maven/version))
```

SEE ALSO

[maven/version-info](#)

Runs the Maven version command and prints the commands output.

[maven/mvn](#)

Runs a Maven command

[maven/home-dir](#)

Returns the Apache Maven home directory or nil if Maven is not installed.

[section](#)

[top](#)

maven/version-info

(maven/version-info)

Runs the Maven version command and prints the commands output.

If run in the REPL tries to locate an installed Maven in the `{REPL_HOME}/tools` dir. If not found tries environment variable `MAVEN_HOME` to access Maven.

If not run in REPL relies on the environment variable `MAVEN_HOME` to access Maven.

Example output:

```
Apache Maven 3.9.15 (98b2cdbfdb5f1ac8781f537ea9accaed7922349)
Maven home: /Users/.../venice/tools/apache-maven-3.9.15
Java version: 1.8.0_462, vendor: Azul Systems, Inc., runtime: /Library/Java/JavaVirtualMachines/zulu-8.jdk
/Contents/Home/jre
Default locale: en_CH, platform encoding: UTF-8
OS name: "mac os x", version: "26.4.1", arch: "aarch64", family: "mac"
```

```
(do
  (load-module :maven)
  (maven/version-info))
```

SEE ALSO

[maven/version](#)

Runs the Maven version command and returns the Maven version (e.g. "3.9.15") parsed from the Maven command result.

[maven/mvn](#)

Runs a Maven command

[maven/home-dir](#)

Returns the Apache Maven home directory or nil if Maven is not installed.

[section](#)

[top](#)

max

```
(max x)
(max x y)
(max x y & more)
```

Returns the greatest of the values

```
(max 1)
=> 1

(max 1 2)
=> 2

(max 4 3 2 1)
=> 4

(max 1I 2I)
=> 2I

(max 1.0)
=> 1.0

(max 1.0 2.0)
=> 2.0

(max 4.0 3.0 2.0 1.0)
=> 4.0
```

```
(max 1.0M)
```

```
=> 1.0M
```

```
(max 1.0M 2.0M)
```

```
=> 2.0M
```

```
(max 4.0M 3.0M 2.0M 1.0M)
```

```
=> 4.0M
```

```
(max 1.0M 2)
```

```
=> 2
```

SEE ALSO

[min](#)

Returns the smallest of the values

[clamp](#)

Restricts a given value between a lower and upper bound. In this way, it acts like a combination of the min and max functions.

[section](#)[top](#)

mbean/attribute

```
(mbean/attribute object-name attribute-name)
```

Returns the value of a Java MBean attribute.

```
// Java MBean example
public interface HelloMBean {
    int getMaxCount();
    void setMaxCount(int c);
}

public class Hello implements HelloMBean {
    @Override
    public int getMaxCount() {
        return maxCount;
    }

    @Override
    public void setMaxCount(int c) {
        maxCount = c;
    }

    private int maxCount = 42;
}
```

```
(-> (mbean/object-name "java.lang:type=OperatingSystem")
    (mbean/attribute :ProcessCpuLoad))
```

```
(-> (mbean/object-name "java.lang:type=OperatingSystem")
    (mbean/attribute :SystemCpuLoad))
```

```
;; static MBean
(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
```

```
(mbean/register bean name)
(mbean/attribute name :MaxCount)))

;; dynamic MBean
(do
  (let [bean (atom (hash-map :count 10))
        name (mbean/object-name "venice:type=Data")]
    (mbean/register-dynamic bean name)
    (mbean/attribute name :count)))
```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/attribute!

```
(mbean/attribute! object-name attribute-name attribute-value)
```

Set the value of a Java MBean attribute.

```
;; static MBean
(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
    (mbean/register bean name)
    (mbean/attribute! name :MaxCount 64I)
    (mbean/attribute name :MaxCount)))

;; dynamic MBean
(do
  (let [bean (atom (hash-map :count 10))
        name (mbean/object-name "venice:type=Data")]
```

```
(mbean/register-dynamic bean name)
(mbean/attribute! name :count 20)
(mbean/attribute name :count)))
```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/info

```
(mbean/info object-name)
```

Return the MBean info of a Java MBean.

Example MBean:

```
public interface HelloMBean {
    void sayHello();
    int add(int x, int y);
    int getMaxCount();
    void setMaxCount(int c);
}
```

Example MBean info:

```
{:classname "com.github.jlangch.venice.demo.mbean.Hello"
 :notifications {}
 :operations {:add {:parameters {:p1 {:descriptor {}
                                     :type "int"
                                     :description ""}
                               :p2 {:descriptor {}
                                     :type "int"
                                     :description ""}}}
              :descriptor {}
              :return-type "int"
              :description "Operation exposed for management"}
 :sayHello {:parameters {}
            :descriptor {}}
```

```

        :return-type "void"
        :description "Operation exposed for management"}}
```

`mbean/info` `name`)

```

        :type "int"
        :description "Attribute exposed for management"}}
```

`mbean/register` `bean` `name`)

```

        :parameters {}
        :descriptor {}
        :description "Public constructor of the MBean"}}
```

`mbean/info` `name`)

```

        :description "Information on the management interface of the MBean"}
```

```

(let [name (mbean/object-name "java.lang:type=OperatingSystem")]
  (mbean/info name))
```

```

(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
    (mbean/register bean name)
    (mbean/info name)))
```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/invoke

```

(mbean/invoke object-name operation params)
(mbean/invoke object-name operation params signature)
```

Invoke a Java MBean operation and return its result value.

Supported for static MBeans only!

```

// Java MBean example
public interface HelloMBean {
  void sayHello();
```

```

    int add(int x, int y);
}

public class Hello implements HelloMBean {
    @Override
    public void sayHello() {
        System.out.println("Hello, world!");
    }

    @Override
    public int add(int x, int y) {
        return x + y;
    }
}

```

```

(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
    (mbean/register bean name)
    ;; use an explicit operation signature
    (mbean/invoke name :add [1I 2I] ["int" "int"])))

(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
    (mbean/register bean name)
    ;; use the :add operation signature from the MBeanInfo
    (mbean/invoke name :add [1I 2I])))

```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

[mbean/jmx-connector-server-alive?](#)

```
(mbean/jmx-connector-server-alive? server)
```

Returns true if the JMX connector server is running else false.

```
(let [cs (mbean/jmx-connector-server-start 9999)]  
  (mbean/jmx-connector-server-alive? cs)  
  (mbean/jmx-connector-server-stop cs))
```

SEE ALSO

[mbean/jmx-connector-server-start](#)

Start a JMX connector server on a given port.

[mbean/jmx-connector-server-stop](#)

Stop a JMX connector server

[section](#)

[top](#)

mbean/jmx-connector-server-start

```
(mbean/jmx-connector-server-start port)
```

Start a JMX connector server on a given port.

A connector server is required if the MBeans should be accessible from a remote Java VM.

It is strongly recommended to configure SSL and authentication for the JMX connector!

```
(let [registry (mbean/jmx-connector-server-start 9999)]  
  (mbean/jmx-connector-server-alive? registry)  
  (mbean/jmx-connector-server-stop registry))
```

SEE ALSO

[mbean/jmx-connector-server-stop](#)

Stop a JMX connector server

[mbean/jmx-connector-server-alive?](#)

Returns true if the JMX connector server is running else false.

[section](#)

[top](#)

mbean/jmx-connector-server-stop

```
(mbean/jmx-connector-server-stop server)
```

Stop a JMX connector server

```
(let [cs (mbean/jmx-connector-server-start 9999)]  
  (mbean/jmx-connector-server-alive? cs)  
  (mbean/jmx-connector-server-stop cs))
```

SEE ALSO

[mbean/jmx-connector-server-start](#)

Start a JMX connector server on a given port.

[mbean/jmx-connector-server-alive?](#)

Returns true if the JMX connector server is running else false.

[section](#)[top](#)

mbean/memory-mxbean

(mbean/memory-mxbean)

Returns the Java Memory MXBean

([mbean/memory-mxbean](#))

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/operating-system-mxbean](#)

Returns the Java Operating System MXBean

[mbean/runtime-mxbean](#)

Returns the Java Runtime MXBean

[section](#)[top](#)

mbean/object-name

(mbean/object-name name)

(mbean/object-name domain key value)

Creates a Java MBean object name

([mbean/object-name](#) "java.lang:type=OperatingSystem")

([mbean/object-name](#) "java.lang" "type" "OperatingSystem")

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/operating-system-mxbean

([mbean/operating-system-mxbean](#))

Returns the Java Operating System MXBean

([mbean/operating-system-mxbean](#))

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/runtime-mxbean](#)

Returns the Java Runtime MXBean

[mbean/memory-mxbean](#)

Returns the Java Memory MXBean

[section](#)

[top](#)

mbean/platform-mbean-server

([mbean/platform-mbean-server](#))

Returns the Java platform MBean server

([mbean/platform-mbean-server](#))

SEE ALSO

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[mbean/operating-system-mxbean](#)

Returns the Java Operating System MXBean

[mbean/runtime-mxbean](#)

Returns the Java Runtime MXBean

[mbean/memory-mxbean](#)

Returns the Java Memory MXBean

[section](#)

[top](#)

mbean/query-mbean-object-names

(mbean/query-mbean-object-names)

Returns a list of the registered Java MBean object names.

([mbean/query-mbean-object-names](#))

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/register

(mbean/register mbean name)

Register a Java MBean

```
(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
    (mbean/register bean name)))
```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/register-dynamic

```
(mbean/register-dynamic bean name)
```

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

```
(do
  (let [bean (atom {:count 10})
        name (mbean/object-name "venice:type=Data")]
    (mbean/register-dynamic bean name)
    (println ":count " (mbean/attribute name :count))
    (mbean/attribute! name :count 20)
    (println ":count " (mbean/attribute name :count))))
```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/query-mbean-object-names](#)

Returns a list of the registered Java MBean object names.

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/info](#)

Return the MBean info of a Java MBean.

[mbean/attribute](#)

Returns the value of a Java MBean attribute.

[mbean/attribute!](#)

Set the value of a Java MBean attribute.

[mbean/invoke](#)

Invoke a Java MBean operation and return its result value.

[mbean/register](#)

Register a Java MBean

[mbean/unregister](#)

Unregister a Java MBean

[section](#)

[top](#)

mbean/runtime-mxbean

(mbean/runtime-mxbean)

Returns the Java Runtime MXBean

([mbean/runtime-mxbean](#))

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/operating-system-mxbean](#)

Returns the Java Operating System MXBean

[mbean/memory-mxbean](#)

Returns the Java Memory MXBean

[section](#)

[top](#)

mbean/unregister

(mbean/unregister name)

Unregister a Java MBean

```
(do
  (import :com.github.jlangch.venice.demo.mbean.Hello)
  (let [bean (. :Hello :new)
        name (mbean/object-name "venice:type=Hello")]
    (mbean/register bean name)
    (mbean/unregister name)))
```

SEE ALSO

[mbean/platform-mbean-server](#)

Returns the Java platform MBean server

[mbean/object-name](#)

Creates a Java MBean object name

[mbean/register](#)

Register a Java MBean

[mbean/register-dynamic](#)

Register a bean (a map wrapped by an atom or a volatile) as a dynamic Java MBean.

[section](#)

[top](#)

memoize

(memoize f)

Returns a memoized version of a referentially transparent function.

Note:

Use memoization for expensive calculations. If used with fast calculations it has the opposite effect and can slow it down actually!

```
(do
  (def fibonacci
    (memoize
      (fn [n]
        (cond
          (<= n 0) 0
          (< n 2) 1
          :else (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))

  (time (fibonacci 25)))
```

Elapsed time: 1.27ms

=> 75025

```
(do
  (defn test [a b]
    (println (str "calculating a=" a ", b=" b))
    (+ a b))

  (def test-memo (memoize test))

  (test-memo 1 1)
  (test-memo 1 2)
  (test-memo 1 1)
  (test-memo 1 2)
  (test-memo 1 1))
```

calculating a=1, b=1

calculating a=1, b=2

=> 2

SEE ALSO

[delay](#)

Takes a body of expressions and yields a Delay object that will invoke the body only the first time it is forced (with force or deref ...

[section](#)

[top](#)

merge

(merge & maps)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from the latter (left-to-right) will be the mapping in the result.

```
(merge {:a 1 :b 2 :c 3} {:b 9 :d 4})  
=> {:a 1 :b 9 :c 3 :d 4}
```

```
(merge {:a 1} nil)  
=> {:a 1}
```

```
(merge nil {:a 1})  
=> {:a 1}
```

```
(merge nil nil)  
=> nil
```

SEE ALSO

[merge-with](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping(s) from ...

[merge-deep](#)

Recursively merges maps.

[into](#)

Returns a new coll consisting of to coll with all of the items of from coll conjoined.

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[section](#)

[top](#)

merge-deep

```
(merge-deep values)  
(merge-deep strategy & values)
```

Recursively merges maps.

If the first parameter is a keyword it defines the strategy to use when merging non-map collections. Options are:

1. *:replace*, the default, the last value is used
2. *:into*, if the value in every map is a collection they are concatenated using `into`. Thus the type of (first) value is maintained.

```
(merge-deep {:a {:c 2}} {:a {:b 1}})  
=> {:a {:b 1 :c 2}}
```

```
(merge-deep :replace {:a [1]} {:a [2]})  
=> {:a [2]}
```

```
(merge-deep :into {:a [1]} {:a [2]})  
=> {:a [1 2]}
```

```
(merge-deep {:a 1} nil)  
=> nil
```

SEE ALSO

[merge](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from ...

[merge-with](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping(s) from ...

[section](#)[top](#)

merge-with

```
(merge-with f & maps)
```

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping(s) from the latter (left-to-right) will be combined with the mapping in the result by calling (f val-in-result val-in-latter).

```
(merge-with + {:a 1 :b 2} {:a 9 :b 98 :c 0})  
=> {:a 10 :b 100 :c 0}
```

```
(merge-with into {:a [1] :b [2]} {:b [3 4] :c [5 6]})  
=> {:a [1] :b [2 3 4] :c [5 6]}
```

SEE ALSO

[merge](#)

Returns a map that consists of the rest of the maps conj-ed onto the first. If a key occurs in more than one map, the mapping from ...

[merge-deep](#)

Recursively merges maps.

[section](#)[top](#)

meta

```
(meta obj)
```

Returns the metadata of obj, returns nil if there is no metadata.

```
(meta (vary-meta [1 2] assoc :foo 3))  
=> {:foo 3 :line 70 :column 28 :file "example"}
```

SEE ALSO

[vary-meta](#)

Returns a copy of the object obj, with (apply f (meta obj) args) as its metadata.

[with-meta](#)

Returns a copy of the object obj, with a map m as its metadata.

[var-val-meta](#)

Returns the var's value meta data.

[var-sym-meta](#)

Returns the var's symbol meta data.

[section](#)[top](#)

mimetypes/probe-content-type


```
(probe-content-type f)
```

Probes the content type of a file.

The function uses a built-in "mime.types" data file to lookup the file's mimetype based on the file extension.

f must be a string or a :java.io.File.

Returns `nil` if a mapping is not defined.

```
(do
  (load-module :mimetypes)
  (mimetypes/probe-content-type "foo.png"))
=> "image/png"
```

[section](#)[top](#)

min

```
(min x)
(min x y)
(min x y & more)
```

Returns the smallest of the values

```
(min 1)
=> 1
```

```
(min 1 2)
=> 1
```

```
(min 4 3 2 1)
=> 1
```

```
(min 1I 2I)
=> 1I
```

```
(min 1.0)
=> 1.0
```

```
(min 1.0 2.0)
=> 1.0
```

```
(min 4.0 3.0 2.0 1.0)
=> 1.0
```

```
(min 1.0M)
=> 1.0M
```

```
(min 1.0M 2.0M)
=> 1.0M
```

```
(min 4.0M 3.0M 2.0M 1.0M)
=> 1.0M
```

```
(min 1.0M 2)
=> 1.0M
```

SEE ALSO

[max](#)

Returns the greatest of the values

[clamp](#)

Restricts a given value between a lower and upper bound. In this way, it acts like a combination of the min and max functions.

[section](#)

[top](#)

mod

```
(mod n d)
```

Modulus of n and d.

```
(mod 10 4)
=> 2
```

```
(mod -1 5)
=> 4
```

```
(mod 10I 4I)
=> 2I
```

SEE ALSO

[mod-floor](#)

floor a number towards 0 to the nearest multiple of a number

[section](#)

[top](#)

mod-floor

```
(mod-floor n m)
```

floor a number towards 0 to the nearest multiple of a number

```
(mod-floor 9 3)
=> 9
```

```
(mod-floor 10 3)
=> 9
```

```
(mod-floor 11 3)
=> 9
```

```
(mod-floor -11 3)
=> -9
```

SEE ALSO

[mod](#)

Modulus of n and d.

[section](#)

[top](#)

module-name

(module-name class)

Returns the Java module name of a class.

([module-name](#) ([class](#) :[java.util.ArrayList](#)))

SEE ALSO

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[class-name](#)

Returns the Java class name of a class.

[section](#)

[top](#)

modules

(modules)

Lists the available Venice modules

SEE ALSO

[load-module](#)

Loads a Venice predefined extension module.

[doc](#)

Prints documentation for a var or special form given x as its name. Prints the definition of custom types.

[ns-list](#)

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

[section](#)

[top](#)

multipart/http-content-type-header

(http-content-type-header)

Returns the HTTP content type header value for *multipart/form-data* HTTP requests.

The `multipart/render` function uses this boundary.

E.g: Content-Type: multipart/form-data; boundary=1234567890N

```
(do
  (load-module :multipart ['multipart :as 'm])

  (m/http-content-type-header))
```

SEE ALSO

[multipart/render](#)

Renders a map of named parts as multipart/form-data format.

[multipart/parse](#)

Parses a multipart bytebuf.

[section](#)[top](#)

multipart/parse

```
(parse buffer boundary)
```

Parses a multipart bytebuf.

```
(do
  (load-module :multipart ['multipart :as 'm])
  (load-module :hexdump ['hexdump :as 'h])

  (defn render []
    (m/render { "Part-1" "xxxxxxxxxxx"
                "Part-4" { :filename "data.xml"
                           :mimetype "application/xml"
                           :charset  "utf-8"
                           :data      "<user><name>foo</name></user>" }}}))

  (let [mp (render)]
    (m/parse mp (m/boundary))))

;; Returns a list of part maps:
;; ( { :name "Part-1"
;;     :filename nil
;;     :mimetype nil
;;     :charset nil
;;     :data [120 ... 120] ;; shortened for brevity
;;     :data-len 11 }
;;   { :name "Part-4"
;;     :filename "data.xml"
;;     :mimetype "application/xml"
;;     :charset "utf-8"
;;     :data [60 ... 62]   ;; shortened for brevity
;;     :data-len 29 })
```

SEE ALSO

[multipart/render](#)

Renders a map of named parts as multipart/form-data format.

[multipart/http-content-type-header](#)

Returns the HTTP content type header value for multipart/form-data HTTP requests.

[section](#)[top](#)

multipart/render

(render parts)

Renders a map of named parts as *multipart/form-data* format.

The part name must be a string and the part data may be of type:

- string
- string ("file:/user/foo/image.png" to reference a file)
- map (describing a part as :name, :mimetype, :data (string or bytebuf), and an optional charset) elements)
- io/file
- all other part data types are converted with `(str data)` to a string

Returns a bytebuf with all the rendered parts.

```
POST / HTTP/1.1
HOST: host.example.com
Connection: Keep-Alive
Content-Type: multipart/form-data; boundary=12345

--12345
Content-Disposition: form-data; name="notes"
Content-Type: text/plain; charset=utf-8

Lorem ipsum ...
--12345
Content-Disposition: form-data; name="foo"; filename="foo.json"
Content-Type: application/json; charset=utf-8

content of foo.json
--12345
Content-Disposition: form-data; name="image"; filename="picture.png"
Content-Type: image/png

content of picture.png
--12345--
```

See [multipart/form-data](#)

```
(do
  (load-module :multipart ['multipart :as 'm])

  (->> (m/render { "Part-1" "xxxxxxxxxxx"
                  "Part-2" "yyyyyyyyyyy" })
    (bytebuf-to-string)
    (println)))

(do
  (load-module :multipart ['multipart :as 'm])

  (m/render { "Part-1" "xxxxxxxxxxx"
              "Part-2" "file:/user/foo/image.png"
              "Part-3" (io/file "/user/foo/image.png")
              "Part-4" { :filename "data.xml"
                        :mimetype "application/xml"
                        :charset  :utf-8
                        :data      "<user><name>foo</name></user>" }}))
```

SEE ALSO

[multipart/parse](#)

Parses a multipart bytebuf.

[multipart/http-content-type-header](#)

Returns the HTTP content type header value for multipart/form-data HTTP requests.

[section](#)[top](#)

mutable-list

(mutable-list & items)

Creates a new mutable list containing the items.

The list is backed by `java.util.ArrayList` and is not thread-safe.

```
(mutable-list)
=> ()

(mutable-list 1 2 3)
=> (1 2 3)

(mutable-list 1 2 3 [:a :b])
=> (1 2 3 [:a :b])
```

[section](#)[top](#)

mutable-list?

(mutable-list? obj)

Returns true if obj is a mutable list

```
(mutable-list? (mutable-list 1 2))
=> true
```

[section](#)[top](#)

mutable-map

(mutable-map & keyvals)

(mutable-map map)

Creates a new mutable threadsafe map containing the items.

```
(mutable-map :a 1 :b 2)
=> {:a 1 :b 2}

(mutable-map {:a 1 :b 2})
=> {:a 1 :b 2}
```

[section](#)[top](#)

mutable-map?

```
(mutable-map? obj)
```

Returns true if obj is a mutable map

```
(mutable-map? (mutable-map :a 1 :b 2))  
=> true
```

[section](#)[top](#)

mutable-set

```
(mutable-set & items)
```

Creates a new mutable set containing the items.

```
(mutable-set)  
=> #{}  
  
(mutable-set nil)  
=> #{nil}  
  
(mutable-set 1)  
=> #{1}  
  
(mutable-set 1 2 3)  
=> #{1 2 3}  
  
(mutable-set [1 2] 3)  
=> #{3 [1 2]}
```

[section](#)[top](#)

mutable-set?

```
(mutable-set? obj)
```

Returns true if obj is a mutable-set

```
(mutable-set? (mutable-set 1))  
=> true
```

[section](#)[top](#)

mutable-vector

```
(mutable-vector & items)
```

Creates a new mutable threadsafe vector containing the items.

```
(mutable-vector)
=> []

(mutable-vector 1 2 3)
=> [1 2 3]

(mutable-vector 1 2 3 [:a :b])
=> [1 2 3 [:a :b]]
```

[section](#)

[top](#)

mutable-vector?

```
(mutable-vector? obj)
```

Returns true if obj is a mutable vector

```
(mutable-vector? (mutable-vector 1 2))
=> true
```

[section](#)

[top](#)

name

```
(name x)
```

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

```
(name 'foo) ;; symbol
=> "foo"

(name 'user/foo) ;; symbol
=> "foo"

(name (symbol "user/foo")) ;; symbol
=> "foo"

(name :foo) ;; keyword
=> "foo"

(name :user/foo) ;; keyword
=> "foo"

(name +) ;; function
=> "+"

(name str/digit?) ;; function
=> "digit?"

(name "ab/text") ;; string
=> "ab/text"
```



```
(name (. :java.time.Month :JANUARY)) ;; java enum
=> "JANUARY"
```

SEE ALSO

[qualified-name](#)

Returns the qualified name String of a string, symbol, keyword, or function

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[fn-name](#)

Returns the qualified name of a function or macro

[section](#)[top](#)

namespace

```
(namespace x)
```

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

Throws an exception if x does not support namespaces like `(namespace 2)`.

```
(namespace 'user/foo) ;; symbol
=> "user"
```

```
(namespace (symbol "user/foo")) ;; symbol
=> "user"
```

```
(namespace :user/foo) ;; keyword
=> "user"
```

```
(namespace str/digit?) ;; function
=> "str"
```

```
(namespace *ns*) ;; symbol
=> "user"
```

SEE ALSO

[name](#)

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

[fn-name](#)

Returns the qualified name of a function or macro

[ns](#)

Opens a namespace.

[*ns*](#)

The current namespace

[var-ns](#)

Returns the namespace of the var's symbol.

[section](#)[top](#)

nan?

```
(nan? x)
```

Returns true if x is a NaN else false. x must be a double!

```
(nan? 0.0)
```

```
=> false
```

```
(nan? (/ 0.0 0))
```

```
=> true
```

```
(nan? (sqrt -1))
```

```
=> true
```

```
(pr (sqrt -1))
```

```
:NaN
```

```
=> nil
```

SEE ALSO

[infinite?](#)

Returns true if x is infinite else false. x must be a double!

[double](#)

Converts to double

[section](#)

[top](#)

nano-time

```
(nano-time)
```

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

```
(nano-time)
```

```
=> 28961090029500
```

```
(let [t (nano-time)
```

```
      _ (sleep 100)
```

```
      e (nano-time)]
```

```
  (format-nano-time (- e t) :precision 2))
```

```
=> "105.05ms"
```

SEE ALSO

[current-time-millis](#)

Returns the current time in milliseconds.

[format-nano-time](#)

Formats a time given in nanoseconds as long or double.

[section](#)

[top](#)

neg?

(neg? x)

Returns true if x smaller than zero else false

```
(neg? -3)  
=> true
```

```
(neg? 3)  
=> false
```

```
(neg? -3I)  
=> true
```

```
(neg? -3.2F)  
=> true
```

```
(neg? -3.2)  
=> true
```

```
(neg? -3.2M)  
=> true
```

SEE ALSO

[zero?](#)

Returns true if x zero else false

[pos?](#)

Returns true if x greater than zero else false

[negate](#)

Negates x

[section](#)

[top](#)

negate

(negate x)

Negates x

```
(negate 10)  
=> -10
```

```
(negate 10I)  
=> -10I
```

```
(negate 1.23)  
=> -1.23
```

```
(negate 1.23M)  
=> -1.23M
```

SEE ALSO

[abs](#)

Returns the absolute value of the number

sgn

sgn function for a number.

[section](#)

[top](#)

newer-version?

```
(newer-version? v1 v2)
```

Returns `true` if v1 is newer than v2 else `false`.

Compares the fields: 'major', 'minor', 'patch' and 'suffix'. 'major', 'minor', and 'patch' are compared as integer numbers. 'suffix' is compared in lexicographical order.

A missing 'minor' or 'patch' level is treated as 0. A missing suffix is treated as empty string.

```
(newer-version? "1.23" "1.20")
```

```
=> true
```

```
(newer-version? "1.12.4" "1.2.56")
```

```
=> true
```

```
(newer-version? "1.12" "1.11.78")
```

```
=> true
```

```
(newer-version? "1.10.0" "1.10.0-rc1")
```

```
=> true
```

```
(newer-version? "1.10.0-rc1" "1.10.0-rc2")
```

```
=> false
```

```
(newer-version? "1.10.0-rc2" "1.10.0-rc1")
```

```
=> true
```

```
(newer-version? "1.10.0-alpha" "1.10.0-epsilon")
```

```
=> false
```

SEE ALSO

[version](#)

Returns the Venice version.

[latest](#)

Returns the latest available Venice version or nil if the internet is not available.

[parse-version](#)

Parses a version to its elements: major, minor, patch and suffix.

[section](#)

[top](#)

newline

```
(newline)
```

```
(newline os)
```

Without arg writes a platform-specific newline to the output channel that is the current value of `*out*`. With arg writes a newline to the passed stream that must be a subclass of either `:java.io.PrintStream` or `:java.io.Writer`.

Returns `nil`.

```
(newline)
=> nil
```

```
(newline *out*)
=> nil
```

```
(newline *err*)
=> nil
```

SEE ALSO

[print](#)

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed stream `os` that must be a subclass of either ...

[println](#)

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed output stream `os` if given followed by a (newline).

[printf](#)

Without output stream prints formatted output as per `format` to the stream that is the current value of `*out*`. With a stream prints ...

[section](#)

[top](#)

nfirst

```
(nfirst coll n)
```

Returns a collection of the first `n` items

```
(nfirst nil 2)
=> ()
```

```
(nfirst [] 2)
=> []
```

```
(nfirst [1] 2)
=> [1]
```

```
(nfirst [1 2 3] 2)
=> [1 2]
```

```
(nfirst '() 2)
=> ()
```

```
(nfirst '(1) 2)
=> (1)
```

```
(nfirst '(1 2 3) 2)
=> (1 2)
```

```
(nfirst "abcdef" 2)
=> (#\a #\b)
```

```
(nfirst (lazy-seq 1 #(+ % 1)) 4)
=> (...)
```

SEE ALSO

[str/nfirst](#)

Returns a string of the n first characters of s.

[section](#)

[top](#)

nil?

```
(nil? x)
```

Returns true if x is nil, false otherwise

```
(nil? nil)
=> true
```

```
(nil? 0)
=> false
```

```
(nil? false)
=> false
```

SEE ALSO

[some?](#)

Returns true if x is not nil, false otherwise

[section](#)

[top](#)

nlast

```
(nlast coll n)
```

Returns a collection of the last n items

```
(nlast nil 2)
=> ()
```

```
(nlast [] 2)
=> []
```

```
(nlast [1] 2)
=> [1]
```

```
(nlast [1 2 3] 2)
=> [2 3]
```

```
(nlast '() 2)
=> ()
```

```
(nlast '(1) 2)
=> (1)
```

```
(nlast '(1 2 3) 2)
=> (2 3)
```

```
(nlast "abcdef" 2)
=> (#\e #\f)
```

SEE ALSO

[str/nlast](#)

Returns a string of the n last characters of s.

[section](#)

[top](#)

not

```
(not x)
```

Returns true if x is logical false, false otherwise.

```
(not true)
=> false
```

```
(not (== 1 2))
=> true
```

SEE ALSO

[and](#)

Ands the predicate forms

[or](#)

Ors the predicate forms

[section](#)

[top](#)

not-any?

```
(not-any? pred coll)
```

Returns false if the predicate is true for at least one collection item, true otherwise

```
(not-any? number? nil)
=> true
```

```
(not-any? number? [])
=> true
```

```
(not-any? number? [1 :a :b])
=> false
```

```
(not-any? number? [1 2 3])
=> false
```

```
(not-any? #(>= % 10) [1 5 10])  
=> false
```

SEE ALSO

[any?](#)

Returns true if the predicate is true for at least one collection item, false otherwise.

[every?](#)

Returns true if coll is a collection and the predicate is true for all collection items, false otherwise.

[not-every?](#)

Returns true if coll is a collection and the predicate is not true for all collection items, false otherwise.

[section](#)[top](#)

not-contains?

```
(not-contains? coll key)
```

Returns true if key is not present in the given collection, otherwise returns false.

```
(not-contains? #{:a :b} :c)  
=> true
```

```
(not-contains? {:a 1 :b 2} :c)  
=> true
```

```
(not-contains? [10 11 12] 1)  
=> false
```

```
(not-contains? [10 11 12] 5)  
=> true
```

```
(not-contains? "abc" 1)  
=> false
```

```
(not-contains? "abc" 5)  
=> true
```

SEE ALSO

[contains?](#)

Returns true if key is present in the given collection, otherwise returns false.

[section](#)[top](#)

not-empty?

```
(not-empty? x)
```

Returns true if x is not empty. Accepts strings, collections and bytebufs.


```
(not-empty? {:a 1})
=> true

(not-empty? [1 2])
=> true

(not-empty? '(1 2))
=> true

(not-empty? "abc")
=> true

(not-empty? nil)
=> false

(not-empty? "")
=> false
```

SEE ALSO

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[section](#)[top](#)

not-every?

```
(not-every? pred coll)
```

Returns true if coll is a collection and the predicate is not true for all collection items, false otherwise.

```
(not-every? number? nil)
=> false

(not-every? number? [])
=> true

(not-every? number? [1 2 3 4])
=> false

(not-every? number? [1 2 3 :a])
=> true

(not-every? #(>= % 10) [10 11 12])
=> false
```

SEE ALSO

[every?](#)

Returns true if coll is a collection and the predicate is true for all collection items, false otherwise.

[any?](#)

Returns true if the predicate is true for at least one collection item, false otherwise.

[not-any?](#)

Returns false if the predicate is true for at least one collection item, true otherwise

not-match?

```
(not-match? s regex)
```

Returns true if the string `s` does not match the regular expression `regex`.

The argument 'regex' may be a string representing a regular expression or a `java.util.regex.Pattern`.

See the functions in the 'regex' namespace if more than a simple regex match is required! E.g. `regex/matches-not?` performs much better on matching many strings against the same pattern:

```
(let [m (regex/matcher #"[0-9]+" "")]
  (filter #(regex/matches-not? m %) ["100" "1a1" "200"]))
```

```
(not-match? "S1000" "[0-9]+")
=> true
```

```
(not-match? "S1000" #"[0-9]+")
=> true
```

```
(not-match? "1000" "[0-9]+")
=> false
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/matches-not?](#)

Attempts to match the entire region against the pattern. Returns false if the patterns matches the string else true.

[regex/matches?](#)

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/find](#)

Returns the next regex match or nil if there is no further match. Returns nil if there is no match.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

not=

```
(not= x)
(not= x y)
(not= x y & more)
```

Same as `(not (= x y))`

```
(not= "abc" "abc")
=> false

(not= 0 0)
=> false

(not= 0 1)
=> true

(not= 0 0.0)
=> true

(not= 0 0.0M)
=> true

(not= "0" 0)
=> true

(not= 4)
=> false

(not= 1 2 3)
=> true
```

SEE ALSO

[=](#)

Returns true if both operands have equivalent type and value

[==](#)

Returns true if both operands have equivalent value.

[section](#)

[top](#)

ns

```
(ns sym)
```

Opens a namespace.

```
(do
  (ns xxx)
  (def foo 1)
  (ns yyy)
  (def foo 5)
  (println xxx/foo foo yyy/foo))
1 5 5
=> nil
```

SEE ALSO

[*ns*](#)

The current namespace

[ns?](#)

Returns true if n is an existing namespace that has been defined with (ns n) else false.

[ns-unmap](#)

Removes the mappings for the symbol from the namespace.

[ns-remove](#)

Removes the mappings for all symbols from the namespace.

[ns-list](#)

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

[ns-alias](#)

Add an alias in the current namespace to another namespace. Arguments are two symbols: the alias to be used, and the symbolic name ...

[ns-meta](#)

Returns the meta data of the namespace n or nil if n is not an existing namespace

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[var-ns](#)

Returns the namespace of the var's symbol.

[section](#)

[top](#)

ns-alias

```
(ns-alias alias namespace-sym)
```

Add an alias in the current namespace to another namespace. Arguments are two symbols: the alias to be used, and the symbolic name of the target namespace.

```
(ns-alias 'p 'parsatron)
=> nil
```

```
(do
  (load-module :hexdump)
  (ns-alias 'h 'hexdump)
  (h/dump [0 1 2 3]))
00000000: 0001 0203                ....
=> nil
```

SEE ALSO

[ns-unalias](#)

Removes a namespace alias in the current namespace.

[ns-aliases](#)

Returns a map of the aliases defined in the current namespace.

[*ns*](#)

The current namespace

[ns](#)

Opens a namespace.

[section](#)

[top](#)

ns-aliases

```
(ns-aliases)
```

Returns a map of the aliases defined in the current namespace.

```
(ns-aliases)
=> {}

(do
  (ns-alias 'h 'hexdump)
  (ns-alias 'p 'parsatron)
  (ns-aliases))
=> {h hexdump p parsatron}
```

SEE ALSO

[ns-alias](#)

Add an alias in the current namespace to another namespace. Arguments are two symbols: the alias to be used, and the symbolic name ...

[ns-unalias](#)

Removes a namespace alias in the current namespace.

[*ns*](#)

The current namespace

[ns](#)

Opens a namespace.

[section](#)

[top](#)

ns-list

```
(ns-list)
(ns-list ns)
```

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

```
(ns-list 'regex)
=> (regex/count regex/find regex/find+ regex/find-all regex/find-all+ regex/find? regex/group regex/groups regex
/matcher regex/matches regex/matches-not? regex/matches? regex/pattern regex/pattern? regex/reset)
```

```
(ns-list)
=> ("ansi" "app" "ascii-canvas" "ascii-charts" "ascii-table" "aviron" "aviron-cycler" "aviron-limiter" "aviron-
queue" "benchmark" "cargo" "cargo-arangodb" "cargo-postgresql" "cargo-qdrant" "chinook-postgresql" "cidr"
"component" "config" "cron" "crypt" "csv" "dag" "dec" "docker" "excel" "fonts" "geoip" "gradle" "gradlew"
"grep" "hexdump" "http-client-j8" "images" "inet" "installer" "io" "ipc" "java" "jdbc-core" "jdbc-postgresql"
"json" "jsonl" "jtokkit" "keystores" "kira" "loadpath" "logger" "math" "matrix" "maven" "mbean" "mimetypes"
"multipart" "openai" "openai-java" "parsifal" "pdf" "qrbill" "qrcode" "qrref" "regex" "repl" "ring" "ring-
multipart" "ring-mw" "ring-session" "ring-util" "rsa" "sandbox" "semver" "server-side-events" "sh" "shell"
"stopwatch" "str" "swiss-cities" "test" "time" "timing" "tomcat" "trace" "xchart" "xml" "zipvault")
```

SEE ALSO

[ns](#)

Opens a namespace.

[*ns*](#)

The current namespace

[ns-unmap](#)

Removes the mappings for the symbol from the namespace.

[ns-remove](#)

Removes the mappings for all symbols from the namespace.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

var-ns

Returns the namespace of the var's symbol.

[section](#)[top](#)

ns-meta

```
(ns-meta n)
```

Returns the meta data of the namespace `n` or `nil` if `n` is not an existing namespace

```
(do
  (ns foo)
  (ns-meta foo))
=> {}

(do
  (ns foo)
  (ns-meta 'foo))
=> {}

(do
  (ns foo)
  (def n 'foo)
  (ns-meta (var-get n)))
=> {}
```

SEE ALSO

[alter-ns-meta!](#)

Alters the metadata for a namespace. `f` must be free of side-effects.

[reset-ns-meta!](#)

Resets the metadata for a namespace

[ns](#)

Opens a namespace.

[section](#)[top](#)

ns-remove

```
(ns-remove ns)
```

Removes the mappings for all symbols from the namespace.

```
(do
  (ns foo)
  (def x 1)
  (ns bar)
  (def y 1)
  (ns-remove 'foo)
  (println "ns foo:" (ns-list 'foo))
  (println "ns bar:" (ns-list 'bar)))
```

```
ns foo: ()
ns bar: (bar/y)
=> nil
```

SEE ALSO

[ns](#)

Opens a namespace.

[ns-unmap](#)

Removes the mappings for the symbol from the namespace.

[ns-list](#)

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[var-ns](#)

Returns the namespace of the var's symbol.

[section](#)

[top](#)

ns-unalias

```
(ns-unalias alias)
```

Removes a namespace alias in the current namespace.

```
(do
  (ns-alias 'h 'hexdump)
  (ns-unalias 'h))
=> nil
```

SEE ALSO

[ns-alias](#)

Add an alias in the current namespace to another namespace. Arguments are two symbols: the alias to be used, and the symbolic name ...

[ns-aliases](#)

Returns a map of the aliases defined in the current namespace.

[*ns*](#)

The current namespace

[ns](#)

Opens a namespace.

[section](#)

[top](#)

ns-unmap

```
(ns-unmap ns sym)
```

Removes the mappings for the symbol from the namespace.

```
(do
  (ns foo)
  (def x 1))
```

```
(ns-unmap 'foo 'x)
(ns-unmap *ns* 'x))
=> nil
```

SEE ALSO

[ns](#)

Opens a namespace.

[*ns*](#)

The current namespace

[ns-remove](#)

Removes the mappings for all symbols from the namespace.

[ns-list](#)

Without arg lists the loaded namespaces, else lists all the symbols in the specified namespace ns.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[var-ns](#)

Returns the namespace of the var's symbol.

[section](#)

[top](#)

ns?

```
(ns? n)
```

Returns true if n is an existing namespace that has been defined with `(ns n)` else false.

```
(do
  (ns foo)
  (ns? foo))
=> true
```

SEE ALSO

[ns](#)

Opens a namespace.

[section](#)

[top](#)

nth

```
(nth coll idx)
(nth coll idx defaultVal)
```

Returns the nth element of coll.

Throws an exception if the index does not exist and there is no default value passed else returns the default value.

```
(nth nil 1)
=> nil
```

```
(nth [1 2 3] 1)
=> 2
```



```
(nth '(1 2 3) 1)
=> 2

(nth "abc" 2)
=> #\c

(nth nil 1 9)
=> nil

(nth [1 2 3] 6 9)
=> 9

(nth '(1 2 3) 6 9)
=> 9

(nth "abc" 6 9)
=> 9
```

[section](#)[top](#)

number?

```
(number? n)
```

Returns true if n is a number (int, long, double, or decimal)

```
(number? 4I))
=> true
```

```
(number? 4)
=> true
```

```
(number? 4.0M)
=> true
```

```
(number? 4.0)
=> true
```

```
(number? true)
=> false
```

```
(number? "a")
=> false
```

[section](#)[top](#)

object-array

```
(object-array coll)
(object-array len)
(object-array len init-val)
```

Returns an array of Java Objects containing the contents of coll or returns an array with the given length and optional init value

```
(object-array '(1 2 3 4 5))
=> [1, 2, 3, 4, 5]

(object-array '(1 2.0 3.45M "4" true))
=> [1, 2.0, 3.45, 4, true]

(object-array 10)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil]

(object-array 10 42)
=> [42, 42, 42, 42, 42, 42, 42, 42, 42]
```

[section](#) [top](#)

odd?

```
(odd? n)
```

Returns true if n is odd, throws an exception if n is not an integer

```
(odd? 3)
=> true
```

```
(odd? 4)
=> false
```

```
(odd? 4I)
=> false
```

SEE ALSO

[even?](#)

Returns true if n is even, throws an exception if n is not an integer

[section](#) [top](#)

offer!

```
(offer! queue v)
(offer! queue timeout v)
```

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait time if necessary for space to become available. For an indefinite timeout pass the timeout value `:indefinite`. If no timeout is given returns immediately false if the queue does not have any more capacity. Returns true if the element was added to this queue, else false.

```
(let [q (queue)]
  (offer! q 1)
  (offer! q 1000 2)
  (offer! q :indefinite 3)
  (offer! q 3)
  (poll! q)
  q)
=> (2 3 3)
```

SEE ALSO

[queue](#)

Creates a new mutable threadsafe bounded or unbounded queue.

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)[top](#)

offer-head!

```
(offer-head! deque v)
(offer-head! deque timeout v)
```

Offers an item to the head of a deque with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait time if necessary for space to become available. For an indefinite timeout pass the timeout value :indefinite. If no timeout is given returns immediately false if the queue does not have any more capacity. Returns true if the element was added to this queue, else false.

```
(let [q (deque)]
  (offer! q 1)
  (offer! q 1000 2)
  (offer! q :indefinite 3)
  (offer-head! q 0)
  (println (poll! q))
  (println q))
0
(1 2 3)
=> nil
```

SEE ALSO

[deque](#)

Creates a new mutable threadsafe bounded or unbounded deque.

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[put-head!](#)

Puts an item to the head of a deque. The operation is synchronous, it waits indefinitely until the value can be placed on the queue.

[take-tail!](#)

Retrieves and removes the tail value of the deque, waiting if necessary until a value becomes available.


```

(let [client (openai-java/client)
      registry (-> (openai-java/create-function-registry)
                    (openai-java/register-function "weather_get_current"
                                                    get-current-weather))

      chat (-> (openai-java/chat-completion client :GPT_5_4)
               (openai-java/add-function
                 "weather_get_current"
                 "Gets the current weather for a city."
                 "object"
                 { :location {
                     :type "string"
                     :description "A city, e.g.: Zurich" }
                   :unit {
                     :type "string"
                     :description "Temperature unit: celsius or fahrenheit" } }
                 [ "location" "unit" ])))]
  ;; ...
))

```

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[section](#)

[top](#)

openai-java/add-user-message

```
(add-user-message chat msg)
```

Adds a *user* message or a list of *user* messages to the chat completion request.

Returns the chat.

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[section](#)

[top](#)

openai-java/add-user-message-with-files

```
(add-user-message-with-files chat msg & file-objects)
```

Adds a *user* message with additional file objects to the chat completion request.

Returns the chat.

```

(do
  (load-module :openai-java)

  (let [client (openai-java/client)
        file-obj (openai-java/create-file-object
                  client
                  (io/file "/Users/foo/Desktop/Tour_Eiffel.pdf"))

```

```

:USER_DATA
3600)

chat      (-> (openai-java/chat-completion client :GPT_5_4)
            (openai-java/max-completion-tokens 2048)
            (openai-java/add-user-message-with-files
             "Describe this image"
             file-obj))
response (openai-java/execute chat)
elapsed  (openai-java/elapsed chat)
usage     (openai-java/usage response)
msg       (first (openai-java/messages response))]
(printf "Elapsed: %dms\n" elapsed)
(printf "Tokens:  %n%s\n" (openai-java/format-usage usage " "))
(printf "Result:  %n%s\n" msg))

```

SEE ALSO

[openai-java/create-file-object](#)

Creates a new OpenAI file object.

[openai-java/create-file-object-binary](#)

Creates a new OpenAI binary file object.

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[section](#)

[top](#)

openai-java/chat-completion

(chat-completion client model)

Defines a OpenAI chat completion request.

It serves as a OpenAI chat completion conversation.

Argument

client A client.

Use `openai-java/client` to create a client

model An OpenAI model. E.g.: `:GPT_5_4`, `:GPT_5_4_MINI`, `:GPT_5_4_NANO`

Returns the chat completion result

[OpenAI Text Prompting Guide](#) [OpenAI Text Cookbook](#)

```

(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        chat   (-> (openai-java/chat-completion client :GPT_5_4)
                    (openai-java/max-completion-tokens 2048)
                    (openai-java/add-user-message "Say this is a test"))
        response (openai-java/execute chat)
        msg      (first (openai-java/messages response))]
    (println msg)))

(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        chat   (-> (openai-java/chat-completion client :GPT_5_4)
                    (openai-java/max-completion-tokens 2048)

```

```

        (openai-java/add-user-message "Say this is a test"))
    response (openai-java/execute chat)
    elapsed (openai-java/elapsed chat)
    usage (openai-java/usage response)
    msg (first (openai-java/messages response))]
(printf "Elapsed: %dms\n\n" elapsed)
(printf "Tokens: %n%s\n" (openai-java/format-usage usage " "))
(printf "Result: %n%s\n" msg)))

;; conversation
(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        chat (-> (openai-java/chat-completion client :GPT_5_4)
                  (openai-java/max-completion-tokens 2048)
                  (openai-java/add-user-message "Say Hello!"))
        response (openai-java/execute chat)]
    (println (first (openai-java/messages response)))
    ;; 1st follow up question
    (openai-java/add-assistant-message chat (openai-java/messages response))
    (openai-java/add-user-message chat "Can you say it more informal?")
    (let [response (openai-java/execute chat)]
      (println (first (openai-java/messages response))))
    ;; 2nd follow up question
    (openai-java/add-assistant-message chat (openai-java/messages response))
    (openai-java/add-user-message chat "Can you say it very formal?")
    (let [response (openai-java/execute chat)]
      (println (first (openai-java/messages response))))))

;; structured output
(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        chat (-> (openai-java/chat-completion client :GPT_5_4)
                  (openai-java/max-completion-tokens 2048)
                  (openai-java/json-response-format
                    "employee-list"
                    "A list of employees"
                    "object"
                    { :employees { :items { "type" "string" }}} )
                  (openai-java/add-user-message "Who works at OpenAI?"))
        response (openai-java/execute chat)
        msg (first (openai-java/messages response))]
    (println (first (openai-java/messages response)))))

;; function call
(do
  (load-module :openai-java)

  (defn celsius-to-fahrenheit [c]
    (-> (double c) (* 9) (/ 5) (+ 32) (long))) ;; (c * 9) / 5 + 32

  (defn degrees [t unit]
    (if (str/equals-ignore-case? unit "celsius") t (celsius-to-fahrenheit t)))

  (defn get-current-weather [fnArgsJson]
    (let [args (json/read-str fnArgsJson)
          location (get args "location")
          unit (get args "unit")]
      (cond
        ;; we just support one hard coded location
        (str/contains? location "Zurich")
        (json/write-str { :location location
                          :unit unit})))

```

```

        :temperature (degrees 21 unit)
        :conditions "Mostly sunny" })

:else
  (json/write-str { :location location
                   :error      (str "No weather data available for " location "!")})))

(let [client (openai-java/client)
      registry (-> (openai-java/create-function-registry)
                   (openai-java/register-function "weather_get_current"
                                                    get-current-weather))
      chat (-> (openai-java/chat-completion client :GPT_5_4 registry)
              (openai-java/max-completion-tokens 2048)
              (openai-java/debug false)
              (openai-java/add-function
                "weather_get_current"
                "Gets the current weather for a city."
                { :location {
                  :type "string"
                  :description "A city, e.g.: Zurich" }
                  :unit {
                    :type "string"
                    :description "Temperature unit: celsius or fahrenheit" } }
                [ "location" "unit" ]])
            (openai-java/add-user-message "What is the weather in Zurich in Celsius?"))
      response (openai-java/execute chat)]
  (println (coalesce (first (openai-java/messages response)) "<no message>
"))
  ;; follow up question
  (openai-java/add-assistant-message chat (openai-java/messages response))
  (openai-java/add-user-message chat "Please give the current weather in Zurich and some ideas what to do in
Zurich on day like this!")
  (let [response (openai-java/execute chat)]
    (println (first (openai-java/messages response))))))

```

SEE ALSO

[openai-java/max-completion-tokens](#)

Sets the max completion tokens OpenAI must use. 'tokens' must be positive number.

[openai-java/json-response-format](#)

Sets a structured JSON response format

[openai-java/add-function](#)

Add a function definition to a chat. Any number of function definitions can be added to a chat.

[openai-java/add-user-message](#)

Adds a user message or a list of user messages to the chat completion request.

[openai-java/add-assistant-message](#)

Adds an assistant message or a list of assistant messages to the chat completion request.

[openai-java/execute](#)

Execute (send) a chat completion. Returns an OpenAI response.

[openai-java/client](#)

Creates an OpenAI client.

[openai-java/close](#)

Closes an OpenAI client.

[openai-java/format-usage](#)

Formats a response's usage info to human readable multiline text.

openai-java/client

(client & options)

Creates an OpenAI client.

Note:

Don't create more than one client in the same application. Each client has a connection pool and thread pools, which are more efficient to share between requests.

Parameter «options»

:openai-api-key k	An optional OpenAI API Key. As fallback the key is read from the environment variable "OPENAI_API_KEY".
:openai-org-id k	An optional OpenAI organisation ID. As fallback the id is read from the environment variable "OPENAI_ORG_ID".
:openai-proj-id k	An optional OpenAI project ID. As fallback the id is read from the environment variable "OPENAI_PROJECT_ID".
:openai-base-url u	An optional OpenAI baseURL. As fallback the url is read from the environment variable "OPENAI_BASE_URL". OpenAI uses the standard base URL 'https://api.openai.com/v1' if it is not explicitly defined.

Install OpenAI Java libraries in the REPL

```
(do
  (load-module :openai-java-install)
  (openai-java-install/install :dir (repl/lib-dir) :silent false))
```

Restart the REPL after installing the libraries!

```
(do
  (load-module :openai-java)
  (let [client (openai-java/client)]
    ;; ...
    (openai-java/close client)))

(do
  (load-module :openai-java)
  (let [client (openai-java/client :openai-api-key "sk-1234")]
    ;; ...
    (openai-java/close client)))
```

SEE ALSO

[openai-java/version](#)

Returns the OpenAI Java library version. E.g.: "4.32.0"

[openai-java/close](#)

Closes an OpenAI client.

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[openai-java/generate-image](#)

Generates an image.

[openai-java/create-speech](#)

Generates audio from the input text.

[section](#)

[top](#)

openai-java/close

```
(close client)
```

Closes an OpenAI client.

```
(do
  (load-module :openai-java)
  (let [client (openai-java/client)]
    ;; ...
    (openai-java/close client)))
```

SEE ALSO

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

openai-java/create-binary-file

```
(create-binary-file client data filename)
(create-binary-file client data filename purpose)
(create-binary-file client data filename expires-after-seconds)
```

Create a binary OpenAI file

Arguments:

client	An OpenAI client
data	A data <code>bytebuf</code>
filename	A filename. E.g.: "planning.pdf"
purpose	A purpose: <code>:USER_DATA</code> , <code>:VISION</code> , <code>:BATCH</code> , <code>:ASSISTANTS</code> , <code>:FINE_TUNE</code>
expires-after-seconds	The file object expiry time in seconds. A negative value means no expiration. Defaults to 3600s

Supported file types:

PDF files	<code>.pdf</code>
Text and code	<code>.txt</code> , <code>.md</code> , <code>.json</code> , <code>.html</code> , <code>.xml</code> , code files
Rich documents	<code>.doc</code> , <code>.docx</code> , <code>.rtf</code> , <code>.odt</code>
Presentations	<code>.ppt</code> , <code>.pptx</code>
Spreadsheets	<code>.csv</code> , <code>.xls</code> , <code>.xlsx</code>

Returns the file object.

```
(do
  (load-module :openai-java)

  (let [client (openai-java/client)
        data   (bytebuf-from-string "Hello")
        filename "demo.md"]
    (openai-java/create-binary-file client
                                   data filename
                                   :USER_DATA 3600)))
```

SEE ALSO

[openai-java/list-files](#)

List the files stored at OpenAI.

[openai-java/create-file](#)

Create an OpenAI file

[openai-java/retrieve-file](#)

Retrieve the info of file stored at OpenAI

[openai-java/delete-file](#)

Delete a file stored at OpenAI

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

openai-java/create-file

```
(create-file client file)
(create-file client file purpose)
(create-file client file expires-after-seconds)
```

Create an OpenAI file

Arguments:

client	An OpenAI client
data	A data <code>bytebuf</code>
filename	A filename. E.g.: "planning.pdf"
purpose	A purpose: <code>:USER_DATA</code> , <code>:VISION</code> , <code>:BATCH</code> , <code>:ASSISTANTS</code> , <code>:FINE_TUNE</code>
expires-after-seconds	The file object expiry time in seconds. A negative value means no expiration. Defaults to 3600s

Supported file types:

PDF files	<code>.pdf</code>
Text and code	<code>.txt</code> , <code>.md</code> , <code>.json</code> , <code>.html</code> , <code>.xml</code> , code files
Rich documents	<code>.doc</code> , <code>.docx</code> , <code>.rtf</code> , <code>.odt</code>
Presentations	<code>.ppt</code> , <code>.pptx</code>
Spreadsheets	<code>.csv</code> , <code>.xls</code> , <code>.xlsx</code>

Returns the file object.

```
(do
  (load-module :openai-java)

  (let [client (openai-java/client)
        file   (io/file "/Users/foo/Desktop/Tour_Eiffel.pdf")]
    (openai-java/create-file client file :USER_DATA 3600)))
```

SEE ALSO

[openai-java/list-files](#)

List the files stored at OpenAI.

[openai-java/create-binary-file](#)

Create a binary OpenAI file

[openai-java/retrieve-file](#)

Retrieve the info of file stored at OpenAI

[openai-java/delete-file](#)

Delete a file stored at OpenAI

openai-java/create-file-object

```
(create-file-object client file)
(create-file-object client file purpose)
(create-file-object client file expires-after-seconds)
```

Creates a new OpenAI file object.

Arguments:

client	An OpenAI client
file	A file path. E.g.: (io/file "/data/planning.pdf")
purpose	A purpose: <code>:USER_DATA</code> , <code>:VISION</code> , <code>:BATCH</code> , <code>:ASSISTANTS</code> , <code>:FINE_TUNE</code>
expires-after-seconds	The file object expiry time in seconds. A negative value means no expiration. Defaults to 3600s

Supported file types:

PDF files	<code>.pdf</code>
Text and code	<code>.txt</code> , <code>.md</code> , <code>.json</code> , <code>.html</code> , <code>.xml</code> , code files
Rich documents	<code>.doc</code> , <code>.docx</code> , <code>.rtf</code> , <code>.odt</code>
Presentations	<code>.ppt</code> , <code>.pptx</code>
Spreadsheets	<code>.csv</code> , <code>.xls</code> , <code>.xlsx</code>

Returns the file object `:com.openai.models.files.FileObject`.

SEE ALSO

[openai-java/create-file-object-binary](#)

Creates a new OpenAI binary file object.

[openai-java/add-user-message-with-files](#)

Adds a user message with additional file objects to the chat completion request.

[openai-java/client](#)

Creates an OpenAI client.

openai-java/create-file-object-binary

```
(create-file-object-binary client data filename)
(create-file-object-binary client data filename purpose)
(create-file-object-binary client data filename expires-after-seconds)
```

Creates a new OpenAI binary file object.

Arguments:

client	An OpenAI client
data	The data. A <code>bytebuf</code> or a Java <code>:InputStream</code>
filename	A filename. E.g.: "planning.pdf"

purpose A purpose: `:USER_DATA`, `:VISION`, `:BATCH`, `:ASSISTANTS`, `:FINE_TUNE`

expires-after-seconds The file object expiry time in seconds. A negative value means no expiration. Defaults to 3600s

Supported file types:

PDF files `.pdf`

Text and code `.txt`, `.md`, `.json`, `.html`, `.xml`, code files

Rich documents `.doc`, `.docx`, `.rtf`, `.odt`

Presentations `.ppt`, `.pptx`

Spreadsheets `.csv`, `.xls`, `.xlsx`

Returns the file object `:com.openai.models.files.FileObject`.

SEE ALSO

[openai-java/create-file-object](#)

Creates a new OpenAI file object.

[openai-java/add-user-message-with-files](#)

Adds a user message with additional file objects to the chat completion request.

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

openai-java/create-function-registry

(create-function-registry)

Create a function registry.

```
(do
  (load-module :openai-java)

  (defn get-weather [fnArgsJson] "sunny, 21°C")

  (let [registry (-> (openai-java/create-function-registry)
                    (openai-java/register-function "GetWeather"
                                                    get-weather))]
    ))
```

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[openai-java/register-function](#)

Register a function with the registry

[section](#)

[top](#)

openai-java/create-speech

(create-speech client model input & options)

Generates audio from the input text.

Parameters

:client c	A client. Use <code>openai-java/client</code> to create a client
:model m	The model to use for audio generation. One of <code>:TTS_1</code> , <code>:TTS_1_HD</code> , or a GPT image model (<code>:GPT_40_MINI_TTS</code>).
:input s	The text to generate audio for. The maximum length is 4096 characters.

Parameter «options»

:instructions s	Control the voice of your generated audio with additional instructions. Does not work with <code>:TTS_1</code> or <code>:TTS_1_HD</code> . max length: 4096 characters You can prompt the model to control aspects of speech, including: • <i>Accent</i> • <i>Emotional range</i> • <i>Intonation</i> • <i>Impressions</i> • <i>Speed of speech</i> • <i>Tone</i> • <i>Whispering</i>
:format f	The audio format. Supported formats are <code>:MP3</code> , <code>:OPIS</code> , <code>:AAC</code> , <code>:FLAC</code> , <code>:WAV</code> , <code>:PCM</code>
:voice v	The voice (string) to use when generating the audio. High quality voices: • "marin" (en_US, female) • "cedar" (en_US, male)

Returns an `:InputStream` with the created audio

```
(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        is      (openai-java/create-speech
                  client
                  :GPT_40_MINI_TTS
                  "Today is a wonderful day to build something people love!"
                  :format :MP3
                  :voice "cedar")])
  (->> (io/slurp-stream is :binary true)
        (io/spit (io/file "./audio.mp3"))))
(println "Audio saved to ./audio.mp3"))

;; MacOS only!!
(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        is      (openai-java/create-speech
                  client
                  :GPT_40_MINI_TTS
                  "Today is a wonderful day to build something people love!"
                  :format :MP3
                  :voice "cedar")])
  (let [audio-file (io/file "./audio.mp3")]
    (println "Downloading audio ...")
    (->> (io/slurp-stream is :binary true)
          (io/spit audio-file))
    (println "Saved audio to" audio-file)
    (if (os-type? :mac-osx)
        (do
          (println "Playing audio ...")
          (sh "afplay" (io/file-path audio-file))
```

```
nil)
(println "Playing audio supported on MacOS only!")))))
```

SEE ALSO

[openai-java/client](#)

Creates an OpenAI client.

section

top

openai-java/delete-file

```
(delete-file client id)
```

Delete a file stored at OpenAI

Returns `true` if the files could be deleted else `false`

```
(do
  (load-module :openai-java)

  (let [client (openai-java/client)]
    (openai-java/delete-file client "...file-id...")))
```

SEE ALSO

[openai-java/list-files](#)

List the files stored at OpenAI.

[openai-java/create-file](#)

Create an OpenAI file

[openai-java/create-binary-file](#)

Create a binary OpenAI file

[openai-java/retrieve-file](#)

Retrieve the info of file stored at OpenAI

[openai-java/client](#)

Creates an OpenAI client.

section

top

openai-java/execute

```
(execute chat)
```

Execute (send) a chat completion. Returns an OpenAI response.

Return the chat.

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

section

top

openai-java/format-usage

(format-usage usage)

Formats a response's usage info to human readable multiline text.

Usage token info:

```
{ :input-tokens 11
  :output-tokens 15
  :output-details { :reasoning-tokens 0 }
  :total-tokens 26 }
```

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[section](#) [top](#)

openai-java/generate-image

(generate-image client model prompt & options)

Generates an image.

Parameters

:client c A client.
 Use `openai-java/client` to create a client

:model m The model to use for image generation. One of `:DALL_E_2` , `:DALL_E_3` , or a GPT image model (`:GPT_IMAGE_1` , `:GPT_IMAGE_1_MINI` , `:GPT_IMAGE_1_5` , or `:GPT_IMAGE_2`). Defaults to `:DALL_E_2` unless a parameter specific to the GPT image models is used.

:prompt p A text description of the desired image(s). The maximum length is 32000 characters for the GPT image models, 1000 characters for dall-e-2 and 4000 characters for dall-e-3.

Parameter «options»

:background b Allows to set transparency for the background of the generated image(s). This parameter is only supported for GPT image models that support transparent backgrounds. Must be one of transparent, opaque, or auto (default value). When `:AUTO` is used, the model will automatically determine the best background for the image.
 `:GPT-IMAGE-2` does not support transparent backgrounds. Requests with background set to transparent will return an error for these models; use opaque or auto instead.
 If transparent, the output format needs to support transparency, so it should be set to either `:PNG` (default value) or `:WEBP` .

:number n The number of images to generate. Must be between 1 and 10. For `:DALL_E_3` , only `n=1` is supported.
 minimum: 1, maximum: 10

:compression c The compression level (0-100%) for the generated images. This parameter is only supported for the GPT image models with the `:WEBP` or `:JPEG` output formats, and defaults to 100.

:format f The format in which the generated images are returned. This parameter is only supported for the GPT image models. Must be one of `:PNG` , `:JPEG` , or `:WEBP` .

:quality f The quality of the image that will be generated.

- `:AUTO` (default value) will automatically select the best quality for the given model.
- `:HIGH` , `:MEDIUM` and `:LOW` are supported for the GPT image models.
- `:HD` and `:STANDARD` are supported for `:DALL_E_3` .
- `:STANDARD` is the only option for `:DALL_E_2` .

<code>:size f</code>	The size of the generated images. For <code>:GPT_IMAGE_2</code>, arbitrary resolutions are supported as WIDTHxHEIGHT strings, for example <code>1536x864</code>. Width and height must both be divisible by 16 and the requested aspect ratio must be between 1:3 and 3:1. Resolutions above <code>2560x1440</code> are experimental, and the maximum supported resolution is <code>3840x2160</code>. The requested size must also satisfy the model's current pixel and edge limits. The standard sizes <code>1024x1024</code>, <code>1536x1024</code>, and <code>1024x1536</code> are supported by the GPT image models; auto is supported for models that allow automatic sizing. For <code>:DALL_E_2</code>, use one of <code>256x256</code>, <code>512x512</code>, or <code>1024x1024</code>. For <code>:DALL_E_3</code>, use one of <code>1024x1024</code>, <code>1792x1024</code>, or <code>1024x1792</code>.
<code>:style f</code>	The style of the generated images. This parameter is only supported for <code>:DALL_E_3</code>. Must be one of <code>:VIVID</code> or <code>:NATURAL</code>. Vivid causes the model to lean towards generating hyper-real and dramatic images. Natural causes the model to produce more natural, less hyper-real looking images.
<code>:response-format f</code>	The format in which generated images with <code>:DALL_E_2</code> and <code>:DALL_E_3</code> are returned. Must be one of <code>:URL</code> or <code>:B64_JSON</code>. URLs are only valid for 60 minutes after the image has been generated. This parameter isn't supported for the GPT image models, which always return base64-encoded images.

Returns an OpenAI image response

[OpenAI Image Prompting Guide](#)

```
(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        response (openai-java/generate-image
                    client
                    :GPT_IMAGE_1_MINI
                    "A cat on bicycle"
                    :format :PNG
                    :number 1
                    :size "1024x1024")
        usage (openai-java/usage response)
        image-1 (first (openai-java/image-binaries response))]
    (printf "Tokens: %n%s%n" (openai-java/format-usage usage " "))
    (io/spit (io/file "./image-1.png") image-1)
    (println "Image #1 saved to ./image-1.png")))
```

SEE ALSO

[openai-java/client](#)

Creates an OpenAI client.

[openai-java/edit-image](#)

Edits an image.

[openai-java/usage](#)

Returns the token usage statistics.

[openai-java/image-binaries](#)

Returns a list of image binaries in the response.

[openai-java/image-urls](#)

Returns a list of image URLs in the response.

[section](#)

[top](#)

openai-java/image-binaries

```
(image-binaries response)
```

Returns a list of image binaries in the response.

SEE ALSO

[openai-java/generate-image](#)

Generates an image.

[openai-java/image-urls](#)

Returns a list of image URLs in the response.

[section](#)

[top](#)

openai-java/image-urls

(image-urls response)

Returns a list of image URLs in the response.

SEE ALSO

[openai-java/generate-image](#)

Generates an image.

[openai-java/image-binaries](#)

Returns a list of image binaries in the response.

[section](#)

[top](#)

openai-java/json-response-format

(json-response-format chat name description type properties)

Sets a structured JSON response format

Returns the chat.

```
(do
  (load-module :openai-java)

  (let [client (openai-java/client)
        chat  (-> (openai-java/chat-completion client :GPT_5_4)
                  (openai-java/json-response-format
                    "employee-list"
                    "A list of employees"
                    "object"
                    { :employees { :items { "type" "string" }}} )
                  (openai-java/add-user-message "Who works at OpenAI?"))]

    ;; ...
  ))
```

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[section](#)

[top](#)

openai-java/list-files

```
(list-files client)
(list-files client limit)
```

List the files stored at OpenAI.

Returns a list of file-object maps

```
(do
  (load-module :openai-java)

  (let [client (openai-java/client)]
    (->> (openai-java/list-files client)
      (docoll #(prn %))))))
```

SEE ALSO

[openai-java/create-file](#)

Create an OpenAI file

[openai-java/create-binary-file](#)

Create a binary OpenAI file

[openai-java/retrieve-file](#)

Retrieve the info of file stored at OpenAI

[openai-java/delete-file](#)

Delete a file stored at OpenAI

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

openai-java/max-completion-tokens

```
(max-completion-tokens chat tokens)
```

Sets the max completion tokens OpenAI must use. 'tokens' must be positive number.

Returns the chat.

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[section](#)

[top](#)

openai-java/messages

```
(messages response)
```

Returns a list of messages of the response.

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

openai-java/models

(models)

Returns all supported OpenAI models

```
(do
  (load-module :openai-java)
  (openai-java/models))
```

SEE ALSO

[openai-java/chat-completion](#)
Defines a OpenAI chat completion request.

openai-java/register-function

(registry name function)

Register a function with the registry

```
(do
  (load-module :openai-java)

  (defn get-weather [fnArgsJson] "sunny, 21°C")

  (let [registry (-> (openai-java/create-function-registry)
                     (openai-java/register-function "GetWeather"
                                                     get-weather))]

    ))
```

SEE ALSO

[openai-java/chat-completion](#)
Defines a OpenAI chat completion request.

[openai-java/create-function-registry](#)
Create a function registry.

openai-java/retrieve-file

(retrieve-file client id)

Retrieve the info of file stored at OpenAI

Returns a file-object map

```
(do
  (load-module :openai-java)

  (let [client (openai-java/client)]
    (openai-java/retrieve-file client "...file-id...")))
```

SEE ALSO

[openai-java/list-files](#)

List the files stored at OpenAI.

[openai-java/create-file](#)

Create an OpenAI file

[openai-java/create-binary-file](#)

Create a binary OpenAI file

[openai-java/delete-file](#)

Delete a file stored at OpenAI

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

openai-java/transcribe

```
(transcribe client model audio & options)
```

Transcribes audio data. The data data can be supplied as an `io/file`, a `bytebuf`, or a Java `InputStream`.

Returns a transcription object in json, diarized_json, or verbose_json format, or a stream of transcript events.

Parameters

:client c	A client. Use <code>openai-java/client</code> to create a client
:model m	The model to use for transcription. One of <code>:GTP_40_TRANSCRIBE</code> , <code>:GTP_40_MINI_TRANSCRIBE</code> , or a <code>:WHISPER_1</code> , or <code>:GTP_40_TRANSCRIBE_DIARIZE</code>
:audio a	The audio data to transcribe, in one of these formats: flac, mp3, mp4, mpeg, mpga, m4a, ogg, wav, or webm. The data data can be supplied as an <code>io/file</code> , a <code>bytebuf</code> , or a Java <code>InputStream</code> .

Parameter «options»

:filename f	If the passed audio is a <code>bytebuf</code> or a Java <code>InputStream</code> an explicit file name must be passed. E.g.: 'audio.wav' The filename's extensions must match the audio type <code>wav</code> , <code>mp3</code> , ...
:language l	The language of the input audio. Supplying the input language in ISO-639-1 (e.g. "en", "de", "fr", "it", ...) format will improve accuracy and latency.
:prompt p	An optional text to guide the model's style or continue a previous audio segment. The prompt should match the audio language. This field is not supported when using <code>:GTP_40_TRANSCRIBE_DIARIZE</code> .
:response-format f	The format of the output, in one of these options: <code>:JSON</code> , <code>:TEXT</code> , <code>:SRT</code> , <code>:VERBOSE_JSON</code> , <code>:VTT</code> , or <code>:DIARIZED_JSON</code> . For <code>:GTP_40_TRANSCRIBE</code> and <code>:GTP_40_MINI_TRANSCRIBE</code> , the only supported format is <code>:JSON</code> . For <code>:GTP_40_TRANSCRIBE_DIARIZE</code> , the supported formats are <code>:JSON</code> , <code>:TEXT</code> , and <code>:DIARIZED_JSON</code> , with <code>:DIARIZED_JSON</code> required to receive speaker annotations.
:temperature t	The sampling temperature, between 0.0 and 1.0. Higher values like 0.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic. If set to 0, the model will use log probability to automatically increase the temperature until certain thresholds are hit.

Returns the audio transcription response.

```

(do
  (load-module :openai-java)
  (let [demo-file (io/file "./audio.wav")
        client (openai-java/client)
        is (openai-java/create-speech
              client
              :GPT_40_MINI_TTS
              "Today is a wonderful day to build something people love!"
              :format :WAV
              :voice "cedar"))]
    (->> (io/slurp-stream is :binary true)
          (io/spit demo-file))
    (println "Audio saved to" demo-file)

    (println "Transcribing" demo-file)
    (let [response (openai-java/transcribe client
      :GPT_40_TRANSCRIBE
      demo-file
      :language "en"
      :temperature 0.1
      :response-format :TEXT)]
      (println "Transcription:")
      (println (openai-java/transcription-text response))))))

(do
  (load-module :openai-java)
  (let [client (openai-java/client)
        is (openai-java/create-speech
              client
              :GPT_40_MINI_TTS
              "Today is a wonderful day to build something people love!"
              :format :WAV
              :voice "cedar"))]

    (println "Transcribing..." )
    (let [response (openai-java/transcribe client
      :GPT_40_TRANSCRIBE
      is
      :filename "audio.wav"
      :language "en"
      :temperature 0.1
      :response-format :TEXT)]
      (println "Transcription:")
      (println (openai-java/transcription-text response))))))

```

SEE ALSO

[openai-java/transcription-text](#)

Returns the transcription text of a transcription response.

[openai-java/usage](#)

Returns the token usage statistics.

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

openai-java/transcription-text

```
(transcription-text response)
```

Returns the transcription text of a transcription response.

SEE ALSO

[openai-java/transcribe](#)

Transcribes audio data. The data data can be supplied as an io/file, a bytebuf, or a Java :InputStream.

[section](#)

[top](#)

openai-java/usage

(usage response)

Returns the token usage statistics.

E.g: { :input-tokens 11 :output-tokens 8 :output-tokens-detail { :reasoning-tokens 0 } :total-tokens 19 }

SEE ALSO

[openai-java/chat-completion](#)

Defines a OpenAI chat completion request.

[openai-java/generate-image](#)

Generates an image.

[section](#)

[top](#)

openai-java/version

(version)

Returns the OpenAI Java library version. E.g.: "4.32.0"

```
(do
  (load-module :openai-java)
  (openai-java/version))
```

SEE ALSO

[openai-java/client](#)

Creates an OpenAI client.

[section](#)

[top](#)

or

```
(or x)
(or x & next)
```

Ors the predicate forms

```
(or true false)
=> true
```

```
(or false false)
=> false
```

```
(or nil 100)
=> 100
```

```
(or)
=> false
```

SEE ALSO

[and](#)

Ands the predicate forms

[not](#)

Returns true if x is logical false, false otherwise.

[section](#)

[top](#)

or-timeout

```
(or-timeout p time time-unit)
```

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

```
(-> (promise (fn [] (sleep 100) "The quick brown fox"))
    (or-timeout 500 :milliseconds)
    (then-apply str/upper-case)
    (deref))
=> "THE QUICK BROWN FOX"
```

```
(-> (promise (fn [] (sleep 300) "The quick brown fox"))
    (or-timeout 200 :milliseconds)
    (then-apply str/upper-case)
    (deref))
=> TimeoutException: java.util.concurrent.TimeoutException
```

```
(-> (promise (fn [] (sleep 300) "The quick brown fox"))
    (then-apply str/upper-case)
    (or-timeout 200 :milliseconds)
    (deref))
=> TimeoutException: java.util.concurrent.TimeoutException
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[then-compose](#)

Composes the result of two promises. f receives the result of the first promise p and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise p with the same result or exception at this stage, that executes the action f. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

ordered-map

```
(ordered-map & keyvals)
(ordered-map map)
```

Creates a new ordered map containing the items.

```
(ordered-map :a 1 :b 2)
=> {:a 1 :b 2}
```

```
(ordered-map (hash-map :a 1 :b 2))
=> {:a 1 :b 2}
```

[section](#)[top](#)

ordered-map?

```
(ordered-map? obj)
```

Returns true if obj is an ordered map

```
(ordered-map? (ordered-map :a 1 :b 2))
=> true
```

[section](#)[top](#)

os-arch

```
(os-arch)
```

Returns the OS architecture. E.g: "x86_64"

```
(os-arch)
=> "aarch64"
```

SEE ALSO

[os-type](#)

Returns the OS type. Type is one of :windows, :mac-osx, :linux, :unix, or :unknown

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, :linux, or :unix

[os-name](#)

Returns the OS name. E.g.: "Mac OS X"

[os-version](#)

Returns the OS version

[section](#)

[top](#)

os-name

(os-name)

Returns the OS name. E.g.: "Mac OS X"

([os-name](#))

```
=> "Mac OS X"
```

SEE ALSO

[os-type](#)

Returns the OS type. Type is one of :windows, :mac-osx, :linux, :unix, or :unknown

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, :linux, or :unix

[os-arch](#)

Returns the OS architecture. E.g: "x86_64"

[os-version](#)

Returns the OS version

[section](#)

[top](#)

os-type

(os-type)

Returns the OS type. Type is one of `:windows`, `:mac-osx`, `:linux`, `:unix`, or `:unknown`

([os-type](#))

```
=> :mac-osx
```

SEE ALSO

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, :linux, or :unix

[os-arch](#)

Returns the OS architecture. E.g: "x86_64"

[os-name](#)

Returns the OS name. E.g.: "Mac OS X"

os-version

Returns the OS version

[section](#)

[top](#)

os-type?

```
(os-type? type)
```

Returns true if the OS id of the type otherwise false. Type is one of `:windows`, `:mac-osx`, `:linux`, or `:unix`

```
(os-type? :mac-osx)  
=> true
```

```
(os-type? :windows)  
=> false
```

SEE ALSO

[os-type](#)

Returns the OS type. Type is one of `:windows`, `:mac-osx`, `:linux`, `:unix`, or `:unknown`

[os-arch](#)

Returns the OS architecture. E.g: "x86_64"

[os-name](#)

Returns the OS name. E.g.: "Mac OS X"

[os-version](#)

Returns the OS version

[section](#)

[top](#)

os-version

```
(os-version)
```

Returns the OS version

```
(os-version)  
=> "26.5.1"
```

SEE ALSO

[os-type](#)

Returns the OS type. Type is one of `:windows`, `:mac-osx`, `:linux`, `:unix`, or `:unknown`

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of `:windows`, `:mac-osx`, `:linux`, or `:unix`

[os-arch](#)

Returns the OS architecture. E.g: "x86_64"

[os-name](#)

Returns the OS name. E.g.: "Mac OS X"

[section](#)

[top](#)

parse-version

```
(parse-version v)
```

Parses a version to its elements: major, minor, patch and suffix.

Returns a map with the version elements:

```
{ :major 1
  :minor 4
  :patch 34
  :suffix "alpha" }
```

```
(parse-version "1.23")
```

```
=> {:major 1 :minor 23 :patch 0 :suffix ""}
```

```
(parse-version "1.23.56")
```

```
=> {:major 1 :minor 23 :patch 56 :suffix ""}
```

```
(parse-version "1.23.56-alpha-01")
```

```
=> {:major 1 :minor 23 :patch 56 :suffix "alpha-01"}
```

```
(parse-version "1.23.56-snapshot")
```

```
=> {:major 1 :minor 23 :patch 56 :suffix "snapshot"}
```

SEE ALSO

[version](#)

Returns the Venice version.

[latest](#)

Returns the latest available Venice version or nil if the internet is not available.

[newer-version?](#)

Returns true if v1 is newer than v2 else false.

[section](#)

[top](#)

parsifal/>>

```
(>> p)
(>> p q)
(>> p q & ps)
```

Returns a new parser that parses a list of parsers. Returns the value of the last parser if all parsers succeed, else the parser fails.

Note: *Parsifal* is not implementing backtracking by default, and instead relies on the programmer to implement backtracking using constructs like `lookahead` and `attempt`.

The parser `>>` does not rewind the input state if any of the sub parsers fails. `>>*` is the backtracking version of `>>` that wraps the parsers within a call to `attempt`. See the backtracking example below.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/>> (p/char #\lparen) (p/digit) (p/char #\rparen)) "(1)")
  ; => #\

  ; Using bindings
```

```

(p/run (p/let->> [l (p/char #\lparen)
                  d (p/digit)
                  r (p/char #\rparen)]
          (p/always (str l d r)))
      "(1)")
; => "(1)"
)

; Backtracking demo
(do
  (load-module :parsifal ['parsifal :as 'p])

  ; No backtracking with `>>` parser
  (p/run (p/either (p/>> (p/letter) (p/digit))
            (p/letter))
          "abc")
; => ParseError: Unexpected token 'b' at line: 1 column: 2

; Backtracking with `>>*` parser
(p/run (p/either (p/>>* (p/letter) (p/digit))
            (p/letter))
        "abc")
; => #\a
)

```

[section](#)
[top](#)

parsifal/SourcePosition

Defines a protocol to add line and column information for custom tokens.

Definition:

```

(defprotocol SourcePosition
  (line [p])
  (column [p]))

(do
  (load-module :parsifal ['parsifal :as 'p])

  (deftype :Token [type :keyword, val :string, line :long, column :long]
    Object
    (toString [this] (str/format "[%s %s (%d,%d)]"
                                   (pr-str (:type this))
                                   (pr-str (:val this))
                                   (:line this)
                                   (:column this)))

    p/SourcePosition
    (line [this] (:line this))
    (column [this] (:column this)))

  (p/defparser lbracket []
    (p/let->> [[l c] (p/pos)
               t      (p/char #\[)]]
      (p/always (Token. :lbracket (str t) l c))))

  (p/run (lbracket) "[1,2,3]")
  ; => [:lbracket "[" (1,1)]
)

```

SEE ALSO

defprotocol

Defines a new protocol with the supplied function specs.

deftype

Defines a new custom record type for the name with the fields.

[section](#)[top](#)

parsifal/always

(always x)

A parser that always succeeds with the value given and consumes no input.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser integer []
    (p/let->> [t (p/many1 (p/digit))]
      (p/always (long (apply str t)))))

  (p/run (integer) "400")
  ; => 400
)

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser optional [p default-value]
    (p/either (p/attempt p)
      (p/always default-value)))

  (p/run (optional (p/char #\X) #\?) "X400")
  ; => #\X

  (p/run (optional (p/char #\X) #\?) "400")
  ; => #\?
)
```

[section](#)[top](#)

parsifal/any

(any)

Consume any single item from the head of the input. This parser will fail to consume if the input is empty.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/any) "Cats")
  ; => #\C

  (p/run (p/any) [#\C #\a #\t #\s])
  ; => #\C
)
```

parsifal/any-char

(any-char)

Consume any character.

Note: Works with char items only!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/any-char) "Cats")
  ; => #\C

  (p/run (p/any-char) [#\C #\a #\t #\s])
  ; => #\C
)
```

parsifal/any-char-of

(any-char-of s)

Consume any of the characters given in the string. E.g.: (any-char-of "([{") .

Note: Works with char items only!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/any-char-of "XYZ") "Hello, world!")
  ; => #\H
)
```

parsifal/attempt

(attempt p)

A parser that will attempt to parse `p`, and upon failure never consume any input.

Note: *Parsifal* is not implementing backtracking by default, and instead relies on the programmer to implement backtracking using constructs like `lookahead` and `attempt`.

The parsers `>>` and `let->>` do not rewind the input state if any of the sub parsers fails. To add backtracking parsers can be wrapped with `attempt`!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser optional [p default-value])
```

```

    (p/either (p/attempt p)
              (p/always default-value)))

(p/run (optional (p/char #\X) #\?) "400")
; => #\?
)

(do
  (load-module :parsifal ['parsifal :as 'p])

  ; Backtracking

  ; No implicit backtracking with `>>` parser!
  (p/run (p/either (p/>> (p/letter) (p/digit))
                  (p/letter))
          "abc")
; => ParseError: Unexpected token 'b' at line: 1 column: 2

; Explicit backtracking with `>>` parser using `attempt`!
(p/run (p/either (p/attempt (p/>> (p/letter) (p/digit)))
                (p/letter))
        "abc")
; => #\a
)

```

[section](#)
[top](#)

parsifal/between

(between open close p)

Returns a new parser that parses `open`, `p`, and `close` returning the value of `p` and discarding the values of `open` and `close`. Does not consume any input on failure.

```

(do
  (load-module :parsifal ['parsifal :as 'p])
  (p/run (p/between (p/char #\lparen)
                    (p/char #\rparen)
                    (p/many1 (p/digit)))
          "(123)")
; => [#\1 #\2 #\3]
)

```

[section](#)
[top](#)

parsifal/char

(char)

Consume the given character.

Note: Works with char items only!

```

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/char #\H) "Hello")
)

```



```
; => #\H

(p/run (p/char #\H) [#\H #\e #\l #\l #\o])
; => #\H
)
```

[section](#)[top](#)

parsifal/choice

```
(choice & p)
```

Returns a new parser that tries each given parsers in turn, returning the value of the first one that succeeds.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/choice (p/many1 (p/digit)) (p/many1 (p/letter)))) "Hello"
; => [#\H #\e #\l #\l #\o]

  (p/run (p/choice (p/many1 (p/digit)) (p/many1 (p/letter)))) "42"
; => [#\4 #\2]
)
```

[section](#)[top](#)

parsifal/defparser

```
(defparser name args & body)
```

The `defparser` macro defines `_functions_` that create parsers.

Note: *Parsifal* is not implementing backtracking by default, and instead relies on the programmer to implement backtracking using constructs like `lookahead` and `attempt`.

The parsers created by this macro do not rewind the input state if one of the sub parsers fails. To allow backtracking `attempt` can be used!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser sample []
    (p/string "Hello")
    (p/always 42))

  (p/run (sample) "Hello, world!")
; => 42
)

(do
  (load-module :parsifal ['parsifal :as 'p])

  ; Backtracking

  (p/defparser letter-and-digit []
    (p/letter)
    (p/digit))
)
```

```
; No implicit backtracking!  
(p/run (p/either (letter-and-digit) (p/letter)) "abc")  
; => ParseError: Unexpected token 'b' at line: 1 column: 2  
  
; Explicit backtracking with `attempt`!  
(p/run (p/either (p/attempt (letter-and-digit)) (p/letter)) "abc")  
; => #\a  
)
```

[section](#)[top](#)

parsifal/digit

(digit)

Consume a digit [0-9] character.

Note: Works with char items only!

```
(do  
  (load-module :parsifal ['parsifal :as 'p])  
  
  (p/run (p/digit) "123")  
  ; => #\1  
  
  (p/run (p/any-char) [#\1 #\2 #\3])  
  ; => #\1  
)
```

[section](#)[top](#)

parsifal/either

(either p q)

Returns a new parser that tries `p`, upon success, returning its value, and upon failure (if no input was consumed) tries to parse `q`

```
(do  
  (load-module :parsifal ['parsifal :as 'p])  
  
  (p/run (p/either (p/many1 (p/digit)) (p/many1 (p/letter))) "Hello")  
  ; => [#\H #\e #\l #\l #\o]  
  
  (p/run (p/either (p/many1 (p/digit)) (p/many1 (p/letter))) "42")  
  ; => [#\4 #\2]  
)
```

[section](#)[top](#)

parsifal/eof

(eof)
(eof err-msg)

A parser to detect the end of input. If there is nothing more to consume from the underlying input, this parser succeeds with a `nil` value, otherwise it fails.

A custom error message can be provided for the case the parser fails.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/eof) "")
  ; => nil

  (p/run (p/eof) "a")
  ; => ParseError: Expected end of input at line: 1 column: 1
)
```

[section](#)[top](#)

parsifal/hexdigit

(hexdigit)

Consume a hex digit [0-9a-fA-F] character.

Note: Works with char items only!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/hexdigit) "A00")
  ; => #\A

  (p/run (p/hexdigit) [#\A #\0 #\0])
  ; => #\A
)
```

[section](#)[top](#)

parsifal/let->>

(let->> [[& bindings_] & body])

Binds parser results to names for further processing input.

Note: *Parsifal* is not implementing backtracking by default, and instead relies on the programmer to implement backtracking using constructs like `lookahead` and `attempt`.

The parser `let->>` does not rewind the input state if one of the sub parsers fails. `let->>*` is the backtracking version of `let->>` that wraps the parsers within a call to `attempt`. See the backtracking example below.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser float []
    (p/let->> [i (p/many1 (p/digit))
              d (p/char #\.)
              f (p/many1 (p/digit))]
      (p/always (apply str (flatten (list i d f))))))
```

```

(p/run (float) "10.56")
; => "10.56"
)

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser int []
    (p/let->> [i (p/many1 (p/digit))]
      (let [n (long (apply str i))]
        (if (even? n)
          (p/always (str n " is even"))
          (p/always (str n " is odd"))))))))

  (p/run (int) "500")
  ; => "500 is even"
)

; Backtracking demo
(do
  (load-module :parsifal ['parsifal :as 'p])

  ; No backtracking with `let->>` parser!
  (p/run (p/either (p/let->> [c (p/letter)
                             d (p/digit)]
                        (p/always (list c d)))
          (p/letter))
    "abc")
  ; => ParseError: Unexpected token 'b' at line: 1 column: 2

  ; Backtracking with `let->>` parser
  (p/run (p/either (p/let->>* [c (p/letter)
                             d (p/digit)]
                        (p/always (list c d)))
          (p/letter))
    "abc")
  ; => #\a
)

```

[section](#)
[top](#)

parsifal/letter

```
(letter)
```

Consume a letter character defined by Java `Character.isLetter(ch)` .

Note: Works with char items only!

```

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/letter) "Cats")
  ; => #\C

  (p/run (p/letter) [#\C #\a #\t #\s])
  ; => #\C
)

```

[section](#)
[top](#)

parsifal/letter-or-digit

(letter-or-digit)

Consume a letter or digit character defined by Java `Character.isLetterOrDigit(ch)`.

Note: Works with char items only!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/letter-or-digit) "Cats")
  ; => #\C

  (p/run (p/letter-or-digit) "5Cats")
  ; => #\5

  (p/run (p/letter-or-digit) [#\C #\a #\t #\s])
  ; => #\C
)
```

[section](#)

[top](#)

parsifal/lineno

(lineno)

A parser that returns the current line number. It consumes no input.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser integer []
    (p/let->> [l (p/lineno)
               t (p/many1 (p/digit))]
      (p/always [:int (apply str t) l])))

  (p/run (integer) "400")
  ; => [:int "400" 1]
)
```

[section](#)

[top](#)

parsifal/lookahead

(lookahead p)

A parser that upon success consumes no input, but returns what was parsed.

Note: *Parsifal* is not implementing backtracking by default, and instead relies on the programmer to implement backtracking using constructs like `lookahead` and `attempt`.

```
(do
  (load-module :parsifal ['parsifal :as 'p])
```

```

(p/defparser block-string-tok []
  (p/between (p/times 3 (p/char #\quote))
    (p/times 3 (p/char #\quote))
    (p/many (p/let->> [cs (p/lookahead (p/times 3 (p/any-char)))]
      (if (= cs [#\quote #\quote #\quote])
        (p/never)
        (p/any-char))))))

(p/defparser block-string []
  (p/let->> [s (block-string-tok)]
    (p/always (apply str s))))

(p/run (block-string) "\"\"A \"string\" with quotes!\"\"")
; => "A \"string\" with quotes!"
)

```

[section](#)
[top](#)

parsifal/many

```
(many p)
```

Returns a new parser that will parse zero or more items that match the given parser `p`. The matched items are concatenated into a sequence.

Note: A `ParseError` will be thrown if this combinator is applied to a parser that accepts the empty string, as that would cause the parser to loop forever.

```

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/many (p/digit)) "1234-0000")
  ; => [#\1 #\2 #\3 #\4]

  (p/run (p/many (p/digit)) "ABC-12345")
  ; => []
)

```

[section](#)
[top](#)

parsifal/many1

```
(many1 p)
```

Returns a new parser that will parse one or more items that match the given parser `p`. The matched items are concatenated into a sequence.

Note: A `ParseError` will be thrown if this combinator is applied to a parser that accepts the empty string, as that would cause the parser to loop forever.

```

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/many1 (p/digit)) "1234-0000")
  ; => [#\1 #\2 #\3 #\4]

  (p/run (p/many1 (p/digit)) "ABC-12345")
  ; => ParseError: Unexpected token 'A' at line: 1 column: 1
)

```

parsifal/never

```
(never)
(never err-msg)
(never err-msg line column)
```

A parser that always fails, consuming no input.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  ;; parse a string with a single integer
  (p/defparser single-integer []
    (p/let->> [i (p/many1 (p/digit))
               t (p/either (p/eof) (p/any))])
    (if (nil? t)
        (p/always (apply str i))
        (p/never (str "Unexpected token '" t "'")))))

  (p/run (single-integer) "400")
  ; => "400"

  (p/run (single-integer) "400-")
  ; => ParseError: Unexpected token '-' at line: 1 column: 5
)
```

parsifal/none-char-of

```
(none-char-of s)
```

Consume all but of the characters given in the string. E.g.: `(none-char-of "({")` .

Note: Works with char items only!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/none-char-of "()[]{}") "Hello, world!")
  ; => #\H
)
```

parsifal/not-char

```
(not-char)
```

Consume all but the given character

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/not-char #\x) "Cats")
  ; => #\C

  (p/run (p/not-char #\x) [#\C #\a #\t #\s])
  ; => #\C
)
```

[section](#)
[top](#)

parsifal/pos

```
(pos)
```

A parser that returns the current line/column number as tuple of `[line col]`. It consumes no input.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/defparser integer []
    (p/let->> [[l c] (p/pos)
              t      (p/many1 (p/digit))]]
    (p/always [:int (apply str t) (list l c)])))

  (p/run (integer) "400")
  ; => [:int "400" (1,1)]
)
```

[section](#)
[top](#)

parsifal/run

```
(run p input)
```

Run a parser `p` over some input. The input can be a string or a seq of tokens, if the parser produces an error, its message is wrapped in a `ParseError` and thrown, and if the parser succeeds, its value is returned.

Parsifal is port of Nate Young's Clojure Parsatron [parser combinators](#) project.

Parsifal is not implementing backtracking by default, and instead relies on the programmer to implement backtracking using constructs like `lookahead` and `attempt`.

A simple parser example:

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/char #\H) "Hello")
  ; => #\H

  (p/run (p/char #\H) [#\H #\e #\l #\l #\o])
  ; => #\H
)
```

[section](#)
[top](#)

parsifal/string

(string s)

Consume the given string and returns a string. Does not consume any input upon failure.

Note: Works with char items only!

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/string "Hello") "Hello, world!")
  ; => "Hello"

  (p/run (p/string "Hello") (seq "Hello, world!"))
  ; => "Hello"
)

(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/either (p/string "Hello") (p/letter)) "Hello, world!")
  ; => "Hello"

  (p/run (p/either (p/string "Hello") (p/letter)) "Hello, world!")
  ; => #\H
)
```

[section](#)

[top](#)

parsifal/times

(times n p)

Returns a new parser that consumes exactly n times what the parser `p` matches. The matched items are concatenated into a sequence. Does not consume any input if not all of the repetitions match.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/times 5 (p/letter)) "Hello, world!")
  ; => [#\H #\e #\l #\l #\o]

  ;; Note: `p/times` is different from parsing letters explicitly
  (p/run (p/>> (p/letter)
              (p/letter)
              (p/letter)
              (p/letter)
              (p/letter))
          "Hello, world!")
  ; => [#\o]
)
```

[section](#)

[top](#)

parsifal/token

(token)

Consume a single item from the head of the input if `(consume? item)` predicate is not `nil`. This parser will fail to consume if either the `consume?` test returns `false` or if the input is empty.

```
(do
  (load-module :parsifal ['parsifal :as 'p])

  (p/run (p/token #(< % 5)) [3 5 7])
  ; => 3

  (p/run (p/token str/upper-case) "Hello")
  ; => #\H
)
```

[section](#)

[top](#)

partial

(partial f args*)

Takes a function `f` and fewer than the normal arguments to `f`, and returns a fn that takes a variable number of additional args. When called, the returned function calls `f` with `args` + additional args.

```
((partial * 2) 3)
=> 6

(map (partial * 2) [1 2 3 4])
=> (2 4 6 8)

(map (partial reduce +) [[1 2 3 4] [5 6 7 8]])
=> (10 26)

(do
  (def hundred-times (partial * 100))
  (hundred-times 5))
=> 500
```

[section](#)

[top](#)

partition

```
(partition n coll)
(partition n step coll)
(partition n step padcoll coll)
```

Returns a collection of lists of `n` items each, at offsets `step` apart. If `step` is not supplied, defaults to `n`, i.e. the partitions do not overlap. If a `padcoll` collection is supplied, use its elements as necessary to complete last partition upto `n` items. In case there are not enough padding elements, return a partition with less than `n` items. `padcoll` may be a lazy sequence

```

(partition 3 [0 1 2 3 4 5 6])
=> ([0 1 2] [3 4 5])

(partition 3 3 (repeat 99) [0 1 2 3 4 5 6])
=> ([0 1 2] [3 4 5] [6 99 99])

(partition 3 3 [] [0 1 2 3 4 5 6])
=> ([0 1 2] [3 4 5] [6])

(partition 2 3 [0 1 2 3 4 5 6])
=> ([0 1] [3 4])

(partition 3 1 [0 1 2 3 4 5 6])
=> ([0 1 2] [1 2 3] [2 3 4] [3 4 5] [4 5 6])

(partition 3 6 ["a"] (range 20))
=> ((0 1 2) (6 7 8) (12 13 14) (18 19 "a"))

(partition 4 6 ["a" "b" "c" "d"] (range 20))
=> ((0 1 2 3) (6 7 8 9) (12 13 14 15) (18 19 "a" "b"))

```

SEE ALSO

[partition-all](#)

Returns a collection of lists of n items each, at offsets step apart. If step is not supplied, defaults to n, i.e. the partitions do ...

[partition-by](#)

Applies f to each value in coll, splitting it each time f returns a new value.

[partition-at](#)

Applies the predicate f to each value in coll, splitting it each time f returns true.

[section](#)

[top](#)

partition-all

```

(partition-all n coll)
(partition-all n step coll)

```

Returns a collection of lists of n items each, at offsets step apart. If step is not supplied, defaults to n, i.e. the partitions do not overlap. May include partitions with fewer than n items at the end.

```

(partition-all 3 [0 1 2 3 4 5 6])
=> ([0 1 2] [3 4 5] [6])

(partition-all 2 3 [0 1 2 3 4 5 6])
=> ([0 1] [3 4] [6])

(partition-all 3 1 [0 1 2 3 4 5 6])
=> ([0 1 2] [1 2 3] [2 3 4] [3 4 5] [4 5 6] [5 6])

(partition-all 3 6 ["a"])
=> (["a"])

(partition-all 2 2 ["a" "b" "c" "d"])
=> (["a" "b"] ["c" "d"])

```

SEE ALSO

[partition](#)

Returns a collection of lists of n items each, at offsets step apart. If step is not supplied, defaults to n, i.e. the partitions do ...

[partition-by](#)

Applies f to each value in coll, splitting it each time f returns a new value.

[partition-at](#)

Applies the predicate f to each value in coll, splitting it each time f returns true.

[section](#)[top](#)

partition-at

```
(partition-at f coll)
```

Applies the predicate f to each value in coll, splitting it each time f returns true.

```
(partition-at even? [1 2 3 4 5 6])  
=> ((1) (2 3) (4 5) (6))
```

```
(partition-at even? [1 2 4 3 5 7])  
=> ((1) (2) (4 3 5 7))
```

```
(partition-at identity (seq "ABBA"))  
=> ((#\A #\B #\B #\A))
```

SEE ALSO

[partition-by](#)

Applies f to each value in coll, splitting it each time f returns a new value.

[partition-all](#)

Returns a collection of lists of n items each, at offsets step apart. If step is not supplied, defaults to n, i.e. the partitions do ...

[partition](#)

Returns a collection of lists of n items each, at offsets step apart. If step is not supplied, defaults to n, i.e. the partitions do ...

[section](#)[top](#)

partition-by

```
(partition-by f coll)
```

Applies f to each value in coll, splitting it each time f returns a new value.

```
(partition-by even? [1 2 3 4 5 6])  
=> ((1) (2) (3) (4) (5) (6))
```

```
(partition-by even? [1 2 4 3 5 7])  
=> ((1) (2 4) (3 5 7))
```

```
(partition-by identity (seq "ABBA"))  
=> ((#\A) (#\B #\B) (#\A))
```

```
(partition-by identity [1 1 1 1 2 2 3])  
=> ((1 1 1 1) (2 2) (3))
```

SEE ALSO

[partition-at](#)

Applies the predicate *f* to each value in *coll*, splitting it each time *f* returns true.

[partition](#)

Returns a collection of lists of *n* items each, at offsets *step* apart. If *step* is not supplied, defaults to *n*, i.e. the partitions do ...

[partition-all](#)

Returns a collection of lists of *n* items each, at offsets *step* apart. If *step* is not supplied, defaults to *n*, i.e. the partitions do ...

[section](#)[top](#)

pcalls

```
(pcalls & fns)
```

Executes the no-arg *fns* in parallel, returning a sequence of their values in the same order the functions are passed. In contrast, side effects of *fns* (if any) are coming in random order!

`pcalls` is implemented using Venice futures and processes `(+ 2 (cpus))` functions in parallel.

```
(pcalls #( + 1 2) #( + 2 3) #( + 3 4))  
=> (3 5 7)
```

SEE ALSO

[pmap](#)

Like *map*, except *f* is applied in parallel. Only useful for computationally intensive functions where the time of *f* dominates the coordination ...

[preduce](#)

Reduces a collection using a parallel reduce-combine strategy. The collection is partitioned into groups of approximately *n* items, ...

[cpus](#)

Returns the number of available processors or number of hyperthreads if the CPU supports hyperthreads.

[section](#)[top](#)

pdf/available?

```
(pdf/available?)
```

Checks if the 3rd party libraries required for generating PDFs are available.

```
(pdf/available?)
```

[section](#)[top](#)

pdf/check-required-libs

```
(pdf/check-required-libs)
```

Checks if the 3rd party libraries required for generating PDFs are available. Throws an exception if not.

(pdf/check-required-libs)

section

top

pdf/copy

(pdf/copy pdf & page-nr)

Copies pages from a PDF to a new PDF. The PDF is passed as bytebuf. Returns the new PDF as a bytebuf.

```
; copy the first and second page
(pdf/copy pdf :1 :2)

; copy the last and second last page
(pdf/copy pdf :-1 :-2)

; copy the pages 1, 2, 6-10, and 12
(pdf/copy pdf :1 :2 :6-10 :12)
```

SEE ALSO

[pdf/merge](#)

Merge multiple PDFs into a single PDF. The PDFs are passed as bytebuf. Returns the new PDF as a bytebuf.

[pdf/pages](#)

Returns the number of pages of a PDF. The PDF is passed as bytebuf.

[pdf/watermark](#)

Adds a watermark text to the pages of a PDF. The passed PDF pdf is a bytebuf. Returns the new PDF as a bytebuf.

section

top

pdf/extract-urls

(pdf/extract-urls pdf)

Extracts the URLs from a PDF.

pdf may be a:

- string file path, e.g: "/temp/foo.pdf"
- bytebuffer
- `java.io.File`, e.g: `(io/file "/temp/foo.pdf")`
- `java.io.InputStream`

Returns a list of URLs given as maps with the keys: `:url`, `:url-text`, and `:page-num`

```
(do
  (def xhtml ""
    <?xml version="1.0" encoding="UTF-8"?>
    <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
      <body>
        <a href="https://github.com/">GitHub</a>
        <a href="https://duckduckgo.com/">DuckDuckGo</a>
```

```
</body>
</html>
""")
```

```
(pdf/extract-urls (pdf/render xhtml))
```

SEE ALSO

[pdf/render](#)

Renders a PDF.

[section](#)

[top](#)

pdf/merge

```
(pdf/merge pdfs)
```

Merge multiple PDFs into a single PDF. The PDFs are passed as bytebuf. Returns the new PDF as a bytebuf.

```
(pdf/merge pdf1 pdf2)
```

```
(pdf/merge pdf1 pdf2 pdf3)
```

SEE ALSO

[pdf/copy](#)

Copies pages from a PDF to a new PDF. The PDF is passed as bytebuf. Returns the new PDF as a bytebuf.

[pdf/pages](#)

Returns the number of pages of a PDF. The PDF is passed as bytebuf.

[pdf/watermark](#)

Adds a watermark text to the pages of a PDF. The passed PDF pdf is a bytebuf. Returns the new PDF as a bytebuf.

[section](#)

[top](#)

pdf/page-count

```
(pdf/page-count pdf)
```

Returns the number of pages in a PDF.

Uses the pdfbox libraries.

SEE ALSO

[pdf/render](#)

Renders a PDF.

[pdf/to-text](#)

Extracts the text from a PDF.

[section](#)

[top](#)

pdf/page-to-image

```
(pdf/page-to-image pdf page-nr)
(pdf/page-to-image pdf page-nr dpi)
```

Converts a page from the PDF to an image buffer.

The passed PDF pdf is a bytebuf. Returns the image buffer as `ajava.awt.image.BufferedImage` that can be further processed or saved with the `:images` module.

SEE ALSO

[pdf/render](#)

Renders a PDF.

[pdf/to-text](#)

Extracts the text from a PDF.

[section](#)

[top](#)

pdf/pages

```
(pdf/pages pdf)
```

Returns the number of pages of a PDF. The PDF is passed as bytebuf.

Uses the openPDF libraries.

```
(->> (str/lorem-ipsum :paragraphs 30)
      (pdf/text-to-pdf)
      (pdf/pages))
=> 3
```

SEE ALSO

[pdf/merge](#)

Merge multiple PDFs into a single PDF. The PDFs are passed as bytebuf. Returns the new PDF as a bytebuf.

[pdf/copy](#)

Copies pages from a PDF to a new PDF. The PDF is passed as bytebuf. Returns the new PDF as a bytebuf.

[pdf/watermark](#)

Adds a watermark text to the pages of a PDF. The passed PDF pdf is a bytebuf. Returns the new PDF as a bytebuf.

[section](#)

[top](#)

pdf/render

```
(pdf/render xhtml & options)
```

Renders a PDF.

Options:

`:base-url url` a base url for resources . E.g.: "classpath:/"

`:resources resmap` a resource map for dynamic resources

```
(do
  (load-module :kira))
```



```

(defn format-ts [t] (time/format t "yyyy-MM-dd"))

;; define the template
(def template (str/strip-indent
  ""<?xml version="1.0" encoding="UTF-8"?>
  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
    <body>
      <div>${ (kira/escape-xml title) }</div>
      <div>${ (kira/escape-xml ts test/format-ts) }</div>
    </body>
  </html>
  """))

(def data { :title "Hello, world"
            :ts (time/local-date 2000 8 1) })

(def xhtml (kira/eval template ["${" "}$" data]))

(pdf/render xhtml)

(pdf/render xhtml :base-url "classpath:/")

(pdf/render xhtml
  :base-url "classpath/"
  :resources {"/chart_1.png" (chart-create :2018)
              "/chart_2.png" (chart-create :2019) })

```

SEE ALSO

[pdf/text-to-pdf](#)

Creates a PDF from simple text. The tool process line-feeds '\n' and form-feeds. To start a new page just insert a form-feed marker ...

[section](#)
[top](#)

pdf/text-to-pdf

```
(pdf/text-to-pdf text & options)
```

Creates a PDF from simple text. The tool process line-feeds '\n' and form-feeds. To start a new page just insert a form-feed marker "<form-feed>".

Options:

:font-size n	font size in pt (double), defaults to 9.0
:font-weight n	font weight (0...1000) (long), defaults to 200
:font-monospace b	if true use monospaced font, defaults to false

```

(->> (pdf/text-to-pdf "Lorem Ipsum...")
      (io/spit "text.pdf"))

```

SEE ALSO

[pdf/render](#)

Renders a PDF.

[pdf/to-text](#)

Extracts the text from a PDF.

pdf/to-text

```
(pdf/to-text pdf)
```

Extracts the text from a PDF.

pdf may be a:

- string file path, e.g: `"/temp/foo.pdf"`
- `bytebuffer`
- `java.io.File`, e.g: `(io/file "/temp/foo.pdf")`
- `java.io.InputStream`

```
(-> (pdf/text-to-pdf "Lorem Ipsum...")  
    (pdf/to-text)  
    (println))
```

SEE ALSO

[pdf/text-to-pdf](#)

Creates a PDF from simple text. The tool process line-feeds '\n' and form-feeds. To start a new page just insert a form-feed marker ...

[pdf/render](#)

Renders a PDF.

pdf/watermark

```
(pdf/watermark pdf options-map)  
(pdf/watermark pdf & options)
```

Adds a watermark text to the pages of a PDF. The passed PDF pdf is a `bytebuf`. Returns the new PDF as a `bytebuf`.

Options:

:text s	watermark text (string), defaults to "WATERMARK"
:font-size n	font size in pt (double), defaults to 24.0
:font-char-spacing n	font character spacing (double), defaults to 0.0
:color s	font color (HTML color string), defaults to #000000
:opacity n	opacity 0.0 ... 1.0 (double), defaults to 0.4
:outline-color s	font outline color (HTML color string), defaults to #000000
:outline-opacity n	outline opacity 0.0 ... 1.0 (double), defaults to 0.8
:outline-width n	outline width 0.0 ... 10.0 (double), defaults to 0.5
:angle n	angle 0.0 ... 360.0 (double), defaults to 45.0
:over-content b	print text over the content (boolean), defaults to true
:skip-top-pages n	the number of top pages to skip (long), defaults to 0
:skip-bottom-pages n	the number of bottom pages to skip (long), defaults to 0

```
(pdf/watermark pdf :text "CONFIDENTIAL" :font-size 64 :font-char-spacing 10.0)
```

```
(let [watermark { :text "CONFIDENTIAL"
                  :font-size 64
                  :font-char-spacing 10.0 } ]
      (pdf/watermark pdf watermark))
```

SEE ALSO

[pdf/merge](#)

Merge multiple PDFs into a single PDF. The PDFs are passed as bytebuf. Returns the new PDF as a bytebuf.

[pdf/copy](#)

Copies pages from a PDF to a new PDF. The PDF is passed as bytebuf. Returns the new PDF as a bytebuf.

[pdf/pages](#)

Returns the number of pages of a PDF. The PDF is passed as bytebuf.

[section](#)[top](#)

peek

```
(peek coll)
```

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the head element (or nil if the queue is empty).

```
(peek '(1 2 3 4))
=> 1
```

```
(peek [1 2 3 4])
=> 4
```

```
(let [s (conj! (stack) 1 2 3 4)]
      (peek s))
=> 4
```

```
(let [q (conj! (queue) 1 2 3 4)]
      (peek q))
=> 1
```

[section](#)[top](#)

perf

```
(perf expr warmup-iterations test-iterations)
```

Performance test with the given expression.

Runs the test in 3 phases:

1. Runs the expr in a warmup phase to allow the HotSpot compiler to do optimizations.
2. Runs the garbage collector.
3. Runs the expression under profiling. Returns nil.

After a test run metrics data can be obtained with (prof :data-formatted)

```
(do
  (perf (+ 120 200) 12000 1000)
  (println (prof :data-formatted)))
```

SEE ALSO

[time](#)

Evaluates expr and prints the time it took. Returns the value of expr.

[prof](#)

Controls the code profiling. See the companion functions/macros 'dorun' and 'perf'. The perf macro is built on prof and dorun and provides ...

[section](#)[top](#)

pid

```
(pid)
```

Returns the PID of this process.

```
(pid)
=> "8408"
```

[section](#)[top](#)

pmap

```
(pmap f coll)
(pmap f coll & colls)
```

Like `map`, except *f* is applied in parallel. Only useful for computationally intensive functions where the time of *f* dominates the coordination overhead.

The result collection is sorted in the same way as for `map`, i.e. it preserves the items' order in the *coll* (or *colls*) parameter(s) of `pmap`. In other words: calculation is done parallel, but the result is delivered in the order the input came (in *coll/colls*). In contrast, side effects of *f* (if any) are coming in random order!

`pmap` is implemented using Venice futures and processes `(+ 2 (cpus))` items in parallel.

;; With `pmap`, the total elapsed time is just over 2 seconds:

```
(do
  (defn long-running-job [n]
    (sleep 2000) ; wait for 2 seconds
    (+ n 10))
  (time (pmap long-running-job (range 4))))
Elapsed time: 2.01s
=> (10 11 12 13)
```

;; With `map`, the total elapsed time is roughly 4 * 2 seconds:

```
(do
  (defn long-running-job [n]
    (sleep 2000) ; wait for 2 seconds
    (+ n 10))
  (time (map long-running-job (range 4))))
Elapsed time: 8.01s
=> (10 11 12 13)
```

SEE ALSO

[pcalls](#)

Executes the no-arg fns in parallel, returning a sequence of their values in the same order the functions are passed. In contrast, ...

[preduce](#)

Reduces a collection using a parallel reduce-combine strategy. The collection is partitioned into groups of approximately n items, ...

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[cpus](#)

Returns the number of available processors or number of hyperthreads if the CPU supports hyperthreads.

[section](#)[top](#)

poll!

```
(poll! queue)
(poll! queue timeout)
```

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite. If no timeout is given returns the item if one is available else returns nil. With a timeout returns the item if one is available within the given timeout else returns nil.

```
(let [q (conj! (queue) 1 2 3 4)]
  (poll! q)
  (poll! q 1000)
  q)
=> (3 4)
```

SEE ALSO

[queue](#)

Creates a new mutable threadsafe bounded or unbounded queue.

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)[top](#)

poll-tail!

```
(poll-tail! queue)
(poll-tail! queue timeout)
```

Polls an item from the tail of a deque with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value `:indefinite`. If no timeout is given returns the item if one is available else returns `nil`. With a timeout returns the item if one is available within the given timeout else returns `nil`.

```
(let [q (conj! (deque) 1 2 3 4)]
  (println q)
  (println (poll! q))
  (println (poll-tail! q 300))
  (println (poll! q))
  (println q))
(1 2 3 4)
1
4
2
(3)
=> nil
```

SEE ALSO

[deque](#)

Creates a new mutable threadsafe bounded or unbounded deque.

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always `nil`.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[put-head!](#)

Puts an item to the head of a deque. The operation is synchronous, it waits indefinitely until the value can be placed on the queue.

[take-tail!](#)

Retrieves and removes the tail value of the deque, waiting if necessary until a value becomes available.

[offer-head!](#)

Offers an item to the head of a deque with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[peek](#)

For a list, same as `first`, for a vector, same as `last`, for a stack the top element (or `nil` if the stack is empty), for a queue the ...

[empty?](#)

Returns true if `x` is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (`count nil`) returns 0. Also works on strings, and Java Collections

[section](#)

[top](#)

pop

```
(pop coll)
```

For a list, returns a new list without the first item, for a vector, returns a new vector without the last item.

```
(pop '(1 2 3 4))
=> (2 3 4)
```

```
(pop [1 2 3 4])
=> [1 2 3]
```

pop!

```
(pop! stack)
```

Pops an item from a stack.

```
(let [s (stack)]
  (push! s 1)
  (push! s 2)
  (push! s 3)
  (pop! s))
=> 3
```

SEE ALSO

[stack](#)

Creates a new mutable threadsafe stack.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[push!](#)

Pushes an item to a stack.

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

pos?

```
(pos? x)
```

Returns true if x greater than zero else false

```
(pos? 3)
=> true
```

```
(pos? -3)
=> false
```

```
(pos? 3I)
=> true
```

```
(pos? 3.2F)
=> true
```

```
(pos? 3.2)
=> true
```

```
(pos? 3.2M)
=> true
```

SEE ALSO

[zero?](#)

Returns true if x zero else false

[neg?](#)

Returns true if x smaller than zero else false

[section](#)

[top](#)

postwalk

```
(postwalk f form)
```

Performs a depth-first, post-order traversal of form. Calls f on each sub-form, uses f's return value in place of the original.

```
(postwalk (fn [x] (println "Walked:" (pr-str x)) x)
          '(1 2 {:a 1 :b 2}))
Walked: 1
Walked: 2
Walked: :a
Walked: 1
Walked: [:a 1]
Walked: :b
Walked: 2
Walked: [:b 2]
Walked: {:a 1 :b 2}
Walked: (1 2 {:a 1 :b 2})
=> (1 2 {:a 1 :b 2})
```

SEE ALSO

[prewalk](#)

Performs a depth-last, pre-order traversal of form. Calls f on each sub-form, uses f's return value in place of the original.

[section](#)

[top](#)

postwalk-replace

```
(postwalk-replace smap form)
```

Recursively transforms form by replacing keys in smap with their values. Like `replace` but works on any data structure. Does replacement at the leaves of the tree first.

`postwalk-replace` is the equivalent of *Common Lisp's* `sublis` function.

```
(postwalk-replace {:a 1 :b 2} [:a :b])
=> [1 2]
```

```
(postwalk-replace {:a 1 :b 2} [:a :b :c])
=> [1 2 :c]
```



```
(postwalk-replace {:a 1 :b 2} [:a :b [:a :b] :c])
=> [1 2 [1 2] :c]
```

```
(postwalk-replace {'x 1 'y 2} '(+ x y))
=> (+ 1 2)
```

SEE ALSO

[prewalk-replace](#)

Recursively transforms form by replacing keys in smap with their values. Like `replace` but works on any data structure. Does replacement ...

[postwalk](#)

Performs a depth-first, post-order traversal of form. Calls `f` on each sub-form, uses `f`'s return value in place of the original.

[section](#)

[top](#)

pow

```
(pow x y)
```

Returns the value of `x` raised to the power of `y`

```
(pow 10 2)
=> 100.0
```

```
(pow 10.23 2)
=> 104.6529
```

```
(pow 10.23 2.5)
=> 334.7257199023319
```

[section](#)

[top](#)

pr

```
(pr & xs)
(pr os & xs)
```

Prints the values `xs` to the output stream that is the current value of `*out*` or to the passed output stream `os` if given. The passed stream must be a subclass of either `:java.io.PrintStream` or `:java.io.Writer`.

Prints the values, separated by spaces if there is more than one. `pr` and `prn` print in a way that objects can be read by the reader.

Returns `nil`.

```
(pr "hello")
"hello"
=> nil
```

```
(pr {:foo "hello" :bar 34.5})
{:foo "hello" :bar 34.5}
=> nil
```

```
(pr ['a :b "\n" #\space "c"])
[a :b "\n" #\space "c"]
=> nil
```

```
(pr *out* [10 20 30])
[10 20 30]
=> nil
```

```
(pr *err* [10 20 30])
[10 20 30]
=> nil
```

SEE ALSO

[prn](#)

Prints the values xs to the output stream that is the current value of *out* or to the passed stream os if given followed by a (newline).

[newline](#)

Without arg writes a platform-specific newline to the output channel that is the current value of *out*. With arg writes a newline ...

[pr-str](#)

With no args, returns the empty string. With one arg x, returns x.toString(). With more than one arg, returns the concatenation of ...

[section](#)[top](#)

pr-str

```
(pr-str & xs)
```

With no args, returns the empty string. With one arg x, returns x.toString(). With more than one arg, returns the concatenation of the str values of the args with delimiter ' '.

```
(pr-str)
=> ""
```

```
(pr-str 1 2 3)
=> "1 2 3"
```

```
(pr-str 1I)
=> "1I"
```

```
(pr-str 3.1415927M)
=> "3.1415927M"
```

```
(pr-str +)
=> "+"
```

```
(pr-str [1 2 3])
=> "[1 2 3]"
```

```
(pr-str "total " 100)
=> "\"total \" 100"
```

```
(pr-str #\h #\i)
=> "#\h #\i"
```

SEE ALSO

[str](#)

With no args, returns the empty string. With one arg x, returns x.toString(). (str nil) returns the empty string. With more than one ...

preduce

```
(preduce n combine-fn combine-seed reduce-fn reduce-seed coll)


```

Reduces a collection using a parallel reduce-combine strategy. The collection is partitioned into groups of approximately `n` items, each of which is reduced with `reduce-fn` (with `reduce-seed` as its seed value) in parallel. The results of these reductions are then reduced with the `combine-fn` (with `combine-seed` as its seed value). Without an explicit `combine-fn` the `reduce-fn` and its seed `reduce-seed` will be used as `combine-fn` and `combine-seed`.

```
(preduce 3 + 0 + 0 [1 2 3 4 5])
=> 15
```

```
(preduce 3 (fn [acc x] (+ acc x)) 0 (fn [acc x] (+ acc x)) 0 [1 2 3 4 5])
=> 15
```

```
(preduce 3 + 0 [1 2 3 4 5])
=> 15
```

```
(preduce 3 (fn [acc x] (+ acc x)) 0 [1 2 3 4 5])
=> 15
```

SEE ALSO

[reduce](#)

`f` should be a function of 2 arguments. If `val` is not supplied, returns the result of applying `f` to the first 2 items in `coll`, then ...

[map](#)

Applies `f` to the set of first items of each `coll`, followed by applying `f` to the set of second items in each `coll`, until any one of the ...

[filter](#)

Returns a collection of the items in `coll` for which `(predicate item)` returns logical true.

[pmap](#)

Like `map`, except `f` is applied in parallel. Only useful for computationally intensive functions where the time of `f` dominates the coordination ...

[pcalls](#)

Executes the no-arg fns in parallel, returning a sequence of their values in the same order the functions are passed. In contrast, ...

prewalk

```
(prewalk f form)
```

Performs a depth-last, pre-order traversal of `form`. Calls `f` on each sub-form, uses `f`'s return value in place of the original.

```
(prewalk (fn [x] (println "Walked:" (pr-str x)) x)
 '(1 2 {:a 1 :b 2}))
Walked: (1 2 {:a 1 :b 2})
Walked: 1
Walked: 2
Walked: {:a 1 :b 2}
Walked: [:a 1]
Walked: :a
Walked: 1
```

```
Walked: [:b 2]
Walked: :b
Walked: 2
=> (1 2 {:a 1 :b 2})
```

SEE ALSO

[postwalk](#)

Performs a depth-first, post-order traversal of form. Calls `f` on each sub-form, uses `f`'s return value in place of the original.

[section](#)[top](#)

prewalk-replace

```
(prewalk-replace smap form)
```

Recursively transforms `form` by replacing keys in `smap` with their values. Like `replace` but works on any data structure. Does replacement at the root of the tree first.

```
(prewalk-replace {:a 1 :b 2} [:a :b])
=> [1 2]
```

```
(prewalk-replace {:a 1 :b 2} [:a :b :c])
=> [1 2 :c]
```

```
(prewalk-replace {:a 1 :b 2} [:a :b [:a :b] :c])
=> [1 2 [1 2] :c]
```

```
(prewalk-replace {'x 1 'y 2} '(+ x y))
=> (+ 1 2)
```

SEE ALSO

[postwalk-replace](#)

Recursively transforms `form` by replacing keys in `smap` with their values. Like `replace` but works on any data structure. Does replacement ...

[prewalk](#)

Performs a depth-last, pre-order traversal of form. Calls `f` on each sub-form, uses `f`'s return value in place of the original.

[section](#)[top](#)

print

```
(print & xs)
(print os & xs)
```

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed stream `os` that must be a subclass of either `:java.io.PrintStream` or `:java.io.Writer`.

Prints the values, separated by spaces if there is more than one. `print` and `println` print in a human readable form.

If the printed data needs to be read back by a Venice reader use the functions `pr` and `prn` instead.

Returns `nil`.

```
(print [10 20 30])
[10 20 30]
=> nil
```

```
(print *out* [10 20 30])
[10 20 30]
=> nil
```

```
(print *err* [10 20 30])
[10 20 30]
=> nil
```

SEE ALSO

[println](#)

Prints the values xs to the stream that is the current value of `*out*` or to the passed output stream os if given followed by a (newline).

[printf](#)

Without output stream prints formatted output as per format to the stream that is the current value of `*out*`. With a stream prints ...

[newline](#)

Without arg writes a platform-specific newline to the output channel that is the current value of `*out*`. With arg writes a newline ...

[section](#)[top](#)

printf

```
(printf fmt & args)
(printf os fmt & args)
```

Without output stream prints formatted output as per format to the stream that is the current value of `*out*`. With a stream prints to that stream that must be a subclass of either `:java.io.PrintStream` or `:java.io.Writer`.

Prints like `print` and `println` in a human readable form.

Returns `nil`.

See: [Java Formatter](#)

```
(printf "%s: %d" "abc" 100)
abc: 100
=> nil
```

```
(printf "line 1: %s\nline 2: %s\n" "123" "456")
line 1: 123
line 2: 456
=> nil
```

```
(printf "%d%" 42)
42%
=> nil
```

```
(printf *out* "%s: %d" "abc" 100)
abc: 100
=> nil
```

```
(printf *err* "%s: %d" "abc" 100)
abc: 100
=> nil
```

SEE ALSO

[print](#)

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed stream `os` that must be a subclass of either ...

[println](#)

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed output stream `os` if given followed by a (newline).

[newline](#)

Without `arg` writes a platform-specific newline to the output channel that is the current value of `*out*`. With `arg` writes a newline ...

[section](#)[top](#)

println

```
(println & xs)
(println os & xs)
```

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed output stream `os` if given followed by a (newline) .
The passed stream must be a subclass of either `:java.io.PrintStream` or `:java.io.Writer` .

Prints the values, separated by spaces if there is more than one. `print` and `println` print in a human readable form.

If the printed data needs to be read back by a Venice reader use the functions `pr` and `prn` instead.

Returns `nil` .

```
(println 200)
200
=> nil

(println [10 20 30])
[10 20 30]
=> nil

(println *out* 200)
200
=> nil

(println *err* 200)
200
=> nil
```

SEE ALSO

[print](#)

Prints the values `xs` to the stream that is the current value of `*out*` or to the passed stream `os` that must be a subclass of either ...

[printf](#)

Without output stream prints formatted output as per `format` to the stream that is the current value of `*out*`. With a stream prints ...

[newline](#)

Without `arg` writes a platform-specific newline to the output channel that is the current value of `*out*`. With `arg` writes a newline ...

[section](#)[top](#)

prn

```
(prn & xs)
```

```
(prn os & xs)
```

Prints the values `xs` to the output stream that is the current value of `*out*` or to the passed stream `os` if given followed by a `(newline)` . The passed stream must be a subclass of either `:java.io.PrintStream` or `:java.io.Writer` .

Prints the values, separated by spaces if there is more than one. `pr` and `prn` print in a way that objects can be read by the reader.

Returns `nil` .

```
(prn "hello")
```

```
"hello"
```

```
=> nil
```

```
(prn {:foo "hello" :bar 34.5})
```

```
{:foo "hello" :bar 34.5}
```

```
=> nil
```

```
(prn ['a :b "\n" #\space "c"])
```

```
[a :b "\n" #\space "c"]
```

```
=> nil
```

```
(prn *out* [10 20 30])
```

```
[10 20 30]
```

```
=> nil
```

```
(prn *err* [10 20 30])
```

```
[10 20 30]
```

```
=> nil
```

SEE ALSO

[pr](#)

Prints the values `xs` to the output stream that is the current value of `*out*` or to the passed output stream `os` if given. The passed ...

[newline](#)

Without `arg` writes a platform-specific newline to the output channel that is the current value of `*out*`. With `arg` writes a newline ...

[pr-str](#)

With no args, returns the empty string. With one arg `x`, returns `x.toString()`. With more than one arg, returns the concatenation of ...

[section](#)

[top](#)

prof

```
(prof opts)
```

Controls the code profiling. See the companion functions/macros `'dorun'` and `'perf'`. The `perf` macro is built on `prof` and `dorun` and provides all for simple Venice profiling.

The profiler reports a function's elapsed time as "time with children"!

Profiling recursive functions:

Because the profiler reports "time with children" and accumulates the elapsed time across all recursive calls the resulting time for a particular recursive function is higher than the effective time.

```
(do
```

```
  (prof :on)    ; turn profiler on
```

```
  (prof :off)   ; turn profiler off
```

```
  (prof :status) ; returns the profiler on/off status
```

```
  (prof :clear) ; clear profiler data captured so far
```

```
(prof :data) ; returns the profiler data as map
(prof :data-formatted) ; returns the profiler data as formatted text
(prof :data-formatted "Metrics") ; returns the profiler data as formatted text with a title
nil)
=> nil
```

SEE ALSO

[perf](#)

Performance test with the given expression.

[time](#)

Evaluates expr and prints the time it took. Returns the value of expr.

[section](#)

[top](#)

promise

```
(promise)
(promise fn)
```

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, unless the variant of `deref` with timeout is used. All subsequent derefs will return the same delivered value without blocking.

Promises are implemented on top of Java's `CompletableFuture`.

```
(do
  (def p (promise))
  (deliver p 10)
  (deliver p 20) ; no effect
  @p)
=> 10

;; deliver the promise from a future
(do
  (def p (promise))
  (defn task1 [] (sleep 500) (deliver p 10))
  (defn task2 [] (sleep 800) (deliver p 20))
  (future task1)
  (future task2)
  @p)
=> 10

;; deliver the promise from a task's return value
(do
  (defn task [] (sleep 300) 10)
  (def p (promise task))
  @p)
=> 10

(let [p (promise #(do (sleep 300) 10))]
  @p)
=> 10
```

SEE ALSO

[deliver](#)

Delivers the supplied value to the promise, releasing any pending derefs. A subsequent call to `deliver` on a promise will have no effect.

[promise?](#)

Returns true if f is a Promise otherwise false

[realized?](#)

Returns true if a value has been produced for a promise, delay, or future.

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[done?](#)

Returns true if the future or promise is done otherwise false

[cancel](#)

Cancels a future or a promise

[cancelled?](#)

Returns true if the future or promise is cancelled otherwise false

[all-of](#)

Returns a new promise that is completed when all of the given promises complete. If any of the given promises complete exceptionally, ...

[any-of](#)

Returns a new promise that is completed when any of the given promises complete, with the same result. Otherwise, if it completed exceptionally, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function f with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function f with the two ...

[then-apply](#)

Applies a function f on the result of the previous stage of the promise p.

[then-combine](#)

Applies a function f to the result of the previous stage of promise p and the result of another promise p-other

[then-compose](#)

Composes the result of two promises. f receives the result of the first promise p and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise p with the same result or exception at this stage, that executes the action f. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a TimeoutException if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[timeout-after](#)

Returns a promise that timeouts after the specified time. The promise throws a TimeoutException.

[section](#)

[top](#)

promise?

(promise? p)

Returns true if f is a Promise otherwise false

```
(promise? (promise))
```

```
=> true
```

proxify

(proxify interface method-map)

Proxies a Java interface to be passed as a Callback object to Java functions. The interface's methods are implemented by Venice functions.

The dynamic invocation handler takes care that the methods are called in the context of a Venice sandbox even if the Java method that invokes the callback methods is running in another thread.

Supports default method implementations in the proxied Java interface. These Java interface methods can be either overridden by a Venice function or just be omitted. In the latter case the return value of methods default implementation will be handed back.

In case a Java `FunctionalInterface` is required the proxy wrappers from the `:java` module are often simpler to use:

- `java/as-runnable`
- `java/as-callable`
- `java/as-predicate`
- `java/as-function`
- `java/as-consumer`
- `java/as-supplier`
- `java/as-bipredicate`
- `java/as-bifunction`
- `java/as-biconsumer`
- `java/as-binaryoperator`

```
(do
  (import :java.io.File :java.io FilenameFilter)

  (def file-filter
    (fn [dir name] (str/ends-with? name ".xxx"))))

  (let [dir (io/tmp-dir)]
    ;; create a dynamic proxy for the interface FilenameFilter
    ;; and implement its function 'accept' by 'file-filter'
    (. dir :list (proxify :FilenameFilter {:accept file-filter}))))
=> []
```

;; Instead of explicit proxies, functional interface wrappers are
;; often simpler to use

```
(do
  (load-module :java)
  (import :java.util.stream.Collectors)

  (-> (. [1 2 3 4] :stream)
    (. :filter (java/as-predicate #(> % 2)))
    (. :map (java/as-function #(* % 10)))
    (. :collect (. :Collectors :toList)))
=> (30 40)
```

SEE ALSO

[java/as-runnable](#)

Wraps the function `f` in a `java.lang Runnable` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html>)

[java/as-callable](#)

Wraps the function `f` in a `java.util.concurrent Callable` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html>)

[java/as-predicate](#)

Wraps the function f in a java.util.function.Predicate (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

[java/as-function](#)

Wraps the function f in a java.util.function.Function (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

[java/as-consumer](#)

Wraps the function f in a java.util.function.Consumer (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

[java/as-supplier](#)

Wraps the function f in a java.util.function.Supplier (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html>)

[java/as-bipredicate](#)

Wraps the function f in a java.util.function.BiPredicate (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html>)

[java/as-bifunction](#)

Wraps the function f in a java.util.function.BiFunction (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html>)

[java/as-biconsumer](#)

Wraps the function f in a java.util.function.BiConsumer (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html>)

[java/as-binaryoperator](#)

Wraps the function f in a java.util.function.BinaryOperator (<https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html>)

[section](#)[top](#)

push!

```
(push! stack v)
```

Pushes an item to a stack.

```
(let [s (stack)]
  (push! s 1)
  (push! s 2)
  (push! s 3)
  (pop! s))
=> 3
```

SEE ALSO

[stack](#)

Creates a new mutable threadsafe stack.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[pop!](#)

Pops an item from a stack.

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)[top](#)

put!

```
(put! queue val)
```

```
(put! queue val delay)
```

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

queue: (put! queue val)

Puts the value 'val' to the tail of the queue.

delay-queue: (put! queue val delay)

Puts the value 'val' with a delay of 'delay' milliseconds to a delay-queue

```
(let [q (queue)]
  (put! q 1)
  (poll! q)
  q)
=> ()

(let [q (delay-queue)]
  (put! q 1 100)
  (take! q))
=> 1
```

SEE ALSO

[queue](#)

Creates a new mutable threadsafe bounded or unbounded queue.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)[top](#)

put-head!

```
(put-head! deque val)
```

Puts an item to the head of a deque. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

```
(let [q (deque)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (put-head! q 0)
  (println (poll! q))
  (println q))
```

```
0
(1 2 3)
=> nil
```

SEE ALSO

[deque](#)

Creates a new mutable threadsafe bounded or unbounded deque.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[take-tail!](#)

Retrieves and removes the tail value of the deque, waiting if necessary until a value becomes available.

[offer-head!](#)

Offers an item to the head of a deque with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll-tail!](#)

Polls an item from the tail of a deque with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)[top](#)

qrbill/address

```
(address name line1 line2)
(address name street house-no postal-code town)
```

Create a combined or structured address

Note: Starting November 25, 2025, banks will no longer accept payments using combined address elements!

```
;; structured addresses
(do
  (load-module :qrbill ['qrbill :as 'q])

  (q/address "Peter Meier"
             "Bahnhofstrasse" "2"
             "8000" "Wil"))

;; combined addresses
(do
  (load-module :qrbill ['qrbill :as 'q])

  (q/address "Peter Meier"
             "Bahnhofstrasse 2"
             "8000 Wil"))
```

SEE ALSO

[qrbill/bill](#)

Create a bill data object

[qrbill/write](#)

Write the bill to a file.

[section](#)

[top](#)

qrbill/bill

(bill account amount qr-ref msg debtor creditor)

Create a bill data object

Note: Starting November 25, 2025, banks will no longer accept payments using combined address elements!

```
(do
  (load-module :qrbill ['qrbill :as 'q])

  ;; use structured debtor and creditor addresses
  (let [debtor    (q/address "Peter Meier"
                             "Bahnhofstrasse" "2"
                             "8000" "Wil")
        creditor  (q/address "Travag GmbH"
                             "Bahnhofstrasse" "3"
                             "8000" "Wil")]
    (q/bill "CH00 3000 2000 1000 0000 0"
            300.00M
            "000004"
            "Testrechnung 4"
            debtor
            creditor)))
```

SEE ALSO

[qrbill/address](#)

Create a combined or structured address

[qrbill/write](#)

Write the bill to a file.

[section](#)

[top](#)

qrbill/write

(write bill out-file)
(write bill payment-slip-pdf-type invoice-pdf-file out-dir)

Write the bill to a file.

Note: Starting November 25, 2025, banks will no longer accept payments using combined address elements!

```
(do
  (load-module :qrbill ['qrbill :as 'q])

  ;; use structured debtor and creditor addresses
```

```

(let [debtor    (q/address "Peter Meier"
                          "Bahnhofstrasse" "2"
                          "8000" "Wil")
      creditor  (q/address "Travag GmbH"
                          "Bahnhofstrasse" "3"
                          "8000" "Wil")
      bill      (q/bill "CH00 3000 2000 1000 0000 0"
                       300.00M
                       "000004"
                       "Testrechnung 4"
                       debtor
                       creditor)]

;; create a PDF invoice with just the payment slip
(q/write bill "./Meier-Rechnung-1.pdf")

;; create a PDF invoice with the payment slip added on the last
;; page of the file "./Meier-Rechnung-2.pdf"
(q/write bill :last-page "./Meier-Rechnung-2.pdf" ".")

;; create a PDF invoice with the payment slip added on a new
;; page to the file "./Meier-Rechnung-3.pdf"
(q/write bill :new-page-at-end "./Meier-Rechnung-3.pdf" ".)))

(do
  (load-module :qrbill ['qrbill :as 'q])

  ;; use structured debtor and creditor addresses
  ;; the fields :debtor, :amount, :message, and :file-in are optional
  (q/write { :debtor      { :name "Peter Meier"
                          :street "Aarestrasse"
                          :house-no "2"
                          :postal-code "9500"
                          :town "Wil" }
            :creditor    { :name "Kenag GmbH"
                          :street "Aarestrasse"
                          :house-no "3"
                          :postal-code "9500 "
                          :town "Wil" }
            :bill        { :account "CH00 3000 2000 1000 0000 0"
                          :amount "300.00"
                          :reference "000004"
                          :message "Testrechnung 4" }
            :qr-slip-pos :new-page-at-end ;; :last-page
            :file-in     "./Meier-Rechnung-1.pdf"
            :file-out     "./Meier-Rechnung-QR-1.pdf" })))

```

SEE ALSO

[qrbill/address](#)

Create a combined or structured address

[qrbill/bill](#)

Create a bill data object

[section](#)

[top](#)

qrcode/decode

(decode source)

Decode a QR code image. Returns the text decoded from the QR code image.

Loads the image from a `:java.io.File`, `:java.io.InputStream` or a `:java.awt.image.BufferedImage`

```
(do
  (load-module :qrcode)

  (qrcode/decode (io/file "./example.png")))
```

SEE ALSO

[qrcode/encode](#)

Encodes text into a PNG QR code image. Writes the generated QR code image to the supplied output stream.

[section](#)

[top](#)

qrcode/encode

```
(encode text width height margin os)
```

Encodes text into a PNG QR code image. Writes the generated QR code image to the supplied output stream.

Arguments:

text	The text to encode in the QR code image
width	the image width in pixels
height	the image height in pixels
margin	the image margin in pixels
os	The output stream to write the image to

```
(do
  (load-module :qrcode)

  (qrcode/encode "Hello" 600 600 3 (io/file-out-stream "./example.png")))
```

SEE ALSO

[qrcode/decode](#)

Decode a QR code image. Returns the text decoded from the QR code image.

[section](#)

[top](#)

qrref/checksum

```
(checksum ref)
```

Computes the checksum for a raw reference.

The passed ref may be a raw QR reference or a QR reference with a checksum digit. It may contain spaces.

Returns the computed checksum digit 0..9.

If the passed ref is a QR reference with a correct checksum digit the computed checksum digit will always be 0. This fact is used for QR reference validation!

```
(do
  (load-module :qrref ['qrref :as 'qr])
```



```
(qr/checksum "230 55361 34663 9301")
(qr/checksum "23055361346639301")
(qr/checksum "00 00000 00230 55361 34663 9301")
(qr/checksum "00000000023055361346639301"))
=> 3
```

SEE ALSO

[qrref/create](#)

Creates a QR reference according to the Swiss payment standards.

[qrref/valid?](#)

Returns true if ref is a valid QR reference else false. The reference may contain spaces.

[qrref/format](#)

Format a QR reference.

[section](#)

[top](#)

qrref/create

```
(create ref-raw)
```

Creates a QR reference according to the Swiss payment standards.

A QR reference has 27 digits. The raw reference plus a checksum digit as the last digit.

The raw reference passed must not have more than 26 digits. With less than 26 digits leading '0' will be used to fill up to 26 digits.

Raw reference: "23055361346639301"

QR reference: "000000000230553613466393013"

The QR reference can be formatted to "00 00000 00230 55361 34663 93013" using:

```
(qrref/format "000000000230553613466393013")
```

[Swiss Payment Standards / de](#)

[Swiss Payment Standards / en](#)

```
(do
  (load-module :qrref ['qrref :as 'qr])
  (qr/create "1234")
  (qr/create "23055361346639301"))
=> "000000000230553613466393013"
```

SEE ALSO

[qrref/valid?](#)

Returns true if ref is a valid QR reference else false. The reference may contain spaces.

[qrref/format](#)

Format a QR reference.

[qrref/checksum](#)

Computes the checksum for a raw reference.

[section](#)

[top](#)

qrref/format

```
(format ref)
```

Format a QR reference.

```
(do
  (load-module :qrref ['qrref :as 'qr])
  (qr/format "000000000230553613466393013"))
=> "00 00000 00230 55361 34663 93013"
```

SEE ALSO

[qrref/create](#)

Creates a QR reference according to the Swiss payment standards.

[qrref/valid?](#)

Returns true if ref is a valid QR reference else false. The reference may contain spaces.

[qrref/checksum](#)

Computes the checksum for a raw reference.

[section](#)

[top](#)

qrref/valid?

```
(valid? ref)
```

Returns true if ref is a valid QR reference else false. The reference may contain spaces.

A valid QR reference must have 27 digits and the checksum must be correct. The last digit is the checksum digits for the first 26 digits.

```
(do
  (load-module :qrref ['qrref :as 'qr])
  (qr/valid? "000000000230553613466393013")
  (qr/valid? "00 00000 00230 55361 34663 93013"))
=> true
```

SEE ALSO

[qrref/create](#)

Creates a QR reference according to the Swiss payment standards.

[qrref/format](#)

Format a QR reference.

[qrref/checksum](#)

Computes the checksum for a raw reference.

[section](#)

[top](#)

qualified-name

```
(name x)
```

Returns the qualified name String of a string, symbol, keyword, or function

```
(qualified-name :user/x)
=> "user/x"
```

```
(qualified-name 'x)
=> "x"

(qualified-name "x")
=> "x"

(qualified-name str/digit?)
=> "str/digit?"
```

SEE ALSO

[name](#)

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[fn-name](#)

Returns the qualified name of a function or macro

[section](#)[top](#)

qualified-symbol?

```
(qualified-symbol? x)
```

Returns true if x is a qualified symbol

```
(qualified-symbol? 'foo/a)
=> true
```

```
(qualified-symbol? (symbol "foo/a"))
=> true
```

```
(qualified-symbol? 'a)
=> false
```

```
(qualified-symbol? nil)
=> false
```

```
(qualified-symbol? :a)
=> false
```

[section](#)[top](#)

quasiquote

```
(quasiquote form)
```

Quasi quotes also called syntax quotes (a backquote) suppress evaluation of the form that follows it and all the nested forms.

unquote:

It is possible to unquote part of the form that is quoted with `~`. Unquoting allows you to evaluate parts of the syntax quoted expression.

unquote-splicing:

Unquote evaluates to a collection of values and inserts the collection into the quoted form. But sometimes you want to unquote a list and insert its elements (not the list) inside the quoted form. This is where `~@` (unquote-splicing) comes to rescue.

```
(quasiquote (16 17 (inc 17)))  
=> (16 17 (inc 17))
```

```
`(16 17 (inc 17))  
=> (16 17 (inc 17))
```

```
`(16 17 ~(inc 17))  
=> (16 17 18)
```

```
`(16 17 ~(map inc [16 17]))  
=> (16 17 (17 18))
```

```
`(16 17 ~@(map inc [16 17]))  
=> (16 17 17 18)
```

```
`(1 2 ~@#{1 2 3})  
=> (1 2 1 2 3)
```

```
`(1 2 ~@{:a 1 :b 2 :c 3})  
=> (1 2 [:a 1] [:b 2] [:c 3])
```

SEE ALSO

[quote](#)

There are two equivalent ways to quote a form either with `quote` or with `'`. They prevent the quoted form from being evaluated.

[section](#)

[top](#)

queue

```
(queue)  
(queue capacity)
```

Creates a new mutable threadsafe bounded or unbounded queue.

The queue can be turned into a synchronous queue when using the functions `put!` and `take!`. `put!` waits until the value be added and `take!` waits until a value is available from queue thus synchronizing the producer and consumer.

```
; unbounded queue  
(let [q (queue)]  
  (offer! q 1)  
  (offer! q 2)  
  (offer! q 3)  
  (poll! q)  
  q)  
=> (2 3)  
  
; unbounded queue  
(let [q (conj! (queue) 1 2 3)]  
  (offer! q 4)  
  (poll! q)  
  q)  
=> (2 3 4)
```

```

; bounded queue
(let [q (queue 10)]
  (offer! q 1000 1)
  (offer! q 1000 2)
  (offer! q 1000 3)
  (poll! q 1000)
  q)
=> (2 3)

; bounded queue
(let [q (conj! (queue 10) 1 2 3)]
  (offer! q 4)
  (poll! q)
  q)
=> (2 3 4)

; synchronous unbounded queue
(let [q (queue)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (take! q)
  q)
=> (2 3)

; synchronous bounded queue
(let [q (queue 10)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (take! q)
  q)
=> (2 3)

```

SEE ALSO

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[take!](#)

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[empty](#)

Returns an empty collection of the same category as coll, or nil if coll is nil. If the collection is mutable clears the collection ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[queue?](#)

Returns true if coll is a queue

[reduce](#)

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

transduce

Reduce with a transformation of a reduction function $f(xf)$. If `init` is not supplied, (f) will be called to produce it. f should be ...

docoll

Applies f to the items of the collection presumably for side effects. Returns `nil`.

into!

Adds all of the items of 'from' conjoined to the mutable 'to' collection

conj!

Returns a new mutable collection with the x , xs 'added'. $(conj! \text{ nil } item)$ returns $(item)$ and $(conj! item)$ returns $item$.

[section](#)

[top](#)

queue?

```
(queue? coll)
```

Returns true if `coll` is a queue

```
(queue? (queue))  
=> true
```

[section](#)

[top](#)

quote

```
(quote form)
```

There are two equivalent ways to quote a form either with `quote` or with `'`. They prevent the quoted form from being evaluated.

Regular quotes work recursively with any kind of forms and types: strings, maps, lists, vectors...

```
(quote (1 2 3))  
=> (1 2 3)
```

```
(quote (+ 1 2))  
=> (+ 1 2)
```

```
'(1 2 3)  
=> (1 2 3)
```

```
'(+ 1 2)  
=> (+ 1 2)
```

```
'(a (b (c d (+ 1 2))))  
=> (a (b (c d (+ 1 2))))
```

SEE ALSO

quasiquote

Quasi quotes also called syntax quotes (a backquote) suppress evaluation of the form that follows it and all the nested forms.

[section](#)

[top](#)

rand-bigint

(rand-bigint bits)

Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to ($2^N - 1$), inclusive.

```
(rand-bigint 256)
=> 17013671328540605827529581245368229705928642456813034585973099873888808971377N
```

SEE ALSO

[rand-long](#)

Without argument returns a random long between 0 and MAX_LONG. With argument max returns a random long between 0 and max exclusive.

[rand-double](#)

Without argument returns a double between 0.0 and 1.0. With argument max returns a random double between 0.0 and max.

[rand-gaussian](#)

Without argument returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0. With argument mean and stddev ...

[bytebuf-allocate-random](#)

Allocates a new bytebuf. The buffer will be preset with random bytes.

[section](#)

[top](#)

rand-double

(rand-double)
(rand-double max)

Without argument returns a double between 0.0 and 1.0. With argument max returns a random double between 0.0 and max.

This function is based on a cryptographically strong random number generator (RNG).

```
(rand-double)
=> 0.15255073914265427
```

```
(rand-double 100.0)
=> 43.70485616758688
```

SEE ALSO

[rand-long](#)

Without argument returns a random long between 0 and MAX_LONG. With argument max returns a random long between 0 and max exclusive.

[rand-bigint](#)

Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to ($2^N - 1$), inclusive.

[rand-gaussian](#)

Without argument returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0. With argument mean and stddev ...

[bytebuf-allocate-random](#)

Allocates a new bytebuf. The buffer will be preset with random bytes.

[section](#)

[top](#)

rand-gaussian

```
(rand-gaussian)
(rand-gaussian mean stddev)
```

Without argument returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0. With argument mean and stddev returns a Gaussian distributed double value with the given mean and standard deviation.

This function is based on a cryptographically strong random number generator (RNG)

```
(rand-gaussian)
=> -0.17052458289402356

(rand-gaussian 0.0 5.0)
=> 4.201224397186673
```

SEE ALSO

[rand-long](#)

Without argument returns a random long between 0 and MAX_LONG. With argument max returns a random long between 0 and max exclusive.

[rand-double](#)

Without argument returns a double between 0.0 and 1.0. With argument max returns a random double between 0.0 and max.

[rand-bigint](#)

Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to $(2^N - 1)$, inclusive.

[bytebuf-allocate-random](#)

Allocates a new bytebuf. The buffer will be preset with random bytes.

[section](#)

[top](#)

rand-long

```
(rand-long)
(rand-long max)
```

Without argument returns a random long between 0 and MAX_LONG. With argument max returns a random long between 0 and max exclusive.

This function is based on a cryptographically strong random number generator (RNG).

```
(rand-long)
=> 2757395408709606722

(rand-long 100)
=> 46
```

SEE ALSO

[rand-double](#)

Without argument returns a double between 0.0 and 1.0. With argument max returns a random double between 0.0 and max.

[rand-bigint](#)

Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to $(2^N - 1)$, inclusive.

[rand-gaussian](#)

Without argument returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0. With argument mean and stddev ...

[bytebuf-allocate-random](#)

Allocates a new bytebuf. The buffer will be preset with random bytes.

[section](#)

[top](#)

range

```
(range)
(range end)
(range start end)
(range start end step)
```

Returns a collection of numbers from start (inclusive) to end (exclusive), by step, where start defaults to 0 and step defaults to 1. When start is equal to end, returns empty list. Without args returns a lazy sequence generating numbers starting with 0 and incrementing by 1.

```
(range 10)
=> (0 1 2 3 4 5 6 7 8 9)
```

```
(range 10 20)
=> (10 11 12 13 14 15 16 17 18 19)
```

```
(range 10 20 3)
=> (10 13 16 19)
```

```
(range (int 10) (int 20))
=> (10I 11I 12I 13I 14I 15I 16I 17I 18I 19I)
```

```
(range (int 10) (int 20) (int 3))
=> (10I 13I 16I 19I)
```

```
(range 10 15 0.5)
=> (10 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5)
```

```
(range 1.1M 2.2M 0.1M)
=> (1.1M 1.2M 1.3M 1.4M 1.5M 1.6M 1.7M 1.8M 1.9M 2.0M 2.1M)
```

```
(range 100N 200N 10N)
=> (100N 110N 120N 130N 140N 150N 160N 170N 180N 190N)
```

```
;; capital letters
(map char (range (int #\A) (inc (int #\Z))))
=> (#\A #\B #\C #\D #\E #\F #\G #\H #\I #\J #\K #\L #\M #\N #\O #\P #\Q #\R #\S #\T #\U #\V #\W #\X #\Y #\Z)
```

[section](#) [top](#)

read-char

```
(read-char)
(read-char is)
```

Without arg reads the next char from the stream that is the current value of `*in*`. With arg reads the next char from the passed stream that must be a subclass of `:java.io.Reader`.

Returns `nil` if the end of the stream is reached.

```
(try-with [rd (io/buffered-reader "1234")]
  (println (read-char rd))
  (println (read-char rd)))
1
```

```
2
=> nil
```

SEE ALSO

[read-line](#)

Without arg reads the next line from the stream that is the current value of `*in*`. With arg reads the next line from the passed stream ...

[section](#)

[top](#)

read-line

```
(read-line)
(read-line is)
```

Without arg reads the next line from the stream that is the current value of `*in*`. With arg reads the next line from the passed stream that must be a subclass of `:java.io.BufferedReader`.

Returns `nil` if the end of the stream is reached.

```
(try-with [rd (io/buffered-reader "1\n2\n3\n4")])
  (println (read-line rd))
  (println (read-line rd)))
1
2
=> nil
```

SEE ALSO

[read-char](#)

Without arg reads the next char from the stream that is the current value of `*in*`. With arg reads the next char from the passed stream ...

[section](#)

[top](#)

read-string

```
(read-string s)
(read-string s origin)
```

Reads Venice source from a string and transforms its content into a Venice data structure, following the rules of the Venice syntax.

```
(do
  (eval (read-string "(def x 100)" "test")))
x)
=> 100
```

SEE ALSO

[eval](#)

Evaluates the form data structure (not text!) and returns the result.

[section](#)

[top](#)

realized?

(realized? x)

Returns true if a value has been produced for a promise, delay, or future.

```
(do
  (def task (fn [] 100))
  (let [f (future task)]
    (println (realized? f))
    (println @f)
    (println (realized? f))))
false
100
true
=> nil
```

```
(do
  (def p (promise))
  (println (realized? p))
  (deliver p 123)
  (println @p)
  (println (realized? p)))
false
123
true
=> nil
```

```
(do
  (def x (delay 100))
  (println (realized? x))
  (println @x)
  (println (realized? x)))
false
100
true
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[delay](#)

Takes a body of expressions and yields a Delay object that will invoke the body only the first time it is forced (with force or deref ...

[promise](#)

Returns a promise object that can be read with deref, and set, once only, with deliver. Calls to deref prior to delivery will block, ...

[section](#)

[top](#)

recur

(recur expr*)

Evaluates the exprs and rebinds the bindings of the recursion point to the values of the exprs. The recur expression must be at the tail position. The tail position is a position which an expression would return a value from.

```
;; tail recursion
(loop [x 10]
  (when (> x 1)
```

```

      (println x)
      (recur (- x 2))))
10
8
6
4
2
=> nil

```

```

;; tail recursion
(do
  (defn sum [n]
    (loop [cnt n acc 0]
      (if (zero? cnt)
        acc
        (recur (dec cnt) (+ acc cnt)))))
    (sum 10000))
=> 50005000

```

SEE ALSO

[loop](#)

Evaluates the exprs and binds the bindings. Creates a recursion point with the bindings.

[section](#)

[top](#)

reduce

```

(reduce f coll)
(reduce f val coll)

```

`f` should be a function of 2 arguments. If `val` is not supplied, returns the result of applying `f` to the first 2 items in `coll`, then applying `f` to that result and the 3rd item, etc. If `coll` contains no items, `f` must accept no arguments as well, and `reduce` returns the result of calling `f` with no arguments. If `coll` has only 1 item, it is returned and `f` is not called. If `val` is supplied, returns the result of applying `f` to `val` and the first item in `coll`, then applying `f` to that result and the 2nd item, etc. If `coll` contains no items, returns `val` and `f` is not called.

`reduce` can work with queues as collection, given that the end of the queue is marked by adding a `nil` element. Otherwise the reducer does not know when to stop reading elements from the queue.

```

(reduce + [1 2 3 4 5 6 7])
=> 28

```

```

(reduce + 10 [1 2 3 4 5 6 7])
=> 38

```

```

(reduce (fn [x y] (+ x y 10)) [1 2 3 4 5 6 7])
=> 88

```

```

(reduce (fn [x y] (+ x y 10)) 10 [1 2 3 4 5 6 7])
=> 108

```

```

((reduce comp [(partial + 1) (partial * 2) (partial + 3)]) 100)
=> 207

```

```

(reduce (fn [m [k v]] (assoc m k v)) {} [[:a 1] [:b 2] [:c 3]])
=> {:a 1 :b 2 :c 3}

```

```

(reduce (fn [m [k v]] (assoc m v k)) {} {:b 2 :a 1 :c 3})
=> {1 :a 2 :b 3 :c}

(reduce (fn [m c] (assoc m (first c) c)) {} [[:a 1] [:b 2] [:c 3]])
=> {:a [:a 1] :b [:b 2] :c [:c 3]}

;; sliding window (width 3) average
(->> (partition 3 1 (repeatedly 10 #(rand-long 30)))
      (map (fn [window] (/ (reduce + window) (count window)))))
=> (17 18 17 11 7 15 20 25)

;; reduce all elements of a queue.
;; calls (take! queue) to get the elements of the queue.
;; note: use nil to mark the end of the queue otherwise
;;       reduce will block forever!
(let [q (conj! (queue) 1 2 3 4 5 6 7 nil)]
  (reduce + q))
=> 28

;; reduce data supplied by a finit lazy seq
(do
  (def counter (atom 5))
  (defn generate []
    (swap! counter dec)
    (if (pos? @counter) @counter nil))
  (reduce + 100 (lazy-seq generate)))
=> 110

```

SEE ALSO

[reduce-kv](#)

Reduces an associative collection. f should be a function of 3 arguments. Returns the result of applying f to init, the first key and ...

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[filter](#)

Returns a collection of the items in coll for which (predicate item) returns logical true.

[section](#)
[top](#)

reduce-kv

```
(reduce-kv f init coll)
```

Reduces an associative collection. f should be a function of 3 arguments. Returns the result of applying f to init, the first key and the first value in coll, then applying f to that result and the 2nd key and value, etc. If coll contains no entries, returns init and f is not called. Note that reduce-kv is supported on vectors, where the keys will be the ordinals.

```

(reduce-kv (fn [m k v] (assoc m v k))
           {}
           {:a 1 :b 2 :c 3})
=> {1 :a 2 :b 3 :c}

(reduce-kv (fn [m k v] (assoc m k (:col v)))
           {}
           {:a {:col :red :len 10}
            :b {:col :green :len 20}}

```

```
      :c {:col :blue :len 30} })
=> {:a :red :b :green :c :blue}
```

SEE ALSO

[reduce](#)

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[filter](#)

Returns a collection of the items in coll for which (predicate item) returns logical true.

[section](#)[top](#)

reduced

```
(reduced x)
```

Wraps x in a way such that a reduce will terminate with the value x.

[section](#)[top](#)

reduced?

```
(reduced? x)
```

Returns true if x is the result of a call to reduced.

[section](#)[top](#)

regex/count

```
(regex/count matcher)
```

Returns the matcher's group count.

```
(let [m (regex/matcher #"([0-9]+)(.*)" "100abc")]
      (regex/count m))
=> 2
```

SEE ALSO

[match?](#)

Returns true if the string s matches the regular expression regex.

[regex/matcher](#)

Returns an instance of java.util.regex.Matcher.

[regex/pattern](#)

Returns an instance of java.util.regex.Pattern.

regex/find

```
(regex/find matcher)
(regex/find pattern s)
```

Returns the next regex match or nil if there is no further match. Returns `nil` if there is no match.

To get the positional data for the matched group use `(regex/find+ matcher)`.

```
(regex/find #"[0-9]+" "672-345-456-3212")
=> "672"

(let [m (regex/matcher #"[0-9]+" "672-345-456-3212")]
  (println (regex/find m))
  (println (regex/find m))
  (println (regex/find m))
  (println (regex/find m))
  (println (regex/find m)))
672
345
456
3212
nil
=> nil
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[regex/find+](#)

Returns the next regex match and returns the group with its positional data. Returns nil if there is no match.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

regex/find+

```
(regex/find+ matcher)
(regex/find+ pattern s)
```

Returns the next regex match and returns the group with its positional data. Returns `nil` if there is no match.

```
(regex/find+ #"[0-9]+" "672-345-456-3212")
=> {:start 0 :end 3 :group "672"}

(let [m (regex/matcher #"[0-9]+" "672-345-456-3212")]
  (println (regex/find+ m)))
```

```
(println (regex/find+ m))
(println (regex/find+ m))
(println (regex/find+ m))
(println (regex/find+ m))

{:start 0 :end 3 :group 672}
{:start 4 :end 7 :group 345}
{:start 8 :end 11 :group 456}
{:start 12 :end 16 :group 3212}
nil
=> nil
```

SEE ALSO

[match?](#)

Returns true if the string *s* matches the regular expression *regex*.

[regex/find-all+](#)

Returns the all *regex* matches and returns the groups with its positional data. Returns an empty list if there are no matches.

[regex/find](#)

Returns the next *regex* match or nil if there is no further match. Returns nil if there is no match.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

regex/find-all

```
(regex/find-all matcher)
(regex/find-all pattern s)
```

Returns all *regex* matches as list or an empty list if there are no matches.

To get the positional data for the matched groups use '*regex/find-all+*'.

```
(regex/find-all #"\\d+" "672-345-456-3212")
=> ("672" "345" "456" "3212")

(->> (regex/matcher #"\\d+" "672-345-456-3212")
      (regex/find-all))
=> ("672" "345" "456" "3212")

(->> (regex/matcher "([\\^\"\\\\S*|\\\".+?\\\")\\\\s*" "1 2 \\\"3 4\\\" 5")
      (regex/find-all))
=> ("1 " "2 " "\\\"3 4\\\" " "5")
```

SEE ALSO

[match?](#)

Returns true if the string *s* matches the regular expression *regex*.

[regex/find](#)

Returns the next *regex* match or nil if there is no further match. Returns nil if there is no match.

[regex/find-all+](#)

Returns the all *regex* matches and returns the groups with its positional data. Returns an empty list if there are no matches.

[regex/groups](#)

Attempts to match the entire region against the pattern and returns all matched groups. The entire regions is the first item in the ...

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)[top](#)

regex/find-all+

```
(regex/find-all+ matcher)
(regex/find-all+ pattern s)
```

Returns the all regex matches and returns the groups with its positional data. Returns an empty list if there are no matches.

```
(regex/find-all+ #"[0-9]+" "672-345-456-3212")
=> ({:start 0 :end 3 :group "672"} {:start 4 :end 7 :group "345"} {:start 8 :end 11 :group "456"} {:start 12 :
end 16 :group "3212"})

(let [m (regex/matcher #"[0-9]+" "672-345-456-3212")]
  (regex/find-all+ m))

=> ({:start 0 :end 3 :group "672"} {:start 4 :end 7 :group "345"} {:start 8 :end 11 :group "456"} {:start 12 :
end 16 :group "3212"})
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/find+](#)

Returns the next regex match and returns the group with its positional data. Returns nil if there is no match.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[regex/groups](#)

Attempts to match the entire region against the pattern and returns all matched groups. The entire regions is the first item in the ...

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)[top](#)

regex/find?

```
(regex/find? matcher)
```

Attempts to find the next subsequence that matches the pattern. If the match succeeds then more information can be obtained via the `regex/group` function

```
(let [m (regex/matcher #"[0-9]+" "100")]
  (regex/find? m))
=> true

(let [m (regex/matcher #"[0-9]+" "xxx: 100")]
  (regex/find? m))
=> true

(let [m (regex/matcher #"[0-9]+" "xxx: 100 200")]
  (when (regex/find? m)
    (println (regex/group m 0))))
  (when (regex/find? m)
    (println (regex/group m 0))))
  (when (regex/find? m)
    (println (regex/group m 0))))
100
200
=> nil
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/group](#)

Returns the input subsequence captured by the given group during the previous match operation.

[regex/matches?](#)

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

regex/group

```
(regex/group matcher group)
```

Returns the input subsequence captured by the given group during the previous match operation.

Note: Do not forget to call the `regex/matches?` function!

```
(let [m (regex/matcher #"(\d+)(.*)" "100abc")]
  (if (regex/matches? m)
    [(regex/group m 1) (regex/group m 2)]
    []))
=> ["100" "abc"]

(do
  (ns-alias 'r 'regex)
  (defn swap [s]
    (let [m (r/matcher #"(\d+)([^\d]*) (\d+)" s)]
      (if (r/matches? m)
        (str (r/group m 3) (r/group m 2) (r/group m 1))
        s)))
  (swap "100::200"))
=> "200::100"
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/groups](#)

Attempts to match the entire region against the pattern and returns all matched groups. The entire regions is the first item in the ...

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches?](#)

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

regex/groups

`(regex/groups matcher)`

Attempts to match the entire region against the pattern and returns all matched groups. The entire regions is the first item in the returned group list. Returns an empty list if the entire region does not match the pattern.

```
(let [m (regex/matcher #"(\d+)(.*)" "100abc")]
      (regex/groups m))
=> ("100abc" "100" "abc")

(let [m (regex/matcher #"(\d+)([a-z]+)" "100abc:")]
      (regex/groups m))
=> ()
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/group](#)

Returns the input subsequence captured by the given group during the previous match operation.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches?](#)

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

regex/matcher

`(regex/matcher pattern str)`

Returns an instance of `java.util.regex.Matcher` .
The pattern can be either a string or a pattern created by `(regex/pattern s)` .

Matchers are mutable and are not safe for use by multiple concurrent threads!

JavaDoc: [Pattern](#)

```
(regex/matcher #"[0-9]+" "100")  
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]
```

```
(regex/matcher (regex/pattern"[0-9]+" "100")  
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]
```

```
(regex/matcher "[0-9]+" "100")  
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[regex/matches?](#)

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

[regex/find?](#)

Attempts to find the next subsequence that matches the pattern. If the match succeeds then more information can be obtained via the ...

[regex/reset](#)

Resets the matcher with a new string

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/find](#)

Returns the next regex match or nil if there is no further match. Returns nil if there is no match.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[section](#)

[top](#)

regex/matches

```
(regex/matches pattern str)
```

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()` .

If the matcher's pattern matches the entire region sequence returns a list with the entire region sequence and the matched groups otherwise returns an empty list.

Returns matching info as meta data on the region and the groups.

Region meta data:

:start start pos of the overall group
:end end pos of the overall group
:group-count the number of matched elements groups

Group meta data:

:start start pos of the element group
:end end pos of the element group

JavaDoc: [Pattern](#)

```

;; Entire region sequence matched
(regex/matches "hello, (.*)" "hello, world")
=> ("hello, world" "world")

;; Entire region sequence not matched
(regex/matches "HEllo, (.*)" "hello, world")
=> ()

;; Matching multiple groups
(regex/matches "([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)" "672-345-456-212")
=> ("672-345-456-212" "672" "345" "456" "212")

;; Matching multiple groups
(let [p (regex/pattern "([0-9]+)-([0-9]+)")]
  (regex/matches p "672-345"))
=> ("672-345" "672" "345")

;; Access matcher's region meta info
(let [pattern "([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)"
      matches (regex/matches pattern "672-345-456-212")]
  (println "meta info:" (pr-str (meta matches)))
  (println "matches: " (pr-str matches)))
meta info: {:group-count 4 :start 0 :end 15}
matches:   ("672-345-456-212" "672" "345" "456" "212")
=> nil

;; Access matcher's region meta info and the meta info of each group
(let [pattern "([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)"
      matches (regex/matches pattern "672-345-456-212")]
  (println "region info: " (pr-str (meta matches)))
  (println "group count: " (count matches) "(region included)"))
  (println "group matches: " (pr-str (nth matches 0)) (meta (nth matches 0)))
  (println " " (pr-str (nth matches 1)) (meta (nth matches 1)))
  (println " " (pr-str (nth matches 2)) (meta (nth matches 2)))
  (println " " (pr-str (nth matches 3)) (meta (nth matches 3)))
  (println " " (pr-str (nth matches 4)) (meta (nth matches 4))))
region info:   {:group-count 4 :start 0 :end 15}
group count:   5 (region included)
group matches: "672-345-456-212" {:start 0 :end 15}
               "672" {:start 0 :end 3}
               "345" {:start 4 :end 7}
               "456" {:start 8 :end 11}
               "212" {:start 12 :end 15}

=> nil

```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

regex/matches-not?

```

(regex/matches-not? matcher)
(regex/matches-not? matcher str)

```

Attempts to match the entire region against the pattern. Returns false if the patterns matches the string else true.

```
(let [m (regex/matcher #"[0-9]+" "10A")]
  (regex/matches-not? m))
=> true

(let [m (regex/matcher #"[0-9]+" "value: 10A")]
  (regex/matches-not? m))
=> true

(let [m (regex/matcher #"[0-9]+" "")]
  (filter #(regex/matches-not? m %) ["100" "10A" "200"]))
=> ("10A")
```

SEE ALSO

[match?](#)

Returns true if the string *s* matches the regular expression *regex*.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

regex/matches?

```
(regex/matches? matcher)
(regex/matches? matcher str)
```

Attempts to match the entire region against the pattern. Returns true if the patterns matches the string else false.

```
(let [m (regex/matcher #"[0-9]+" "100")]
  (regex/matches? m))
=> true

(let [m (regex/matcher #"[0-9]+" "value: 100")]
  (regex/matches? m))
=> false

(let [m (regex/matcher #"[0-9]+" "")]
  (filter #(regex/matches? m %) ["100" "1a1" "200"]))
=> ("100" "200")
```

SEE ALSO

[match?](#)

Returns true if the string *s* matches the regular expression *regex*.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)[top](#)

regex/pattern

(`regex/pattern s`)

Returns an instance of `java.util.regex.Pattern`.

Patterns are immutable and are safe for use by multiple concurrent threads!

Alternatively regex pattern literals can be used to define a pattern: `"[0-9+]"`

```
"\\d" ;; regex string to match one digit
```

Notice that you have to escape the backslash to get a literal backslash in the string. However, regex pattern literals are smart. They don't need to double escape:

```
"#\\d" ;; regex pattern literal to match one digit
```

JavaDoc: [Pattern](#)

```
(regex/pattern "[0-9]+")
```

```
=> [0-9] +
```

```
(regex/pattern "\\d+")
```

```
=> \\d +
```

```
"#[0-9] +"
```

```
=> [0-9] +
```

```
"#\\d +"
```

```
=> \\d +
```

SEE ALSO

[match?](#)

Returns true if the string `s` matches the regular expression `regex`.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/find](#)

Returns the next regex match or nil if there is no further match. Returns nil if there is no match.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[section](#)[top](#)

regex/pattern?

(`regex/pattern? p`)

Returns `true` if 'p' is a regex pattern else `false`.

```
(regex/pattern? (regex/pattern "[0-9]+"))  
=> true
```

```
(regex/pattern? #"[0-9]+")  
=> true
```

SEE ALSO

[match?](#)

Returns true if the string *s* matches the regular expression *regex*.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches](#)

Returns the matches, if any, for the matcher with the pattern of a string, using `java.util.regex.Matcher.matches()`.

[regex/find](#)

Returns the next regex match or nil if there is no further match. Returns nil if there is no match.

[regex/find-all](#)

Returns all regex matches as list or an empty list if there are no matches.

[section](#)

[top](#)

regex/reset

```
(regex/reset matcher str)
```

Resets the matcher with a new string

```
(do  
  (let [m (regex/matcher #"[0-9]+" "100")]  
    (println (regex/find m))  
    (let [m (regex/reset m "200")]  
      (println (regex/find m)))))  
100  
200  
=> nil
```

SEE ALSO

[match?](#)

Returns true if the string *s* matches the regular expression *regex*.

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[section](#)

[top](#)

release

```
(release lock)
```

Releases a lock.


```
(let [l (lock)]
  (acquire l)
  ;; do something
  (release l))
=> nil
```

SEE ALSO

[lock](#)

Creates a new lock object.

[acquire](#)

Acquires a lock, blocking until the lock is available.

[try-acquire](#)

Acquires a lock within the given timeout time. Without a timeout returns immediately if the lock is not available.

[locked?](#)

Returns true if the lock is in use else false.

[section](#)[top](#)

remove

```
(remove predicate coll)
```

Returns a collection of the items in `coll` for which `(predicate item)` returns logical false.
Returns a transducer when no collection is provided.

```
(remove nil? [1 nil nil 4 5 6])
=> (1 4 5 6)
```

```
(remove even? [1 2 3 4 5 6 7])
=> (1 3 5 7)
```

```
(remove #{3 5} '(1 3 5 7 9))
=> (1 7 9)
```

```
(remove #(= 3 %) '(1 2 3 4 5 6))
=> (1 2 4 5 6)
```

[section](#)[top](#)

remove-formal-type

```
(remove-formal-type object)
```

Removes the *formal type* from a Java object.

This is identical to casting an object back to its real type without knowing its real type.

```
(do
  (let [p0 (. :java.awt.Point :new 0 0)
        p1 (cast :java.lang.Object p0)
        p2 (remove-formal-type p1)]
    (println "p0 ->" (formal-type p0))))
```

```
(println "p1 ->" (formal-type p1))
(println "p2 ->" (formal-type p2)))
p0 -> :java.awt.Point
p1 -> :java.lang.Object
p2 -> :java.awt.Point
=> nil
```

SEE ALSO

[formal-type](#)

Returns the formal type of a Java object.

[cast](#)

Casts a Java object to a specific type

[class](#)

Returns the Java class for the given name. Throws an exception if the class is not found.

[section](#)

[top](#)

remove-tap

```
(remove-tap f)
```

Remove f from the tap set.

```
(do
  (add-tap prn)
  (remove-tap prn))
=> nil
```

SEE ALSO

[add-tap](#)

adds f, a fn of one argument, to the tap set. This function will be called with anything sent via tap>.

[tap>](#)

Sends x to any taps. Will not block. Returns true if there was room in the queue, false if not (x is dropped).

[section](#)

[top](#)

remove-watch

```
(remove-watch ref key)
```

Removes a watch function from an agent/atom reference.

```
(do
  (def x (agent 10))
  (defn watcher [key ref old new]
    (println "watcher: " key))
  (add-watch x :test watcher)
  (remove-watch x :test))
=> nil
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)[top](#)

repeat

```
(repeat x)
(repeat n x)
```

Returns a lazy sequence of x values or a collection with the value x repeated n times.

```
(repeat 3 "hello")
=> ("hello" "hello" "hello")

(repeat 5 [1 2])
=> ([1 2] [1 2] [1 2] [1 2] [1 2])

(repeat ":")
=> (...)

(interleave [:a :b :c] (repeat 100))
=> (:a 100 :b 100 :c 100)
```

SEE ALSO

[repeatedly](#)

Takes a function of no args, presumably with side effects, and returns a collection of n calls to it

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[constantly](#)

Returns a function that takes any number of arguments and returns always the value x.

[section](#)[top](#)

repeatedly

```
(repeatedly n fn)
```

Takes a function of no args, presumably with side effects, and returns a collection of n calls to it

```
(repeatedly 5 #(rand-long 11))
=> (9 0 5 0 5)

;; compare with repeat, which only calls the 'rand-long'
;; function once, repeating the value five times.
(repeat 5 (rand-long 11))
=> (10 10 10 10 10)
```

SEE ALSO

[repeat](#)

Returns a lazy sequence of x values or a collection with the value x repeated n times.

[dotimes](#)

Repeatedly executes body with name bound to integers from 0 through n-1.

[constantly](#)

Returns a function that takes any number of arguments and returns always the value x.

[section](#)[top](#)

repl/ad-hoc?

```
(repl/ad-hoc?)
```

Returns `true` if running a REPL is an *Ad hoc* REPL else `false`.

An *Ad hoc* REPL has very limited capabilities: no installation of 3rd party libraries, no *Maven* tool, no demo scripts!

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/complete-install?](#)

Returns true if running a REPL with a complete installation else false.

[section](#)[top](#)

repl/add-env

```
(repl/add-env name value)
```

Add (or replace) an env var to the REPL's local env file.

The REPL env file ('repl.env' on Unix or 'repl.env.bat' on Windows) is 'sourced' at REPL start time to make the contained vars available as system env vars!

DO NO FORGET to restart the REPL after adding an env var!

Note: This function is only available when called from within a REPL!

Example

1. Add env var:

```
(repl/add-env "DEMO" "100")
```

2. Restart the REPL:

```
venice> !restart
```

3. Test:

```
(system-env "DEMO")
```

```
(repl/add-env "DEMO" "100")
```

SEE ALSO

[system-env](#)

Returns the system env variable with the given name. Returns the default-val if the variable does not exist or it's value is nil.

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/get-env](#)

Returns the value of a REPL local env var.

[repl/cat-env](#)

Returns the content of the REPL's local env file.

[repl/remove-env](#)

Remove an env var to the REPL's local env file.

[section](#)

[top](#)

repl/ansi-terminal?

```
(repl/ansi-terminal?)
```

Returns `true` if the REPL runs in an ANSI terminal else `false`

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[section](#)

[top](#)

repl/cat-env

```
(repl/cat-env)
```

Returns the content of the REPL's local env file.

The REPL env file ('repl.env' on Unix or 'repl.env.bat' on Windows) is 'sourced' at REPL start time to make the contained vars available as system env vars!

Note: This function is only available when called from within a REPL!

```
(println (repl/cat-env))
```

SEE ALSO

[system-env](#)

Returns the system env variable with the given name. Returns the default-val if the variable does not exist or it's value is nil.

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/get-env](#)

Returns the value of a REPL local env var.

[repl/add-env](#)

Add (or replace) an env var to the REPL's local env file.

[repl/remove-env](#)

Remove an env var to the REPL's local env file.

[section](#)[top](#)

repl/color-theme

([repl/color-theme](#))

Returns REPL's color theme (:light, :dark, :none)

([repl/color-theme](#))

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/color-theme!](#)

Set the REPL's color theme (:light, :dark)

[repl/prompt!](#)

Sets the REPL prompt string

[repl/handler!](#)

Sets the REPL command handler

[repl/info](#)

Returns information on the REPL.

[section](#)[top](#)

repl/color-theme!

([repl/color-theme!](#) theme)

Set the REPL's color theme (:light, :dark)

([repl/color-theme!](#))

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/color-theme](#)

Returns REPL's color theme (:light, :dark, :none)

[repl/prompt!](#)

Sets the REPL prompt string

[repl/handler!](#)

Sets the REPL command handler

[repl/info](#)

Returns information on the REPL.

[section](#)[top](#)

repl/complete-install?

```
(repl/complete-install?)
```

Returns `true` if running a REPL with a complete installation else `false`.

An complete REPL supports installation of 3rd party librarie, has *Maven* tool available, and has demo scripts installed!

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/ad-hoc?](#)

Returns true if running a REPL is an Ad hoc REPL else false.

[section](#)[top](#)

repl/get-env

```
(repl/get-env name)
```

Returns the value of a REPL local env var.

The REPL env file ('repl.env' on Unix or 'repl.env.bat' on Windows) is 'sourced' at REPL start time to make the contained vars available as system env vars!

Note: This function is only available when called from within a REPL!

```
(repl/get-env "DEMO")
```

SEE ALSO

[system-env](#)

Returns the system env variable with the given name. Returns the default-val if the variable does not exist or it's value is nil.

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/cat-env](#)

Returns the content of the REPL's local env file.

[repl/add-env](#)

Add (or replace) an env var to the REPL's local env file.

[repl/remove-env](#)

Remove an env var to the REPL's local env file.

[section](#)[top](#)

repl/handler!

```
(repl/handler! f)
```

Sets the REPL command handler

```
(do
  (defn handle-command [cmd]
    ;; run the command 'cmd'
    (println "Demo:" cmd))

  (repl/handler! handle-command))
```

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/prompt!](#)

Sets the REPL prompt string

[repl/color-theme](#)

Returns REPL's color theme (:light, :dark, :none)

[repl/info](#)

Returns information on the REPL.

[section](#)

[top](#)

repl/home-dir

```
(repl/home-dir)
```

Returns the REPL home directory.

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/libs-dir](#)

Returns the REPL libs directory

[repl/tools-dir](#)

Returns the REPL tools directory

[repl/scripts-dir](#)

Returns the REPL scripts directory

[section](#)

[top](#)

repl/info

```
(repl/info)
```

Returns information on the REPL.

Note: This function is only available when called from within a REPL!

E.g.:


```
{ :term-name "JLine terminal"
  :term-type "xterm-256color"
  :term-cols 80
  :term-rows 24
  :term-colors 256
  :term-class :org.repackage.org.jline.terminal.impl.PosixSysTerminal
  :color-mode :light }
```

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/term-rows](#)

Returns number of rows in the REPL terminal.

[repl/term-cols](#)

Returns number of columns in the REPL terminal.

[section](#)

[top](#)

repl/libs-dir

```
(repl/libs-dir)
```

Returns the REPL libs directory

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/tools-dir](#)

Returns the REPL tools directory

[repl/scripts-dir](#)

Returns the REPL scripts directory

[section](#)

[top](#)

repl/prompt!

```
(repl/prompt! s)
```

Sets the REPL prompt string

```
(repl/prompt! "venice> ")
```

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/handler!](#)

Sets the REPL command handler

[repl/color-theme](#)

Returns REPL's color theme (:light, :dark, :none)

[repl/info](#)

Returns information on the REPL.

[section](#)

[top](#)

repl/remove-env

```
(repl/remove-env name)
```

Remove an env var to the REPL's local env file.

The REPL env file ('repl.env' on Unix or 'repl.env.bat' on Windows) is 'sourced' at REPL start time to make the contained vars available as system env vars!

To take a removed env var into effect a whole new REPL has to be started! A simple restart does not work!

Note: This function is only available when called from within a REPL!

```
(repl/remove-env "DEMO")
```

SEE ALSO

[system-env](#)

Returns the system env variable with the given name. Returns the default-val if the variable does not exist or it's value is nil.

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/get-env](#)

Returns the value of a REPL local env var.

[repl/cat-env](#)

Returns the content of the REPL's local env file.

[repl/add-env](#)

Add (or replace) an env var to the REPL's local env file.

[section](#)

[top](#)

repl/scripts-dir

```
(repl/scripts-dir)
```

Returns the REPL scripts directory

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/libs-dir](#)

Returns the REPL libs directory

[repl/tools-dir](#)

Returns the REPL tools directory

[section](#)

[top](#)

repl/term-cols

([repl/term-cols](#))

Returns number of columns in the REPL terminal.

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/term-rows](#)

Returns number of rows in the REPL terminal.

[repl/info](#)

Returns information on the REPL.

[section](#)

[top](#)

repl/term-rows

([repl/term-rows](#))

Returns number of rows in the REPL terminal.

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/term-cols](#)

Returns number of columns in the REPL terminal.

[repl/info](#)

Returns information on the REPL.

[section](#)

[top](#)

repl/tools-dir

([repl/tools-dir](#))

Returns the REPL tools directory

Note: This function is only available when called from within a REPL!

SEE ALSO

[repl?](#)

Returns true if running a REPL else false.

[repl/home-dir](#)

Returns the REPL home directory.

[repl/libs-dir](#)

Returns the REPL libs directory

[repl/scripts-dir](#)

Returns the REPL scripts directory

[section](#)

[top](#)

repl?

```
(repl?)
```

Returns `true` if running a REPL else `false`.

SEE ALSO

[repl/ad-hoc?](#)

Returns true if running a REPL is an Ad hoc REPL else false.

[repl/complete-install?](#)

Returns true if running a REPL with a complete installation else false.

[section](#)

[top](#)

replace

```
(replace smap coll)
```

Given a map of replacement pairs and a collection, returns a collection with any elements that are a key in smap replaced with the corresponding value in smap.

```
(replace {2 :two, 4 :four} [4 2 3 4 5 6 2])  
=> [:four :two 3 :four 5 6 :two]
```

```
(replace {2 :two, 4 :four} #{1 2 3 4 5})  
=> #{1 3 5 :four :two}
```

```
(replace [{:a 10} [:c 30]] {:a 10 :b 20})  
=> {:b 20 :c 30}
```

[section](#)

[top](#)

reset!

```
(reset! box newval)
```

Sets the value of an atom or a volatile to newval without regard for the current value. Returns newval.

```
(do
  (def counter (atom 0))
  (reset! counter 99)
  @counter)
=> 99
```

```
(do
  (def counter (atom 0))
  (reset! counter 99))
=> 99
```

```
(do
  (def counter (volatile 0))
  (reset! counter 99)
  @counter)
=> 99
```

SEE ALSO

[atom](#)

Creates an atom with the initial value x.

[volatile](#)

Creates a volatile with the initial value x

[section](#)

[top](#)

reset-ns-meta!

```
(reset-ns-meta! n datamap)
```

Resets the metadata for a namespace

```
(do
  (ns foo)
  (reset-ns-meta! foo {}))
=> {}
```

```
(do
  (ns foo)
  (def n 'foo)
  (reset-ns-meta! (var-get n) {}))
  (pr-str (ns-meta (var-get n))))
=> "{}"
```

SEE ALSO

[ns-meta](#)

Returns the meta data of the namespace n or nil if n is not an existing namespace

[alter-ns-meta!](#)

Alters the metadata for a namespace. f must be free of side-effects.

[ns](#)

Opens a namespace.

[section](#)

[top](#)

resolve

```
(resolve symbol)
```

Resolves a symbol.

```
(resolve '+)
=> +
```

```
(resolve 'y)
=> nil
```

```
(resolve (symbol "+"))
=> +
```

```
((resolve (symbol "core" "+")) 1 2)
=> 3
```

```
((-> "first" symbol resolve) [1 2 3])
=> 1
```

SEE ALSO

[symbol](#)

Returns a symbol from the given name

[section](#)

[top](#)

rest

```
(rest coll)
```

Returns a possibly empty collection of the items after the first.

```
(rest nil)
=> nil
```

```
(rest [])
=> []
```

```
(rest [1])
=> []
```

```
(rest [1 2 3])
=> [2 3]
```

```
(rest '())
=> ()
```

```
(rest '(1))
=> ()

(rest '(1 2 3))
=> (2 3)

(rest "1234")
=> (#\2 #\3 #\4)
```

SEE ALSO

[str/rest](#)

Returns a possibly empty string of the characters after the first.

[section](#)

[top](#)

restart-agent

```
(restart-agent agent state)
```

When an agent is failed, changes the agent state to new-state and then un-fails the agent so that sends are allowed again.

```
(do
  (def x (agent 100))
  (restart-agent x 200)
  (deref x))
=> 200
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)

[top](#)

reverse

```
(reverse coll)
```

Returns a collection of the items in coll in reverse order.
Returns a stateful transducer when no collection is provided.

```
(reverse [1 2 3 4 5 6])
=> [6 5 4 3 2 1]

(reverse "abcdef")
=> (#\f #\e #\d #\c #\b #\a)
```

SEE ALSO

[str/reverse](#)

Reverses a string

rf-any?

```
(rf-any? pred)
```

Returns a reducing function for a transducer that returns true if the predicate is true for at least one the items, false otherwise.

```
(transduce (filter number?) (rf-any? pos?) [true -1 1 2 false])  
=> true
```

SEE ALSO

[rf-first](#)

Returns a reducing function for a transducer that returns the first item.

[rf-last](#)

Returns a reducing function for a transducer that returns the last item.

[rf-every?](#)

Returns a reducing function for a transducer that returns true if the predicate is true for all the items, false otherwise.

rf-every?

```
(rf-every? pred)
```

Returns a reducing function for a transducer that returns true if the predicate is true for all the items, false otherwise.

```
(transduce (filter number?) (rf-every? pos?) [1 2 3])  
=> true
```

SEE ALSO

[rf-first](#)

Returns a reducing function for a transducer that returns the first item.

[rf-last](#)

Returns a reducing function for a transducer that returns the last item.

[rf-any?](#)

Returns a reducing function for a transducer that returns true if the predicate is true for at least one the items, false otherwise.

rf-first

```
(rf-first)
```

Returns a reducing function for a transducer that returns the first item.

```
(transduce (filter number?) rf-first [false 1 2])  
=> 1
```



```
(transduce identity rf-first [nil 1 2])  
=> nil
```

SEE ALSO

[rf-last](#)

Returns a reducing function for a transducer that returns the last item.

[rf-any?](#)

Returns a reducing function for a transducer that returns true if the predicate is true for at least one the items, false otherwise.

[rf-every?](#)

Returns a reducing function for a transducer that returns true if the predicate is true for all the items, false otherwise.

[section](#)[top](#)

rf-last

```
(rf-last)
```

Returns a reducing function for a transducer that returns the last item.

```
(transduce (filter number?) rf-last [false 1 2])  
=> 2
```

```
(transduce identity rf-last [1 2 1.2])  
=> 1.2
```

SEE ALSO

[rf-first](#)

Returns a reducing function for a transducer that returns the first item.

[rf-any?](#)

Returns a reducing function for a transducer that returns true if the predicate is true for at least one the items, false otherwise.

[rf-every?](#)

Returns a reducing function for a transducer that returns true if the predicate is true for all the items, false otherwise.

[section](#)[top](#)

ring-multipart/multipart-request?

```
(ring-multipart/multipart-request? req)
```

Returns true if the request is a multipart request 'multipart/form-data'

SEE ALSO

[ring-multipart/parts](#)

Returns a list of parts of a multipart HTTP request.

[ring-multipart/parts-delete-all](#)

Safely deletes for all parts the underlying storage for the file items, including deleting any associated temporary disk files.

[section](#)[top](#)

ring-multipart/parts

```
(ring-multipart/parts req)
```

Returns a list of parts of a multipart HTTP request.

A part is map with the fields:

:name	The name of the part
:file-name	The file-name of the part or <i>nil</i> if not available
:size	The size of the file
:content-type	The content type of the part
:headers	A map of part's headers. key: header name, value: list of header values. The header names are mapped to lower case. Use <code>(first ("xxxx" :headers))</code> to get a single value header
:in-stream	The content part as input stream
:delete-fn	A function that deletes the underlying storage for a file item, including deleting any associated temporary disk file.

The part list is empty if the request is not a multipart request.

SEE ALSO

[ring-multipart/multipart-request?](#)

Returns true if the request is a multipart request 'multipart/form-data'

[ring-multipart/parts-delete-all](#)

Safely deletes for all parts the underlying storage for the file items, including deleting any associated temporary disk files.

[section](#)

[top](#)

ring-multipart/parts-delete-all

```
(ring-multipart/parts-delete-all req)
```

Safely deletes for all parts the underlying storage for the file items, including deleting any associated temporary disk files.

Calls the *delete-fn* on every part data map.

SEE ALSO

[ring-multipart/multipart-request?](#)

Returns true if the request is a multipart request 'multipart/form-data'

[ring-multipart/parts](#)

Returns a list of parts of a multipart HTTP request.

[section](#)

[top](#)

ring-mw/mw-debug

```
(ring-mw/mw-debug handler option)
```

Turns handler debug flag on the request on/off and then calls the handler with the modified request.

SEE ALSO

[ring-mw/mw-identity](#)

Identity, does effectively just delegate to the handler

[ring-mw/mw-print-uri](#)

Prints the URI from the request and then calls the handler

[ring-mw/mw-request-counter](#)

Increments the number requests, stores it in the attribute 'request-counter' in the session, and then calls the handler.

[ring-mw/mw-add-session](#)

Adds the session to the request. If a new session is created the given timeout is set as the MaxInactiveInterval. If a timeout is not ...

[ring-mw/mw-dump-request](#)

Dumps the request and then calls the handler.

[ring-mw/mw-dump-response](#)

Calls the handler and the dumps the handler's response.

[section](#)

[top](#)

ring-mw/mw-dump-request

(ring-mw/mw-dump-request handler)

Dumps the request and then calls the handler.

SEE ALSO

[ring-mw/mw-identity](#)

Identity, does effectively just delegate to the handler

[ring-mw/mw-debug](#)

Turns handler debug flag on the request on/off and then calls the handler with the modified request.

[ring-mw/mw-print-uri](#)

Prints the URI from the request and then calls the handler

[ring-mw/mw-request-counter](#)

Increments the number requests, stores it in the attribute 'request-counter' in the session, and then calls the handler.

[ring-mw/mw-add-session](#)

Adds the session to the request. If a new session is created the given timeout is set as the MaxInactiveInterval. If a timeout is not ...

[ring-mw/mw-dump-response](#)

Calls the handler and the dumps the handler's response.

[section](#)

[top](#)

ring-mw/mw-dump-response

(ring-mw/mw-dump-response handler)

Calls the handler and the dumps the handler's response.

SEE ALSO

[ring-mw/mw-identity](#)

Identity, does effectively just delegate to the handler

[ring-mw/mw-debug](#)

Turns handler debug flag on the request on/off and then calls the handler with the modified request.

[ring-mw/mw-print-uri](#)

Prints the URI from the request and then calls the handler

[ring-mw/mw-request-counter](#)

Increments the number requests, stores it in the attribute 'request-counter' in the session, and then calls the handler.

[ring-mw/mw-add-session](#)

Adds the session to the request. If a new session is created the given timeout is set as the MaxInactiveInterval. If a timeout is not ...

[ring-mw/mw-dump-request](#)

Dumps the request and then calls the handler.

[ring-mw/mw-dump-response](#)

Calls the handler and the dumps the handler's response.

[section](#)[top](#)

ring-mw/mw-identity

(ring-mw/mw-identity handler)

Identity, does effectively just delegate to the handler

SEE ALSO

[ring-mw/mw-debug](#)

Turns handler debug flag on the request on/off and then calls the handler with the modified request.

[ring-mw/mw-print-uri](#)

Prints the URI from the request and then calls the handler

[ring-mw/mw-request-counter](#)

Increments the number requests, stores it in the attribute 'request-counter' in the session, and then calls the handler.

[ring-mw/mw-add-session](#)

Adds the session to the request. If a new session is created the given timeout is set as the MaxInactiveInterval. If a timeout is not ...

[ring-mw/mw-dump-request](#)

Dumps the request and then calls the handler.

[ring-mw/mw-dump-response](#)

Calls the handler and the dumps the handler's response.

[section](#)[top](#)

ring-mw/mw-print-uri

(ring-mw/mw-print-uri handler)

Prints the URI from the request and then calls the handler

SEE ALSO

[ring-mw/mw-identity](#)

Identity, does effectively just delegate to the handler

[ring-mw/mw-debug](#)

Turns handler debug flag on the request on/off and then calls the handler with the modified request.

[ring-mw/mw-request-counter](#)

Increments the number requests, stores it in the attribute 'request-counter' in the session, and then calls the handler.

[ring-mw/mw-add-session](#)

Adds the session to the request. If a new session is created the given timeout is set as the MaxInactiveInterval. If a timeout is not ...

[ring-mw/mw-dump-request](#)

Dumps the request and then calls the handler.

[ring-mw/mw-dump-response](#)

Calls the handler and then dumps the handler's response.

[section](#)

[top](#)

ring-mw/mw-request-counter

(`ring-mw/mw-request-counter handler`)

Increments the number requests, stores it in the attribute 'request-counter' in the session, and then calls the handler.

SEE ALSO

[ring-mw/mw-identity](#)

Identity, does effectively just delegate to the handler

[ring-mw/mw-debug](#)

Turns handler debug flag on the request on/off and then calls the handler with the modified request.

[ring-mw/mw-print-uri](#)

Prints the URI from the request and then calls the handler

[ring-mw/mw-add-session](#)

Adds the session to the request. If a new session is created the given timeout is set as the MaxInactiveInterval. If a timeout is not ...

[ring-mw/mw-dump-request](#)

Dumps the request and then calls the handler.

[ring-mw/mw-dump-response](#)

Calls the handler and then dumps the handler's response.

[section](#)

[top](#)

ring-session/session-clear

(`ring-session/session-clear req`)

Removes all attributes from the session

SEE ALSO

[ring-session/session-invalidate](#)

Invalidate the session

[ring-session/session-id](#)

Get the session ID

[ring-session/session-set-value](#)

Sets a value on the session

[ring-session/session-get-value](#)

Get a value from the session

[ring-session/session-remove-value](#)

Remove a value from the session

[ring-session/session-last-access-time](#)

Returns the time (milliseconds since epoch) when this session was last accessed.

[ring-session/session-creation-time](#)

Returns the time (milliseconds since epoch) when this session was created.

[section](#)

[top](#)

ring-session/session-creation-time

```
(ring-session/session-creation-time req)
```

Returns the time (milliseconds since epoch) when this session was created.

SEE ALSO

[ring-session/session-invalidate](#)

Invalidate the session

[ring-session/session-clear](#)

Removes all attributes from the session

[ring-session/session-id](#)

Get the session ID

[ring-session/session-set-value](#)

Sets a value on the session

[ring-session/session-get-value](#)

Get a value from the session

[ring-session/session-remove-value](#)

Remove a value from the session

[ring-session/session-last-access-time](#)

Returns the time (milliseconds since epoch) when this session was last accessed.

[section](#)

[top](#)

ring-session/session-get-value

```
(ring-session/session-get-value req name)
```

Get a value from the session

SEE ALSO

[ring-session/session-invalidate](#)

Invalidate the session

[ring-session/session-clear](#)

Removes all attributes from the session

[ring-session/session-id](#)

Get the session ID

[ring-session/session-set-value](#)

Sets a value on the session

[ring-session/session-remove-value](#)

Remove a value from the session

[ring-session/session-last-access-time](#)

Returns the time (milliseconds since epoch) when this session was last accessed.

[ring-session/session-creation-time](#)

Returns the time (milliseconds since epoch) when this session was created.

[section](#)

[top](#)

ring-session/session-id

(ring-session/session-id req)

Get the session ID

SEE ALSO

[ring-session/session-invalidate](#)

Invalidate the session

[ring-session/session-clear](#)

Removes all attributes from the session

[ring-session/session-set-value](#)

Sets a value on the session

[ring-session/session-get-value](#)

Get a value from the session

[ring-session/session-remove-value](#)

Remove a value from the session

[ring-session/session-last-access-time](#)

Returns the time (milliseconds since epoch) when this session was last accessed.

[ring-session/session-creation-time](#)

Returns the time (milliseconds since epoch) when this session was created.

[section](#)

[top](#)

ring-session/session-invalidate

(ring-session/session-invalidate req)

Invalidate the session

SEE ALSO

[ring-session/session-clear](#)

Removes all attributes from the session

[ring-session/session-id](#)

Get the session ID

[ring-session/session-set-value](#)

Sets a value on the session

[ring-session/session-get-value](#)

Get a value from the session

[ring-session/session-remove-value](#)

Remove a value from the session

[ring-session/session-last-access-time](#)

Returns the time (milliseconds since epoch) when this session was last accessed.

[ring-session/session-creation-time](#)

Returns the time (milliseconds since epoch) when this session was created.

[section](#)

[top](#)

ring-session/session-remove-value

```
(ring-session/session-remove-value req name)
```

Remove a value from the session

SEE ALSO

[ring-session/session-invalidate](#)

Invalidate the session

[ring-session/session-clear](#)

Removes all attributes from the session

[ring-session/session-id](#)

Get the session ID

[ring-session/session-set-value](#)

Sets a value on the session

[ring-session/session-get-value](#)

Get a value from the session

[ring-session/session-last-access-time](#)

Returns the time (milliseconds since epoch) when this session was last accessed.

[ring-session/session-creation-time](#)

Returns the time (milliseconds since epoch) when this session was created.

[section](#)

[top](#)

ring-util/debug?

```
(ring-util/debug? req)
```

Returns true if debugging is turned on else false

[section](#)

[top](#)

ring-util/get-request-header


```
(ring-util/get-request-header req name)
```

Returns the first value of the specified case independent request header name.

If the request did not include a header of the specified name, this method returns `nil`. If there are multiple headers with the same name, this method returns the first header in the request.

[section](#)[top](#)

ring-util/get-request-header-accept-mimetypes

```
(ring-util/get-request-header-accept-mimetypes req)
```

Returns all 'Accept' header mime-types of the request as a set. Strips off the ratings

[section](#)[top](#)

ring-util/get-request-long-parameter

```
(ring-util/get-request-long-parameter request name)
(ring-util/get-request-long-parameter request name value)
```

Returns the first parameter a the multi value request parameter with the name 'name'. Accepts an optional default value.

Converts the parameter value to long. Returns the default value if the parameter is not of type long.

Returns `nil` if the parameter does not exist and a default value is not passed.

[section](#)[top](#)

ring-util/get-request-parameter

```
(ring-util/get-request-parameter req name)
```

Returns the first value of the specified case independent request parameter name.

If the request did not include a parameter of the specified name, this method returns `nil`. If there are multiple headers with the same name, this method returns the first parameter in the request.

[section](#)[top](#)

ring-util/get-request-parameters

```
(ring-util/get-request-parameters req name)
```

Returns all values of the specified case independent request parameter name as a list.

[section](#)[top](#)

ring-util/html-request?

```
(ring-util/html-request? req)
```

Returns true if the request has content type 'text/html'

[section](#)[top](#)

ring-util/json-request?

```
(ring-util/json-request? req)
```

Returns true if the request has content type 'application/json'

[section](#)[top](#)

ring-util/not-found-response

```
(ring-util/not-found-response)  
(ring-util/not-found-response msg)
```

Create a HTTP Not-Found 404 response with content-type text/html.

[section](#)[top](#)

ring-util/parse-charset

```
(ring-util/parse-charset header)
```

Parses the charset from a header value

E.g.: Returns `utf-8` for a content tye header like: `Content-Type: text/html; charset=utf-8`

[section](#)[top](#)

ring-util/redirect

```
(ring-util/redirect request url)
```

Redirect to the given URL.

[section](#)[top](#)

ring/create-servlet

(ring/create-servlet handler)

Create a ring servlet.

SEE ALSO

[ring/match-routes](#)

Compile the routes and return a function that calls the handler matching the URI.

[section](#)

[top](#)

ring/match-routes

(ring/match-routes routes)

Compile the routes and return a function that calls the handler matching the URI.

A route is defined by a HTTP verb, a URI filter and a handle function. If multiple routes match the route with the longest URI filter will be chosen.

```
(def routes [
  [:get    "/"**"                hello-world-handler]
  [:get    "/test/**"            test-handler]
  [:get    "/static/images/*.png" image-handler]

  [:get    "/employees"          get-all-employees]
  [:get    "/employees/:id"       get-employee]
  [:post   "/employees"          create-employee]
  [:put    "/employees/:id"       update-employee]
  [:delete "/employees/:id"       delete-employee] ])
```

Routing URI pattern filters:

- `"/**"`
- `"/app/**"`
- `"/static/images/chart.png"`
- `"/static/images/*.png"`
- `"/static/**/*.png"`

A routing handler is single argument function that receives the request and returns a response.

Handler request:

:server-port	The server port. E.g.: 8080
:server-name	The server name. E.g.: localhost
:remote-addr	The remote address. E.g.: "0:0:0:0:0:0:1"
:uri	The request URI. E.g.: "/employees"
:query-string	The query string
:scheme	The scheme {: <code>http</code> , <code>https</code> }
:request-method	The lower case request method. {: <code>get</code> , <code>post</code> , <code>put</code> , <code>delete</code> , <code>head</code> , <code>options</code> , <code>trace</code> }
:protocol	The protocol. E.g. "HTTP/1.1"
:headers	A map of part's headers. Key: header name, value: list of header values. The header names are mapped to lower case. Use <code>(first ("xxxx" :headers))</code> to get a single value header
:parameters	A name/value map of the request parameters
:cookies	A map of the cookies. Key: cookie name, value: the Java servlet cookie object
:content-type	The content type (may be nil)
:content-length	The content length
:character-encoding	The character encoding

:ssl-client-cert	The client certificate, if available
:parts	A list of parts, empty for non multipart requests
:body	The content part as input stream

Handler response:

```
{ :status 400
  :headers { "Content-Type" "text/plain" }
  :body "Not a json request!" }
```

The `:body` element of a handler response may be a:

- string
- bytebuf
- `:java.io.InputStream`
- `:java.io.File`

Rigging up a Ring WEB App and starting Tomcat:

```
(tc/run-tomcat
 (ring/create-servlet (-> (ring/match-routes routes)
                          (ring-mw/mw-dump-request)
                          (ring-mw/mw-request-counter)
                          (ring-mw/mw-add-session 3600)
                          (ring-mw/mw-print-uri)
                          (ring-mw/mw-debug :on)))
  { :await? false}))
```

SEE ALSO

[ring/create-servlet](#)

Create a ring servlet.

[section](#)

[top](#)

rsa/decrypt

```
(rsa/decrypt data private-key)
```

Decrypts the data with a private key. The data may be a Base64 encoded `string` or a `bytebuf`

Returns a string if data is a Base64 encoded `string` else a `bytebuf`.

```
(let [key-pair (rsa/generate-key-pair)]
  (-> (rsa/encrypt "Hello World" (rsa/public-key key-pair))
      (rsa/decrypt (rsa/private-key key-pair))))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/private-key](#)

Returns the private key of a key pair (`:java.security.KeyPair`)

[rsa/encrypt](#)

Encrypts the data with a public key. The data may be a string or a bytebuf.

[rsa/sign](#)

Signs the data with a private key. The data may be a string or a bytebuf

[rsa/verify](#)

Verifies the integrity of a signature. The data may be a string or a bytebuf

[section](#)[top](#)

rsa/encrypt

```
(rsa/encrypt data public-key)
```

Encrypts the data with a public key. The data may be a `string` or a `bytebuf`.

Returns a Base64 encoded string if data is a `string` else a `bytebuf`.

```
(let [key-pair (rsa/generate-key-pair)]
  (-> (rsa/encrypt "Hello World" (rsa/public-key key-pair))
    (rsa/decrypt (rsa/private-key key-pair))))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/decrypt](#)

Decrypts the data with a private key. The data may be a Base64 encoded string or a bytebuf

[rsa/sign](#)

Signs the data with a private key. The data may be a string or a bytebuf

[rsa/verify](#)

Verifies the integrity of a signature. The data may be a string or a bytebuf

[section](#)[top](#)

rsa/generate-key-pair

```
(rsa/generate-key-pair)
(rsa/generate-key-pair key-size)
```

Generates a RSA public/private key pair with a default key size of 2048 bits

Returns a `:java.security.KeyPair`

```
(rsa/generate-key-pair)
```

```
(rsa/generate-key-pair 3072)
```

SEE ALSO

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[rsa/encrypt](#)

Encrypts the data with a public key. The data may be a string or a bytebuf.

[rsa/decrypt](#)

Decrypts the data with a private key. The data may be a Base64 encoded string or a bytebuf

[rsa/sign](#)

Signs the data with a private key. The data may be a string or a bytebuf

[rsa/verify](#)

Verifies the integrity of a signature. The data may be a string or a bytebuf

[section](#)

[top](#)

rsa/key-pair

```
(rsa/key-pair private-key public-key)
```

Creates a RSA key pair from a public and a private key

Returns a `:java.security.KeyPair`

```
(let [[pub priv] (rsa/keys (rsa/generate-key-pair))]  
  (rsa/key-pair pub priv))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[section](#)

[top](#)

rsa/keys

```
(rsa/keys key-pair)
```

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

```
(let [[pub priv] (rsa/keys (rsa/generate-key-pair))]  
  ;; ...  
)
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[section](#)

[top](#)

rsa/load-key

```
(rsa/load-key pem-file)
```

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

Returns the public or private key.

```
(rsa/load-key "demo-public.pem")
```

```
(rsa/load-key "demo-private.pem")
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[section](#)[top](#)

rsa/load-key-pair

```
(rsa/load-key-pair dir name)
```

Loads a key-pair from public and private key pem files.

Loads the files from:

- {dir}/{name}-public.pem
- {dir}/{name}-private.pem

Returns a :java.security.KeyPair

```
;; PEM files: "./demo-public.pem" and "./demo-private.pem"
(do
  (-> (rsa/generate-key-pair)
    (rsa/save-key-pair "." "demo"))
  (rsa/load-key-pair "." "demo"))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[section](#)[top](#)

rsa/private-key

```
(rsa/private-key key-pair)
```

Returns the private key of a key pair (:java.security.KeyPair)

```
(rsa/private-key key-pair)
```


SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[section](#)

[top](#)

rsa/public-key

([rsa/public-key](#) key-pair)

Returns the public key of a key pair (:java.security.KeyPair)

([rsa/public-key](#) key-pair)

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/save-key-pair](#)

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[rsa/encrypt](#)

Encrypts the data with a public key. The data may be a string or a bytebuf.

[section](#)

[top](#)

rsa/save-key-pair

```
(rsa/save-key-pair key-pair dir name)
```

Saves a key-pair (:java.security.KeyPair) to public and private key pem files.

Creates the files:

- {dir}/{name}-public.pem
- {dir}/{name}-private.pem

```
;; creates two files: "./demo-public.pem" and "./demo-private.pem"
(-> (rsa/generate-key-pair)
    (rsa/save-key-pair "." "demo"))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/key-pair](#)

Creates a RSA key pair from a public and a private key

[rsa/keys](#)

Returns a vector with the public and private key of a key pair (:java.security.KeyPair)

[rsa/load-key-pair](#)

Loads a key-pair from public and private key pem files.

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/load-key](#)

Loads a public or private key from a PEM file. Venice analyzes the PEM file header to detect if the file holds a public or private key.

[rsa/encrypt](#)

Encrypts the data with a public key. The data may be a string or a bytebuf.

[section](#)

[top](#)

rsa/sign

```
(rsa/sign data private-key)
```

Signs the data with a private key. The data may be a `string` or a `bytebuf`

Returns the signature.

```
(let [key-pair (rsa/generate-key-pair)]
  (-> (rsa/sign "Hello World" (rsa/private--key key-pair))
      (rsa/verify "Hello World" (rsa/public-key key-pair))))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/private-key](#)

Returns the private key of a key pair (:java.security.KeyPair)

[rsa/encrypt](#)

Encrypts the data with a public key. The data may be a string or a bytebuf.

[rsa/decrypt](#)

Decrypts the data with a private key. The data may be a Base64 encoded string or a bytebuf

[rsa/verify](#)

Verifies the integrity of a signature. The data may be a string or a bytebuf

[section](#)[top](#)

rsa/verify

```
(rsa/verify signature data public-key)
```

Verifies the integrity of a signature. The data may be a `string` or a `bytebuf`

Returns `true` if the signature is valid else `false` .

```
(let [key-pair (rsa/generate-key-pair)]
  (-> (rsa/sign "Hello World" (rsa/private--key key-pair))
    (rsa/verify "Hello World" (rsa/public-key key-pair))))
```

SEE ALSO

[rsa/generate-key-pair](#)

Generates a RSA public/private key pair with a default key size of 2048 bits

[rsa/public-key](#)

Returns the public key of a key pair (:java.security.KeyPair)

[rsa/encrypt](#)

Encrypts the data with a public key. The data may be a string or a bytebuf.

[rsa/decrypt](#)

Decrypts the data with a private key. The data may be a Base64 encoded string or a bytebuf

[rsa/sign](#)

Signs the data with a private key. The data may be a string or a bytebuf

[section](#)[top](#)

run!

```
(run! f coll)
```

Runs the supplied function, for purposes of side effects, on successive items in the collection. Returns `nil`

```
(run! prn [1 2 3 4])
1
2
3
4
=> nil
```

SEE ALSO

[docoll](#)

Applies f to the items of the collection presumably for side effects. Returns nil.

[mapv](#)

Returns a vector consisting of the result of applying f to the set of first items of each coll, followed by applying f to the set of ...

sandbox/functions

(sandbox/functions group)

Lists the sandboxed functions defined by a sandbox function group.

Groups:

- :io
- :print
- :concurrency
- :java-interop
- :system
- :special-forms
- :unsafe

([sandbox/functions](#) [:print](#))

SEE ALSO

[sandboxed?](#)

Returns true if there is a sandbox other than :AcceptAllInterceptor otherwise false.

sandbox/type

(sandbox/type)

Returns the sandbox type.

Venice sandbox types:

- `:AcceptAllInterceptor` - accepts all (no restrictions)
- `:RejectAllInterceptor` - safe sandbox, rejects access to all I/O functions, system properties, environment vars, extension modules, dynamic code loading, multi-threaded functions (futures, agents, ...), and Java calls
- `:SandboxInterceptor` - customized sandbox

([sandbox/type](#))

=> `:AcceptAllInterceptor`

SEE ALSO

[sandboxed?](#)

Returns true if there is a sandbox other than :AcceptAllInterceptor otherwise false.

sandboxed?

```
(sandboxed?)
```

Returns true if there is a sandbox other than `:AcceptAllInterceptor` otherwise false.

```
(sandboxed?)  
=> false
```

SEE ALSO

[sandbox/type](#)

Returns the sandbox type.

[section](#)

[top](#)

schedule-at-fixed-rate

```
(schedule-at-fixed-rate fn initial-delay period time-unit)
```

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period. Returns a future. `(future? f)`, `(cancel f)`, and `(done? f)` will work on the returned future. Time unit is one of `:milliseconds`, `:seconds`, `:minutes`, `:hours`, or `:days`.

```
(schedule-at-fixed-rate #(println "test") 1 2 :seconds)  
  
(let [s (schedule-at-fixed-rate #(println "test") 1 2 :seconds)]  
  (sleep 16 :seconds)  
  (cancel s))
```

SEE ALSO

[schedule-delay](#)

Creates and executes a one-shot action that becomes enabled after the given delay.

[section](#)

[top](#)

schedule-delay

```
(schedule-delay fn delay time-unit)
```

Creates and executes a one-shot action that becomes enabled after the given delay. Returns a future. `(deref f)`, `(future? f)`, `(cancel f)`, and `(done? f)` will work on the returned future. Time unit is one of `:milliseconds`, `:seconds`, `:minutes`, `:hours`, or `:days`.

```
(schedule-delay (fn [] (println "test"))) 1 :seconds  
  
(deref (schedule-delay (fn [] 100) 2 :seconds))
```

SEE ALSO

[schedule-at-fixed-rate](#)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period.

[section](#)

[top](#)

second

```
(second coll)
```

Returns the second element of coll.

```
(second nil)
=> nil
```

```
(second [])
=> nil
```

```
(second [1 2 3])
=> 2
```

```
(second '())
=> nil
```

```
(second '(1 2 3))
=> 2
```

[section](#)[top](#)

select-keys

```
(select-keys map keyseq)
```

Returns a map containing only those entries in map whose key is in keys

```
(select-keys {:a 1 :b 2} [:a])
=> {:a 1}
```

```
(select-keys {:a 1 :b 2} [:a :c])
=> {:a 1}
```

```
(select-keys {:a 1 :b 2 :c 3} [:a :c])
=> {:a 1 :c 3}
```

SEE ALSO

[keys](#)

Returns a collection of the map's keys.

[entries](#)

Returns a collection of the map's entries.

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[section](#)[top](#)

semver/cmp

```
(semver/cmp a b)
```

Compares versions a and b, returning -1 if a is older than b, 0 if they're the same version, and 1 if a is newer than b.

```
(semver/cmp "1.2.3" "1.5.4")  
=> -1
```

```
(semver/cmp (semver/version "1.2.3") (semver/version "1.5.4"))  
=> -1
```

SEE ALSO

[semver/equal?](#)

Is version a the same as version b?

[semver/newer?](#)

Is version a newer than version b?

[semver/older?](#)

Is version a older than version b?

[section](#)

[top](#)

semver/equal?

```
(semver/equal? a b)
```

Is version a the same as version b?

```
(semver/newer? "1.2.3" "1.2.3")  
=> false
```

```
(semver/newer? (semver/version "1.2.3") (semver/version "1.2.3"))  
=> false
```

SEE ALSO

[semver/newer?](#)

Is version a newer than version b?

[semver/older?](#)

Is version a older than version b?

[semver/cmp](#)

Compares versions a and b, returning -1 if a is older than b, 0 if they're the same version, and 1 if a is newer than b.

[section](#)

[top](#)

semver/newer?

```
(semver/newer? a b)
```

Is version a newer than version b?

```
(semver/newer? "1.5.4" "1.2.3")  
=> true
```

```
(semver/newer? (semver/version "1.5.4") (semver/version "1.2.3"))
=> true
```

SEE ALSO

[semver/older?](#)

Is version a older than version b?

[semver/equal?](#)

Is version a the same as version b?

[semver/cmp](#)

Compares versions a and b, returning -1 if a is older than b, 0 if they're the same version, and 1 if a is newer than b.

[section](#)

[top](#)

semver/older?

```
(semver/older? a b)
```

Is version a older than version b?

```
(semver/newer? "1.2.3" "1.5.4")
=> false
```

```
(semver/newer? (semver/version "1.2.3") (semver/version "1.5.4"))
=> false
```

SEE ALSO

[semver/newer?](#)

Is version a newer than version b?

[semver/equal?](#)

Is version a the same as version b?

[semver/cmp](#)

Compares versions a and b, returning -1 if a is older than b, 0 if they're the same version, and 1 if a is newer than b.

[section](#)

[top](#)

semver/parse

```
(semver/parse s)
```

Parses string 's' into a semantic version map.

Semantic versioning format:

standard

version:	1.0.0
pre-release:	1.0.0-beta
meta data:	1.0.0-beta+001

with revision

version:	1.0.0.0
pre-release:	1.0.0.0-beta
meta data:	1.0.0.0-beta+001


```
E.g.: { :major 1, :minor 3, :patch 5 }
      { :major 1, :minor 3, :patch 5 :pre-release "beta"}
      { :major 1, :minor 3, :patch 5 :pre-release "beta"}
      { :major 1, :minor 3, :patch 5 :pre-release "beta" :meta "001"}
```

```
(semver/parse "1.2.3")
=> {:patch 3 :meta-data nil :minor 2 :major 1 :revision nil :pre-release nil}

(semver/parse "1.2.3-beta")
=> {:patch 3 :meta-data nil :minor 2 :major 1 :revision nil :pre-release "beta"}

(semver/parse "1.2.3-beta+001")
=> {:patch 3 :meta-data "001" :minor 2 :major 1 :revision nil :pre-release "beta"}
```

SEE ALSO

[semver/version](#)

If 'v' is a valid version map, returns the map. Otherwise, it'll attempt to parse 'v' and return a version map.

[semver/valid-format?](#)

Checks the string 's' for semantic versioning formatting

[section](#)

[top](#)

semver/valid-format?

```
(semver/valid-format? s)
```

Checks the string 's' for semantic versioning formatting

```
(semver/valid-format? "1.2.3")
=> true
```

SEE ALSO

[semver/parse](#)

Parses string 's' into a semantic version map.

[semver/valid?](#)

Checks if the supplied version map is valid regarding semantic versioning or not.

[section](#)

[top](#)

semver/valid?

```
(semver/valid? v)
```

Checks if the supplied version map is valid regarding semantic versioning or not.

```
(semver/valid? (semver/parse "1.2.3"))
=> true
```

SEE ALSO

[semver/parse](#)

Parses string 's' into a semantic version map.

[semver/valid?](#)

Checks if the supplied version map is valid regarding semantic versioning or not.

[section](#)[top](#)

semver/version

```
(semver/version o)
```

If 'o' is a valid version map, returns the map. Otherwise, it'll attempt to parse 'o' and return a version map.

```
(semver/version "1.2.3")  
=> {:patch 3 :meta-data nil :minor 2 :major 1 :revision nil :pre-release nil}
```

SEE ALSO

[semver/parse](#)

Parses string 's' into a semantic version map.

[section](#)[top](#)

send

```
(send agent action-fn args)
```

Dispatch an action to an agent. Returns the agent immediately.

The state of the agent will be set to the value of:

```
(apply action-fn state-of-agent args)
```

```
(do  
  (def x (agent 100))  
  (send x + 5)  
  (send x (partial + 7))  
  (sleep 100)  
  (deref x))  
=> 112
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[send-off](#)

Dispatch a potentially blocking action to an agent. Returns the agent immediately.

[section](#)[top](#)

send-off

```
(send-off agent fn args)
```

Dispatch a potentially blocking action to an agent. Returns the agent immediately.

The state of the agent will be set to the value of:

```
(apply action-fn state-of-agent args)
```

```
(do
  (def x (agent 100))
  (send-off x + 5)
  (send-off x (partial + 7))
  (sleep 100)
  (deref x))
=> 112
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[send](#)

Dispatch an action to an agent. Returns the agent immediately.

[section](#)[top](#)

seq

```
(seq coll)
```

Returns a sequence on the collection.

If the collection is empty, returns nil!! `(seq nil)` returns nil.

seq also works on strings and converts Java streams to lists.

```
(seq nil)
=> nil
```

```
(seq [])
=> nil
```

```
(seq [1 2 3])
=> (1 2 3)
```

```
(seq '(1 2 3))
=> (1 2 3)
```

```
(seq {:a 1 :b 2})
=> ([:a 1] [:b 2])
```

```
(seq #{1 2})
=> (1 2)
```

```
(seq "abcd")
=> (#\a #\b #\c #\d)
```

```
(flatten (seq {:a 1 :b 2}))
=> (:a 1 :b 2)
```

[section](#)[top](#)

sequential?

```
(sequential? coll)
```

Returns true if coll is a sequential collection

```
(sequential? '(1))  
=> true
```

```
(sequential? [1])  
=> true
```

```
(sequential? {:a 1})  
=> false
```

```
(sequential? nil)  
=> false
```

```
(sequential? "abc")  
=> false
```

[section](#)

[top](#)

server-side-events/parse

```
(parse s)
```

Parses a server side event in string representation to a map.

```
(do  
  (load-module :server-side-events ['server-side-events :as 'sse])  
  (-> (sse/render { :id "100"  
                   :event "scores"  
                   :data ["100" "200"] } ))  
      (sse/parse)))  
=> {:data ["100" "200"] :event "scores" :id "100"}
```

SEE ALSO

[server-side-events/render](#)

Renders a server side event to a string.

[section](#)

[top](#)

server-side-events/read-event

```
(read-event rd)
```

Read a single event from a `java.io.BufferedReader`.

Returns the event or `nil` if the underlying stream has been closed.

```
(do
  (load-module :server-side-events ['server-side-events :as 'sse])

  (defn sample-events []
    (str (sse/render { :id "100" :event "scores" :data ["100"] } )
         (sse/render { :id "101" :event "scores" :data ["101"] } )
         (sse/render { :id "102" :event "scores" :data ["102"] } )))

  (try-with [is (io/string-in-stream (sample-events))
              rd (io/wrap-is-with-buffered-reader is :utf-8)]
    (sse/read-event rd)))
=> {:data ["100"] :event "scores" :id "100"}
```

SEE ALSO

[server-side-events/read-events](#)

Reads multiple events from a `:java.io.BufferedReader`.

[section](#)

[top](#)

server-side-events/read-events

```
(read-events rd limit)
```

Reads multiple events from a `:java.io.BufferedReader`.

Returns a list of events. Stops reading events if the limit is reached or the underlying stream has been closed.

```
(do
  (load-module :server-side-events ['server-side-events :as 'sse])

  (defn sample-events []
    (str (sse/render { :id "100" :event "scores" :data ["100"] } )
         (sse/render { :id "101" :event "scores" :data ["101"] } )
         (sse/render { :id "102" :event "scores" :data ["102"] } )
         (sse/render { :id "103" :event "scores" :data ["103"] } )))

  (try-with [is (io/string-in-stream (sample-events))
              rd (io/wrap-is-with-buffered-reader is :utf-8)]
    (sse/read-events rd 3)))
=> [{:data ["100"] :event "scores" :id "100"} {:data ["101"] :event "scores" :id "101"} {:data ["102"] :event "scores" :id "102"}]
```

SEE ALSO

[server-side-events/read-event](#)

Read a single event from a `:java.io.BufferedReader`.

[section](#)

[top](#)

server-side-events/render

```
(render event)
```

Renders a server side event to a string.

Returns the event as string or `nil` if the event is `nil` or all its fields are empty or `nil`,

Note: SSE is restricted to transporting UTF-8 messages.

The event is a map. E.g. :

```
{ :id      "1"
  :event   "score"
  :data    [ "GOAL Liverpool 1 - 1 Arsenal"
            "GOAL Manchester United 3 - 3 Manchester City" ] }
```

with the text representation

```
id: 1\n
event: score\n
data: GOAL Liverpool 1 - 1 Arsenal\n
data: GOAL Manchester United 3 - 3 Manchester City\n\n
```

The event fields :id, :event, and :data must not contain newline, carriage return, backspace, or formfeed characters!

A HTTP request to initiate SSE streaming looks like:

```
GET /api/v1/live-scores
Accept: text/event-stream
Cache-Control: no-cache
Connection: keep-alive
```

```
(do
  (load-module :server-side-events ['server-side-events :as 'sse])
  (sse/render { :id "100"
               :event "scores"
               :data ["100" "200"] } ))
=> "id: 100\r\nevent: scores\r\ndata: 100\r\ndata: 200\r\n\r\n"
```

SEE ALSO

[server-side-events/parse](#)

Parses a server side event in string representation to a map.

[section](#)

[top](#)

service

(service name method & args)

Calls a service with the specified name from the Venice's service registry.

Venice's service registry is used with application scripting scenarios where multiple external services must be made available to Venice. E.g.: the service registry can be used to register an application's *Spring Framework* services and make them discoverable by a Venice script.

Example:

```
Venice venice = new Venice();

venice.getServiceRegistry()
    .register("Calculator", new Calculator())
    .registerServiceDiscovery(new TestServiceDiscovery());

long r = (Long)venice.eval("(service :Calculator :multiply 10 20)");
venice.eval("(service :Logger :log (version))");
```

while `Calculator` and `TestServiceDiscovery` are defined as:

```
public static class TestServiceDiscovery implements IServiceDiscovery {
    @Override public Object lookup(final String name) {
        if (name == null) {
            throw new IllegalArgumentException("A service name must not be null");
        }
    }
}
```

```

        else if (name.equals("Logger")) {
            return logger;
        }
        else {
            throw new VncException("Service " + name + " is not registered");
        }
    }
    private final Logger logger = new Logger();
}

public class Calculator {
    public long multiply(long v1, long v2) {
        return v1 * v2;
    }
}

public static class Logger {
    public void log(String message) {
        System.out.println(message);
    }
}

```

```
(service :UserService :find "Smith" "John")
```

SEE ALSO

[service?](#)

Returns true if the named service exists otherwise false

[section](#)

[top](#)

service?

```
(service? name)
```

Returns true if the named service exists otherwise false

```
(service? :UserService
```

SEE ALSO

[service](#)

Calls a service with the specified name from the Venice's service registry.

[section](#)

[top](#)

set

```
(set & items)
```

Creates a new set containing the items.

```
(set)
=> #[]
```

```
(set nil)
=> #{nil}

(set 1)
=> #{1}

(set 1 2 3)
=> #{1 2 3}

(set [1 2] 3)
=> #{[1 2] 3}
```

[section](#)[top](#)

set!

```
(set! var-symbol expr)
```

Sets a global or thread-local variable to the value of the expression.

```
(do
  (def x 10)
  (set! x 20)
  x)
=> 20

(do
  (def-dynamic x 100)
  (set! x 200)
  x)
=> 200

(do
  (def-dynamic x 100)
  (without-str
    (print x)
    (binding [x 200]
      (print (str "-" x))
      (set! x (inc x))
      (print (str "-" x)))
    (print (str "-" x))))
=> "100-200-201-100"
```

SEE ALSO

[def](#)

Creates a global variable.

[def-dynamic](#)

Creates a dynamic variable that starts off as a global variable and can be bound with 'binding' to a new value on the local thread.

[section](#)[top](#)

set-error-handler!


```
(set-error-handler! agent handler-fn)
```

Sets the error-handler of an agent to `handler-fn` . If an action being run by the agent throws an exception `handler-fn` will be called with two arguments: the agent and the exception.

```
(do
  (def x (agent 100))
  (defn err-handler-fn [ag ex]
    (println "error occurred: "
              (:message ex)
              " and we still have value"
              @ag))
  (set-error-handler! x err-handler-fn)
  (send x (fn [n] (/ n 0))))
=> (agent :value 100)
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[agent-error-mode](#)

Returns the agent's error mode

[agent-error](#)

Returns the exception thrown during an asynchronous action of the agent if the agent is failed. Returns nil if the agent is not failed.

[section](#)[top](#)

set?

```
(set? obj)
```

Returns true if obj is a set

```
(set? (set 1))
=> true
```

[section](#)[top](#)

sgn

```
(sgn x)
```

sgn function for a number.

```
-1 if x < 0
 0 if x = 0
 1 if x > 0
```

```
(sgn -10)
=> -1
```

```
(sgn 0)
=> 0
```

```
(sgn 10)
=> 1

(sgn -10I)
=> -1

(sgn -10.1)
=> -1

(sgn -10.12M)
=> -1
```

SEE ALSO

abs

Returns the absolute value of the number

negate

Negates x

section

top

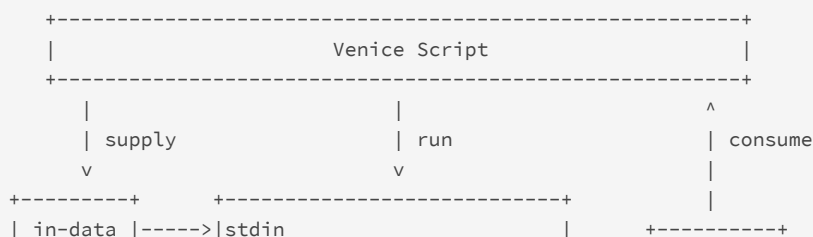
sh

```
(sh & args)
```

Launches a new sub-process.

Options:

:in	may be given followed by input source as InputStream, Reader, File, ByteBuf, or String, to be fed to the sub-process's stdin.
:in-enc	option may be given followed by a String, used as a character encoding name (for example "UTF-8" or "ISO-8859-1") to convert the input string specified by the :in option to the sub-process's stdin. Defaults to "UTF-8". If the :in option provides a byte array, then the bytes are passed unencoded, and this option is ignored.
:out-enc	option may be given followed by :bytes or a String. If a String is given, it will be used as a character encoding name (for example "UTF-8" or "ISO-8859-1") to convert the sub-process's stdout to a String which is returned. If :bytes is given, the sub-process's stdout will be stored in a Bytebuf and returned. Defaults to UTF-8.
:out-fn	a function with a single string argument that receives line by line from the process' stdout. If passed the :out value in the return map will be empty.
:err-fn	a function with a single string argument that receives line by line from the process' stderr. If passed the :err value in the return map will be empty.
:env	override the process env with a map.
:dir	override the process dir with a String or java.io.File.
:throw-ex	If true throw an exception if the exit code is not equal to zero, if false returns the exit code. Defaults to false. It's recommended to use <pre>(with-sh-throw (sh "ls" "-l"))</pre> instead.
:timeout	A timeout in milliseconds



	SHELL PROCESS	stdout	stderr	out-fn	err-fn
+	-----+	----->	----->	-----+	-----+

You can bind `:env`, `:dir` for multiple operations using `with-sh-env` or `with-sh-dir`. `with-sh-throw` is binds `:throw-ex` as `true`.

`sh` returns a map of:

```
:exit => sub-process's exit code
:out  => sub-process's stdout (as Bytebuf or String)
:err  => sub-process's stderr (String via platform default encoding)
```

E.g.:

```
(sh "uname" "-r")
=> {:err "" :out "20.5.0\n" :exit 0}
```

```
(println (sh "ls" "-l"))

(println (sh "ls" "-l" "/tmp"))

(println (sh "sed" "s/[aeiou]/oo/g" :in "hello there\n"))

(println (sh "cat" :in "x\u25bax\n"))

(println (sh "echo" "x\u25bax"))

(println (sh "/bin/sh" "-c" "ls -l"))

(sh "ls" "-l" :out-fn println)

(sh "ls" "-l" :out-fn println :err-fn println)

;; background process
(println (sh "/bin/sh" "-c" "sleep 30 >/dev/null 2>&1 &"))

(println (sh "/bin/sh" "-c" "nohup sleep 30 >/dev/null 2>&1 &"))

;; asynchronously slurping stdout and stderr
(sh "/bin/sh"
  "-c" "for i in {1..5}; do sleep 1; echo \"Hello $i\"; done"
  :out-fn println
  :err-fn println)

;; asynchronously slurping stdout and stderr with a timeout
(sh "/bin/sh"
  "-c" "for i in {1..5}; do sleep 1; echo \"Hello $i\"; done"
  :out-fn println
  :err-fn println
  :timeout 2500)

;; reads 4 single-byte chars
(println (sh "echo" "x\u25bax" :out-enc "ISO-8859-1"))

;; reads binary file into bytes[]
(println (sh "cat" "birds.jpg" :out-enc :bytes))

;; working directory
(println (with-sh-dir "/tmp" (sh "ls" "-l") (sh "pwd")))

(println (sh "pwd" :dir "/tmp"))
```

```
;; throw an exception if the shell's subprocess exit code is not equal to 0
(println (with-sh-throw (sh "ls" "-l"))))

(println (sh "ls" "-l" :throw-ex true))

;; windows
(println (sh "cmd" "/c dir 1>&2"))
```

SEE ALSO

[with-sh-throw](#)

Shell commands executed within a with-sh-throw context throw an exception if the spawned shell process returns an exit code other than 0.

[with-sh-dir](#)

Sets the directory for use with sh, see sh for details.

[with-sh-env](#)

Sets the environment for use with sh.

[section](#)

[top](#)

sh/alive?

```
(sh/alive? pid)
```

Returns true if the process represented by the PID is alive otherwise false.

Runs internally the Unix command `ps -p {pid}` to check if there is a process with the given pid.

Note: This function is available for Linux and MacOS only!

```
(sh/alive? "2345")
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[sh/kill](#)

Kills a process.

[sh/killall](#)

Kills all processes with the given name.

[sh/pgrep](#)

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

[sh/pkill](#)

Sends a signal to all processes with the given name.

[sh/pargs](#)

Returns a process' command line.

[section](#)

[top](#)

sh/kill

```
(sh/kill pid)
(sh/kill pid signal)
```

Kills a process.

The signal to be sent is one of {:[sighup](#), [:sigint](#), [:sigquit](#), [:sigkill](#), [:sigterm](#)}. If no signal is specified, the [:sigterm](#) signal is sent.

Runs the Unix command: `kill -signal {pid}`

Throws an exception if the process does not exist or cannot be killed!

Note: This function is available for Linux and MacOS only!

```
(sh/kill "2345")
```

```
(sh/kill "2345" :sigkill)
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[sh/alive?](#)

Returns true if the process represented by the PID is alive otherwise false.

[sh/killall](#)

Kills all processes with the given name.

[sh/pgrep](#)

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

[sh/pkill](#)

Sends a signal to all processes with the given name.

[sh/pargs](#)

Returns a process' command line.

[section](#)

[top](#)

sh/killall

```
(sh/killall name)
```

```
(sh/killall name signal)
```

Kills all processes with the given name.

The signal to be sent is one of {:[sighup](#), [:sigint](#), [:sigquit](#), [:sigkill](#), [:sigterm](#)}. If no signal is specified, the [:sigterm](#) signal is sent.

Runs the Unix command: `killall -signal -e {name}`

Note: This function is available for Linux and MacOS only!

```
(sh/killall "clamd")
```

```
(sh/killall "clamd" :sigkill)
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[sh/alive?](#)

Returns true if the process represented by the PID is alive otherwise false.

[sh/kill](#)

Kills a process.

[sh/pgrep](#)

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

[sh/pkill](#)

Sends a signal to all processes with the given name.

[sh/pargs](#)

Returns a process' command line.

[section](#)[top](#)

sh/load-pid

```
(sh/load-pid pid-file)
```

Load a process PID from a PID file.

Returns the PID or `nil` if the file does not exist or is empty

```
(sh/load-pid "/data/scan.pid")
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[sh/alive?](#)

Returns true if the process represented by the PID is alive otherwise false.

[sh/kill](#)

Kills a process.

[sh/killall](#)

Kills all processes with the given name.

[sh/pgrep](#)

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

[sh/pkill](#)

Sends a signal to all processes with the given name.

[sh/pargs](#)

Returns a process' command line.

[section](#)[top](#)

sh/open

```
(sh/open f)
```

Opens a *file* or an *URL* with the associated platform specific application.

Uses the OS commands:

- *MacOS*: `/usr/bin/open f`
- *Windows*: `cmd /C start f`
- *Linux*: `/usr/bin/xdg-open f`

Note: `sh/open` can only be run from a REPL!

```
(sh/open "sample.pdf")
```

```
(sh/open "https://github.com/jlangch/venice")
```

[section](#)[top](#)

sh/pargs

```
(sh/pargs pid)
```

```
(sh/pargs :parse pid)
```

Returns a process' command line.

Without the `:parse` option returns the command line as a string or `nil` if there is no process matching the PID. With the `:parse` option returns a list of the command line arguments or an empty list the process does not exist.

Runs the Unix command: `ps -p {pid} -ww -o args` to get the command line.

Note: This function is available for Linux and MacOS only!

```
(sh/pargs "1234")
```

```
(sh/pargs :parse "1234")
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[sh/alive?](#)

Returns true if the process represented by the PID is alive otherwise false.

[sh/kill](#)

Kills a process.

[sh/killall](#)

Kills all processes with the given name.

[sh/pgrep](#)

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

[sh/pkill](#)

Sends a signal to all processes with the given name.

[section](#)[top](#)

sh/pgrep

```
(sh/pgrep name)
```

Returns a list of all pids for a process with the given name or `nil` if there are no processes matching the name.

Runs the Unix command: `pgrep -x {name}`

Note: This function is available for Linux and MacOS only!

```
(sh/pgrep "clamd")
```

SEE ALSO

sh

Launches a new sub-process.

sh/alive?

Returns true if the process represented by the PID is alive otherwise false.

sh/kill

Kills a process.

sh/killall

Kills all processes with the given name.

sh/pkill

Sends a signal to all processes with the given name.

sh/pargs

Returns a process' command line.

[section](#)

[top](#)

sh/pkill

```
(sh/pkill name)
```

```
(sh/pkill name signal)
```

Sends a signal to all processes with the given name.

The signal to be sent is one of {:`sig`hup, :`sig`int, :`sig`quit, :`sig`kill, :`sig`term}.If no signal is specified, the :`sig`term signal is sent.

Runs the Unix command: `pkill -signal -x {name}`

Note: This function is available for Linux and MacOS only!

```
(sh/pkill "clamd")
```

```
(sh/pkill "clamd" :sigkill)
```

SEE ALSO

sh

Launches a new sub-process.

sh/alive?

Returns true if the process represented by the PID is alive otherwise false.

sh/kill

Kills a process.

sh/killall

Kills all processes with the given name.

sh/pgrep

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

sh/pargs

Returns a process' command line.

[section](#)

[top](#)

sh/pwd

```
(sh/pwd)
```


Returns the current working directory.

Note:

You can't change the current working directory of the Java VM but if you were to launch another process using (sh & args) you can specify the working directory for the new spawned process.

(sh/pwd)

SEE ALSO

[sh](#)

Launches a new sub-process.

[section](#)

[top](#)

sh/which

(sh/which program)

Locates a program's file in the user's path.

Returns the program's full path or `nil` if not found.

E.g. (sh/which "ps") returns "/bin/ps" for MacOSX and "/usr/bin/ps" for Linux RHEL variants.

Note: This function is available for Linux and MacOS only!

(sh/which "ps")

SEE ALSO

[sh](#)

Launches a new sub-process.

[sh/alive?](#)

Returns true if the process represented by the PID is alive otherwise false.

[sh/kill](#)

Kills a process.

[sh/killall](#)

Kills all processes with the given name.

[sh/pgrep](#)

Returns a list of all pids for a process with the given name or nil if there are no processes matching the name.

[sh/pargs](#)

Returns a process' command line.

[section](#)

[top](#)

shell/alive?

(alive? pid)

(alive? process-handle)

Returns true if the process represented by a PID or a process handle is alive otherwise false.

Requires Java 9+.

([shell/alive?](#) [4556](#))

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of :java.lang.ProcessHandle for the processes.

[section](#)

[top](#)

shell/descendant-processes

(descendant-processes pid)
(descendant-processes process-handle)

Returns the descendants (:java.lang.ProcessHandle) of a process represented by a PID or a process handle.

Requires Java 9+.

([shell/descendant-processes](#) [4556](#))

```
(->> (shell/current-process)
      (shell/descendant-processes)
      (map shell/process-info))
```

SEE ALSO

[shell/process-info](#)

Returns the process info for a process represented by a PID or a process handle.

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[section](#)

[top](#)

shell/diff

(diff file1 file2)

Compare two files and print the differences.

```
(diff "/tmp/x.txt" "/tmp/y.txt")
```

[section](#)

[top](#)

shell/exists-process-with-pid?

(exists-process-with-pid? pid)

Returns `true` if a process with given pid exists else `false`.

Runs a "ps -p {pid}" command under the hood.

Is available on Mac OSX or Linux only! Runs on Java 8+.

```
(shell/exists-process-with-pid? 12345)
```

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/kill](#)

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process ...

[shell/wait-for-process-exit](#)

Waits until the process with the pid exits. Waits max timeout seconds. Returns nil if the process exits before reaching the timeout, ...

[section](#)

[top](#)

shell/get-process-pids

```
(get-process-pids process-name)
```

Returns a list of the pids of all processes running with the given name.

Runs a "pgrep {name}" command under the hood.

Is available on Mac OSX or Linux only! Runs on Java 8+.

```
(shell/get-process-pids "clamscan")
```

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/kill](#)

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process ...

[shell/wait-for-process-exit](#)

Waits until the process with the pid exits. Waits max timeout seconds. Returns nil if the process exits before reaching the timeout, ...

[section](#)

[top](#)

shell/kill

```
(kill pid)
(kill process-handle)
```

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process does not exist. Accepts a PID or a process handle (:java.lang.ProcessHandle).

Requires Java 9+.

```
(shell/kill 4556)
```

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/kill-forcibly](#)

Requests the process to be killed forcibly. Returns true if the process is killed and false if the process stays alive. Returns nil ...

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of :java.lang.ProcessHandle for the processes.

[section](#)[top](#)

shell/kill-forcibly

(kill-forcibly pid)

(kill-forcibly process-handle)

Requests the process to be killed forcibly. Returns true if the process is killed and false if the process stays alive. Returns nil if the process does not exist. Accepts a PID or a process handle (:java.lang.ProcessHandle).

Requires Java 9+.

([shell/kill-forcibly](#) 4556)

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/kill](#)

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process ...

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of :java.lang.ProcessHandle for the processes.

[section](#)[top](#)

shell/nice

(nice n)

(nice pid n)

The *Nice* command in linux allows users to prioritize process execution. The nice value determines the priority of a process, with lower values indicating higher priority.

A niceness of -20 is the lowest niceness, or highest priority. The default niceness for processes is inherited from its parent process and is usually 0.

Systems have diverged on what priority is the lowest. Linux systems document a niceness of 19 as the lowest priority, BSD systems document 20 as the lowest priority. In both cases, the "lowest" priority is documented as running only when nothing else wants to.

(nice n) change the nice value of this process

(nice pid n) change the nice value of this process with the given pid

Command line equivalent: `nice -n -10 tar cvzf archive.tgz largefile`

Requires Java 9+.

([shell/nice](#) -10)

([shell/nice](#) ([shell/pid](#)) -10)

SEE ALSO

shell/pid

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[section](#)[top](#)

shell/open

(open url)

Opens a file or an url with the associated platform specific application.

```
(shell/open "img.png")
```

```
(shell/open "https://www.heise.de/")
```

SEE ALSO

[shell/open-macos-app](#)

Opens a Mac OSX app.

[section](#)[top](#)

shell/open-macos-app

(open-macos-app name & args)

Opens a Mac OSX app.

```
(shell/open-macos-app "Calendar")
```

```
(shell/open-macos-app "Maps")
```

```
(shell/open-macos-app "TextEdit" "example.txt")
```

SEE ALSO

[shell/open](#)

Opens a file or an url with the associated platform specific application.

[section](#)[top](#)

shell/parent-process

(parent-process pid)

(parent-process process-handle)

Returns the parent (:java.lang.ProcessHandle) of a process represented by a PID or a process handle.

Requires Java 9+.

```
(shell/parent-process 4556)
```

```
(->> (shell/current-process)
      (shell/parent-process)
      (shell/process-info))
```

SEE ALSO

[shell/process-info](#)

Returns the process info for a process represented by a PID or a process handle.

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of :java.lang.ProcessHandle for the processes.

[section](#)

[top](#)

shell/pid

```
(pid)
(pid process-handle)
```

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for the process represented by the handle.

Requires Java 9+.

```
(shell/pid)
```

SEE ALSO

[shell/process-handle](#)

Returns the process handle (:java.lang.ProcessHandle) for a PID or nil if there is no process.

[shell/process-info](#)

Returns the process info for a process represented by a PID or a process handle.

[shell/alive?](#)

Returns true if the process represented by a PID or a process handle is alive otherwise false.

[shell/kill](#)

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process ...

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of :java.lang.ProcessHandle for the processes.

[section](#)

[top](#)

shell/process-handle

```
(process-handle pid)
```

Returns the process handle (:java.lang.ProcessHandle) for a PID or nil if there is no process.

Requires Java 9+.

```
(shell/process-handle 4556)
```

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/alive?](#)

Returns true if the process represented by a PID or a process handle is alive otherwise false.

[shell/process-info](#)

Returns the process info for a process represented by a PID or a process handle.

[shell/kill](#)

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process ...

[section](#)[top](#)

shell/process-handle?

```
(process-handle? p)
```

Returns true if p is a process handle (:java.lang.ProcessHandle).

Requires Java 9+.

[section](#)[top](#)

shell/process-info

```
(process-info pid)
(process-info process-handle)
```

Returns the process info for a process represented by a PID or a process handle.

The process info is a map with the keys:

:pid	the PID
:alive	true if the process is alive else false
:arguments	the list of strings of the arguments of the process
:command	the executable pathname of the process
:command-line	the command line of the process
:start-time	the start time of the process
:total-cpu-millis	the total cputime accumulated of the process
:user	the user of the process.

Requires Java 9+.

```
(shell/process-info 4556)
```

```
;; find the PID of the ArangoDB process
;; like: pgrep -lf ArangoDB3 | cut -d ' ' -f 1
(->> (shell/processes)
      (map shell/process-info)
      (filter #(str/contains? (:command-line %) "ArangoDB3"))
      (map :pid))
```

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (:java.lang.ProcessHandle) returns the PID for ...

[shell/process-handle](#)

Returns the process handle (:java.lang.ProcessHandle) for a PID or nil if there is no process.

[section](#)

[top](#)

shell/processes

(processes)

Returns a snapshot of all processes visible to the current process. Returns a list of :java.lang.ProcessHandle for the processes.

Requires Java 9+.

([shell/processes](#))

```
;; find the PID of the ArangoDB process
;; like: pgrep -lf ArangoDB3 | cut -d ' ' -f 1
(->> (shell/processes)
      (map shell/process-info)
      (filter #(str/contains? (:command-line %) "ArangoDB3"))
      (map :pid))
```

SEE ALSO

[shell/processes-info](#)

Returns a snapshot of all processes visible to the current process. Returns a list of process infos for the processes.

[section](#)

[top](#)

shell/processes-info

(processes-info)

Returns a snapshot of all processes visible to the current process. Returns a list of process infos for the processes.

The process info is a map with the keys:

:pid	the PID
:alive	true if the process is alive else false
:arguments	the list of strings of the arguments of the process
:command	the executable pathname of the process
:command-line	the command line of the process
:start-time	the start time of the process
:total-cpu-millis	the total cputime accumulated of the process
:user	the user of the process.

Requires Java 9+.

([shell/processes-info](#))

```
;; find the PID of the ArangoDB process
;; like: pgrep -lf ArangoDB3 | cut -d ' ' -f 1
(->> (shell/processes-info)
      (filter #(str/contains? (:command-line %) "ArangoDB3"))
      (map :pid))
```


SEE ALSO

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of `:java.lang.ProcessHandle` for the processes.

[section](#)

[top](#)

shell/wait-for-process-exit

```
(wait-for-process-exit pid timeout)
(wait-for-process-exit process-handle timeout)
```

Waits until the process with the pid exits. Waits max timeout seconds. Returns nil if the process exits before reaching the timeout, else the pid is returned. Accepts a PID or a process handle (`:java.lang.ProcessHandle`).

Requires Java 9+.

```
(shell/wait-for-process-exit 12345 20)
```

SEE ALSO

[shell/pid](#)

Without argument returns the PID (type long) of this process. With a process-handle (`:java.lang.ProcessHandle`) returns the PID for ...

[shell/kill](#)

Requests the process to be killed. Returns true if the process is killed and false if the process stays alive. Returns nil if the process ...

[shell/processes](#)

Returns a snapshot of all processes visible to the current process. Returns a list of `:java.lang.ProcessHandle` for the processes.

[section](#)

[top](#)

shuffle

```
(shuffle coll)
```

Returns a collection of the items in coll in random order.

```
(shuffle '(1 2 3 4 5 6))
=> (4 3 2 6 5 1)
```

```
(shuffle [1 2 3 4 5 6])
=> [4 2 3 1 6 5]
```

```
(shuffle "abcdef")
=> (#\c #\a #\e #\d #\f #\b)
```

[section](#)

[top](#)

shutdown-agents

```
(shutdown-agents)
```

Initiates a shutdown of the thread pools that back the agent system. Running actions will complete, but no new actions will be accepted

```
(do
  (def x1 (agent 100))
  (def x2 (agent 100))
  (shutdown-agents))
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)[top](#)

shutdown-agents?

```
(shutdown-agents?)
```

Returns true if the thread-pool that backs the agents is shut down

```
(do
  (def x1 (agent 100))
  (def x2 (agent 100))
  (shutdown-agents)
  (sleep 300)
  (shutdown-agents?))
```

SEE ALSO

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)[top](#)

shutdown-hook

```
(shutdown-hook f)
```

Registers the function `f` as a JVM shutdown hook.

Shutdown hooks can be tested in a REPL:

- start a REPL
- run `(shutdown-hook (fn [] (println "SHUTDOWN") (sleep 3000)))`
- exit the REPL with `!exit`

The sandbox is active within the shutdown hook:

- start a REPL
- run `!sandbox customized`
- run `!sandbox add-rule blacklist:venice:func:+`
- run `(shutdown-hook (fn [] (try (+ 1 2) (catch :SecurityException ex (println ex) (sleep 3000)))))`
- exit the REPL with `!exit`

```
(shutdown-hook (fn [] (println "shutdown")))
```

sleep

```
(sleep n)
(sleep n time-unit)
```

Sleep for the time n. The default time unit is milliseconds.

Time unit is one of :milliseconds, :seconds, :minutes, :hours, or :days or their abbreviations :msec, :ms, :sec, :s, :min, :hr, :h, :d.

```
(sleep 30)
=> nil

(sleep 30 :milliseconds)
=> nil

(sleep 30 :msec)
=> nil

(sleep 5 :seconds)
=> nil

(sleep 5 :sec)
=> nil
```

some

```
(some pred coll)
```

Returns the first logical true value of (pred x) for any x in coll, else nil.

Stops processing the collection if the first value is found that meets the predicate.

```
(some even? '(1 2 3 4))
=> true

(some even? '(1 3 5 7))
=> nil

(some #{5} [1 2 3 4 5])
=> 5

(some #(== 5 %) [1 2 3 4 5])
=> true

(some #(if (even? %) %) [1 2 3 4])
=> 2
```

some->

```
(some-> expr & forms)
```

When expr is not nil, threads it into the first form (via `->`), and when that result is not nil, through the next etc.

```
(some-> {:y 3 :x 5}
      :y
      (- 2))
=> 1
```

```
(some-> {:y 3 :x 5}
      :z
      (- 2))
=> nil
```

SEE ALSO

[some->>](#)

When expr is not nil, threads it into the first form (via `->>`), and when that result is not nil, through the next etc.

[section](#)

[top](#)

some->>

```
(some->> expr & forms)
```

When expr is not nil, threads it into the first form (via `->>`), and when that result is not nil, through the next etc.

```
(some->> {:y 3 :x 5}
      :y
      (- 2))
=> -1
```

```
(some->> {:y 3 :x 5}
      :z
      (- 2))
=> nil
```

SEE ALSO

[some->](#)

When expr is not nil, threads it into the first form (via `->`), and when that result is not nil, through the next etc.

[section](#)

[top](#)

some?

```
(some? x)
```

Returns true if x is not nil, false otherwise

```
(some? nil)
=> false
```

```
(some? 0)
=> true

(some? 4.0)
=> true

(some? false)
=> true

(some? [])
=> true

(some? {})
=> true
```

SEE ALSO

[nil?](#)
Returns true if x is nil, false otherwise

[section](#) [top](#)

sort

```
(sort coll)
(sort comparefn coll)
```

Returns a sorted list of the items in coll. If no compare function comparefn is supplied, uses the natural compare. The compare function takes two arguments and returns -1, 0, or 1

```
(sort [3 2 5 4 1 6])
=> [1 2 3 4 5 6]

(sort compare [3 2 5 4 1 6])
=> [1 2 3 4 5 6]

; reversed
(sort (comp - compare) [3 2 5 4 1 6])
=> [6 5 4 3 2 1]

(sort {:c 3 :a 1 :b 2})
=> ([:a 1] [:b 2] [:c 3])
```

SEE ALSO

[sort-by](#)
Returns a sorted sequence of the items in coll, where the sort order is determined by comparing (keyfn item). If no comparator is supplied, ...

[section](#) [top](#)

sort-by

```
(sort-by keyfn coll)
(sort-by keyfn compfn coll)
```

Returns a sorted sequence of the items in coll, where the sort order is determined by comparing (keyfn item). If no comparator is supplied, uses compare.

To sort by multiple values use `juxt`, see the examples below.

```
(sort-by :id [{:id 2 :name "Smith"} {:id 1 :name "Jones"} ])
=> [{:name "Jones" :id 1} {:name "Smith" :id 2}]

(sort-by count ["aaa" "bb" "c"])
=> ["c" "bb" "aaa"]

; reversed
(sort-by count (comp - compare) ["aaa" "bb" "c"])
=> ["aaa" "bb" "c"]

(sort-by first [[1 2] [3 4] [2 3]])
=> [[1 2] [2 3] [3 4]]

; sort tuples by first value, and where first value is equal,
; sort by second value
(sort-by (juxt first second) [[3 2] [1 3] [3 1] [1 2]])
=> [[1 2] [1 3] [3 1] [3 2]]

; reversed
(sort-by first (comp - compare) [[1 2] [3 4] [2 3]])
=> [[3 4] [2 3] [1 2]]

(sort-by :rank [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 1} {:rank 2} {:rank 3}]

; reversed
(sort-by :rank (comp - compare) [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 3} {:rank 2} {:rank 1}]

; sort entries in a map by value
(sort-by val {:foo 7, :bar 3, :baz 5})
=> ([:bar 3] [:baz 5] [:foo 7])

; sort by :foo, and where :foo is equal, sort by :bar
(do
  (def x [ {:foo 2 :bar 11}
            {:foo 1 :bar 99}
            {:foo 2 :bar 55}
            {:foo 1 :bar 77} ])
  (sort-by (juxt :foo :bar) x))
=> [{:foo 1 :bar 77} {:foo 1 :bar 99} {:foo 2 :bar 11} {:foo 2 :bar 55}]

; sort by a given key order
(do
  (def x [ {:foo 2 :bar 11}
            {:foo 1 :bar 99}
            {:foo 2 :bar 55}
            {:foo 1 :bar 77} ])
  (def order [55 77 99 11])
  (sort-by #((into {} (map-indexed (fn [i e] [e i]) order)) (:bar %))
    x))
=> [{:foo 2 :bar 55} {:foo 1 :bar 77} {:foo 1 :bar 99} {:foo 2 :bar 11}]
```

SEE ALSO

[sort](#)

Returns a sorted list of the items in coll. If no compare function comparefn is supplied, uses the natural compare. The compare function ...

[section](#)

[top](#)

sorted

```
(sorted cmp coll)
```

Returns a sorted collection using the compare function cmp. The compare function takes two arguments and returns -1, 0, or 1.
Returns a stateful transducer when no collection is provided.

```
(sorted compare [4 2 1 5 6 3])  
=> [1 2 3 4 5 6]
```

```
(sorted (comp (partial * -1) compare) [4 2 1 5 6 3])  
=> [6 5 4 3 2 1]
```

[section](#)

[top](#)

sorted-map

```
(sorted-map & keyvals)  
(sorted-map map)
```

Creates a new sorted map containing the items.

```
(sorted-map :a 1 :b 2)  
=> {:a 1 :b 2}
```

```
(sorted-map (hash-map :a 1 :b 2))  
=> {:a 1 :b 2}
```

[section](#)

[top](#)

sorted-map?

```
(sorted-map? obj)
```

Returns true if obj is a sorted map

```
(sorted-map? (sorted-map :a 1 :b 2))  
=> true
```

[section](#)

[top](#)

sorted-set

```
(sorted-set & items)
```

Creates a new sorted-set containing the items.

```
(sorted-set)
=> #{}

(sorted-set nil)
=> #{nil}

(sorted-set 1)
=> #{1}

(sorted-set 6 2 4)
=> #{2 4 6}

(str (sorted-set [2 3] [1 2]))
=> "#{[1 2] [2 3]}"
```

[section](#)

[top](#)

sorted-set?

```
(sorted-set? obj)
```

Returns true if obj is a sorted-set

```
(sorted-set? (sorted-set 1))
=> true
```

[section](#)

[top](#)

split-at

```
(split-at n coll)
```

Returns a vector of [(take n coll) (drop n coll)]

```
(split-at 2 [1 2 3 4 5])
=> [(1 2) (3 4 5)]

(split-at 3 [1 2])
=> [(1 2) ()]
```

[section](#)

[top](#)

split-with

```
(split-with pred coll)
```

Splits the collection at the first false/nil predicate result in a vector with two lists


```
(split-with odd? [1 3 5 6 7 9])  
=> [(1 3 5) (6 7 9)]
```

```
(split-with odd? [1 3 5])  
=> [(1 3 5) ()]
```

```
(split-with odd? [2 4 6])  
=> [()] (2 4 6)]
```

[section](#)[top](#)

sqrt

```
(sqrt x)
```

Square root of x

```
(sqrt 10)  
=> 3.1622776601683795
```

```
(sqrt 10I)  
=> 3.1622776601683795
```

```
(sqrt 10.23)  
=> 3.1984371183438953
```

```
(sqrt 10.23M)  
=> 3.198437118343895324557024650857783854007720947265625M
```

```
(sqrt 10N)  
=> 3.162277660168379522787063251598738133907318115234375M
```

SEE ALSO

[square](#)

Square of x

[section](#)[top](#)

square

```
(square x)
```

Square of x

```
(square 10)  
=> 100
```

```
(square 10I)  
=> 100I
```

```
(square 10.23)  
=> 104.6529
```

```
(square 10.23M)
=> 104.6529M
```

SEE ALSO

[sqrt](#)

Square root of x

[section](#)

[top](#)

stack

```
(stack)
```

Creates a new mutable threadsafe stack.

```
(let [s (stack)]
  (push! s 1)
  (push! s 2)
  (push! s 3))
=> (3 2 1)
```

SEE ALSO

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[pop!](#)

Pops an item from a stack.

[push!](#)

Pushes an item to a stack.

[empty](#)

Returns an empty collection of the same category as coll, or nil if coll is nil. If the collection is mutable clears the collection ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[into!](#)

Adds all of the items of 'from' conjoined to the mutable 'to' collection

[conj!](#)

Returns a new mutable collection with the x, xs 'added'. (conj! nil item) returns (item) and (conj! item) returns item.

[stack?](#)

Returns true if coll is a stack

[section](#)

[top](#)

stack?

```
(stack? coll)
```

Returns true if coll is a stack

```
(stack? (stack))  
=> true
```

[section](#)[top](#)

stacktrace

```
(stacktrace ex)
```

Returns the stacktrace of a java exception

```
(println (stacktrace (. :VncException :new (str "test"))))
```

[section](#)[top](#)

stopwatch/copy

```
(copy sw)
```

Copies a stop watch with its internal state.

Returns the copied stop watch.

```
(do  
  (load-module :stopwatch ['stopwatch :as 'sw])  
  (let [sw-1 (sw/create)]  
    (sleep 500)  
    (let [sw-2 (sw/copy sw-1)]  
      (sw/stop sw-1)  
      (sw/stop sw-2)  
      (println (str "sw-1: " (sw/elapsed sw-1) "ms"))  
      (println (str "sw-2: " (sw/elapsed sw-2) "ms")))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/create

(create)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

Returns the created stop watch.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms"))))
```

SEE ALSO

[stopwatch/create-time-limit](#)

Create a new stop watch with a time limit and implicitly starts it by setting the start time as now. The stop watch has a resolution ...

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)

[top](#)

stopwatch/create-time-limit

(create-time-limit unit n)

Create a new stop watch with a time limit and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

This stop watch has the same functionality as a standard stop watch. But in addition the time limit can be checked with `expired?`

Returns the created stop watch.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create-time-limit :seconds 3)]
    (sleep 2000)
    (println "Expired:" (sw/expired? sw))
    (sleep 2000)
    (println "Expired:" (sw/expired? sw)))
```

```
(sw/stop sw)
(println (str (sw/elapsed sw) "ms"))))
```

SEE ALSO

[stopwatch/expired?](#)

Returns true if the stop watch has expired, meaning it exceeded the time limit, else false.

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/elapsed

```
(elapsed sw)
(elapsed sw time-unit)
```

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

time-unit:

- `:milliseconds`
- `:seconds`
- `:minutes`
- `:hours`
- `:days`

Returns the elapsed time.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms"))
    (println (str (sw/elapsed sw :milliseconds) "ms"))))

(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
```

```
(sleep 500)
(sw/stop sw)
(println (str (sw/elapsed sw) "ms"))
(sleep 300)
(sw/stop sw)
(println (str (sw/elapsed sw) "ms"))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/elapsed-formatted

```
(elapsed-formatted sw)
```

Returns the elapsed time (now - last stop time) formatted.

Format:

* 245ms - for elapsed times < 1s

- 45s 245ms - for elapsed times < 1h
- 10m 45s - for elapsed times >= 1h

Returns the formatted elapsed time.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/stop sw)
    (println (sw/elapsed-formatted sw)))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[section](#)[top](#)

stopwatch/expired?

```
(expired? sw)
```

Returns `true` if the stop watch has expired, meaning it exceeded the time limit, else `false` .

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create-time-limit :seconds 3)]
    (sleep 2000)
    (println "Expired:" (sw/expired? sw))
    (sleep 2000)
    (println "Expired:" (sw/expired? sw))
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms")))))
```

SEE ALSO

[stopwatch/create-time-limit](#)

Create a new stop watch with a time limit and impicetely starts it by setting the start time as now. The stop watch has a resolution ...

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/resume

```
(resume sw)
```

Resumes a stop watch. Sets the start time to now.

Returns the stop watch.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/resume sw)
    (sleep 300)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms")))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/split

```
(split sw)
(split sw time-unit)
```

Returns the split time (now - start time) in milli seconds or in the requested time unit.

time-unit:

- `:milliseconds`
- `:seconds`
- `:minutes`
- `:hours`
- `:days`

Returns the split time.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (println (str (sw/split sw) "ms"))
    (sleep 300)
    (println (str (sw/split sw) "ms"))
    (println (str (sw/split sw :milliseconds) "ms")))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/start

```
(start sw)
```

Restart a stop watch by resetting the start time to now.

Returns the stop watch.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/start sw)
    (sleep 200)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms"))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/stop](#)

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)[top](#)

stopwatch/stop

```
(stop sw)
```

Stops a stop watch. Remembers the elapsed time and sets the start time to now.

Returns the stop watch.

```
(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms"))))

(do
  (load-module :stopwatch ['stopwatch :as 'sw])
  (let [sw (sw/create)]
    (sleep 500)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms")))
    (sleep 300)
    (sw/stop sw)
    (println (str (sw/elapsed sw) "ms"))))
```

SEE ALSO

[stopwatch/create](#)

Create a new stop watch and implicitly starts it by setting the start time as now. The stop watch has a resolution of milliseconds.

[stopwatch/copy](#)

Copies a stop watch with its internal state.

[stopwatch/start](#)

Restart a stop watch by resetting the start time to now.

[stopwatch/resume](#)

Resumes a stop watch. Sets the start time to now.

[stopwatch/split](#)

Returns the split time (now - start time) in milli seconds or in the requested time unit.

[stopwatch/elapsed](#)

Returns the elapsed time (now - last stop time) in milliseconds or in the requested time unit.

[stopwatch/elapsed-formatted](#)

Returns the elapsed time (now - last stop time) formatted.

[section](#)

[top](#)

str

```
(str & xs)
```

With no args, returns the empty string. With one arg x, returns x.toString(). (str nil) returns the empty string. With more than one arg, returns the concatenation of the str values of the args.

```
(str)
=> ""
```

```
(str 1 2 3)
=> "123"
```

```
(str 1I)
=> "1"
```

```
(str 3.1415927M)
=> "3.1415927"

(str +)
=> "+"

(str [1 2 3])
=> "[1 2 3]"

(str "total " 100)
=> "total 100"

(str #\h #\i)
=> "hi"
```

SEE ALSO

[pr-str](#)

With no args, returns the empty string. With one arg x, returns x.toString(). With more than one arg, returns the concatenation of ...

[section](#)[top](#)

str/align

```
(str/align width align overflow text)
```

Aligns a text within a string of width characters.

align: :left, :center, :right

overflow: :newline :clip-left, :clip-right, :ellipsis-left, :ellipsis-right

```
(str/align 6 :left :clip-right "abc")
=> "abc   "
```

```
(str/align 6 :center :clip-right "abc")
=> "  abc  "
```

```
(str/align 6 :right :clip-right "abc")
=> "    abc"
```

```
(str/align 6 :left :clip-left "abcdefgh")
=> "cdefgh"
```

```
(str/align 6 :left :ellipsis-left "abcdefgh")
=> "...defgh"
```

```
(str/align 6 :left :ellipsis-right "abcdefgh")
=> "abcde..."
```

SEE ALSO

[str/trim-to-nil](#)

Trims leading and trailing whitespaces from s. Returns nil if the resulting string is empty

[str/trim-left](#)

Trims leading whitespaces from s.

[str/trim-right](#)

Trims trailing whitespaces from s.

[section](#)

[top](#)

str/blank?

```
(str/blank? s)
```

True if s is nil, empty, or contains only whitespace.

```
(str/blank? nil)
=> true
```

```
(str/blank? "")
=> true
```

```
(str/blank? " ")
=> true
```

```
(str/blank? "abc")
=> false
```

SEE ALSO

[str/not-blank?](#)

True if s contains at least one non whitespace char.

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[not-empty?](#)

Returns true if x is not empty. Accepts strings, collections and bytebufs.

[nil?](#)

Returns true if x is nil, false otherwise

[section](#)

[top](#)

str/butlast

```
(str/butlast s)
```

Returns a possibly empty string of the characters without the last.

```
(str/butlast "abcdef")
=> "abcde"
```

[section](#)

[top](#)

str/butnlast

```
(str/butnlast s n)
```

Returns a possibly empty string of the characters without the n last characters.

```
(str/butnlast "abcdef" 3)
=> "abc"
```

[section](#)

[top](#)

str/bytebuf-to-hex

```
(str/bytebuf-to-hex data)
(str/bytebuf-to-hex data :upper)
```

Converts byte data to a hex string using the hexadecimal digits: 0123456789abcdef .
If the :upper options is passed the hex digits 0123456789ABCDEF are used.

```
(str/bytebuf-to-hex (bytebuf [0 1 2 3 4 5 6]))
=> "00010203040506"

(str/bytebuf-to-hex (bytebuf [202 254]) :upper)
=> "CAFE"
```

[section](#)

[top](#)

str/char?

```
(str/char? s)
```

Returns true if s is a char or a single char string.

```
(str/char? "x")
=> true
```

```
(str/char? #\x)
=> true
```

[section](#)

[top](#)

str/chars

```
(str/chars s)
```

Converts a string to a char list.

```
(str/chars "abcdef")
=> (#\a #\b #\c #\d #\e #\f)
```

```
(str/join (str/chars "abcdef"))
=> "abcdef"
```

[section](#)

[top](#)

str/contains?

```
(str/contains? s substr)
```

True if s contains with substr.

```
(str/contains? "abc" "ab")  
=> true
```

```
(str/contains? "abc" #\b)  
=> true
```

[section](#)[top](#)

str/cr-lf

```
(str/cr-lf s mode)
```

Convert a text to use LF or CR-LF.

```
(str/cr-lf "line1  
line2  
line3" :cr-lf)
```

```
(str/cr-lf "line1  
line2  
line3" :lf)
```

[section](#)[top](#)

str/crlf-to-lf

```
(str/crlf-to-lf s)
```

Converts CR-LF to LF

```
(str/crlf-to-lf nil)
```

```
(str/crlf-to-lf "100  
200  
")
```

SEE ALSO

[str/cr-lf](#)

Convert a text to use LF or CR-LF.

[section](#)[top](#)

str/decode-base64

```
(str/decode-base64 s)
(str/decode-base64 s schema)
```

Base64 decode.

Decoding schema:

- `:Standard` (RFC4648)
- `:UrlSafe` (RFC4648_URLSAFE)

Standard Base64 uses: [A-Z], [a-z], [0-9], '+', '/' URL safe Base64 uses: [A-Z], [a-z], [0-9], '-', '_'

```
(-> (str/encode-base64 (bytebuf [0 1 2 3 4 5 6]))
    (str/decode-base64))
=> [0 1 2 3 4 5 6]

(-> (str/encode-base64 (bytebuf [0 1 2 3 4 5 6]) :Standard)
    (str/decode-base64 :Standard))
=> [0 1 2 3 4 5 6]

(-> (str/encode-base64 (bytebuf [0 1 2 3 4 5 6]) :UrlSafe)
    (str/decode-base64 :UrlSafe))
=> [0 1 2 3 4 5 6]
```

[section](#)

[top](#)

str/decode-url

```
(str/decode-url s)
```

URL decode.

```
(str/decode-url "The+string+%C3%BC%40foo-bar")
=> "The string ü@foo-bar"
```

[section](#)

[top](#)

str/digit?

```
(str/digit? s)
```

True if `s` is a char and the char is a digit.

Defined by Java `Character.isDigit(ch)`.

```
(str/digit? #\8)
=> true
```

```
(str/digit? "8")
=> false
```

SEE ALSO

[str/letter?](#)

True if `s` is a char and the char is a letter.

[str/hexdigit?](#)

True if s is a char and the char is a hex digit.

[section](#)[top](#)

str/double-quote

```
(str/double-quote str)
```

Double quotes a string.

```
(str/double-quote "abc")  
=> "\"abc\""
```

```
(str/double-quote "")  
=> "\"\""
```

[section](#)[top](#)

str/double-quoted?

```
(str/double-quoted? str)
```

Returns true if the string is double quoted.

```
(str/double-quoted? "\"abc\"")  
=> true
```

[section](#)[top](#)

str/double-unquote

```
(str/double-unquote str)
```

Unquotes a double quoted string.

```
(str/double-unquote "\"abc\"")  
=> "abc"
```

```
(str/double-unquote "\"\"")  
=> ""
```

```
(str/double-unquote nil)  
=> nil
```

[section](#)[top](#)

str/encode-base64


```
(str/encode-base64 data)
(str/encode-base64 data schema)
```

Base64 encode.

Encoding schema:

- `:Standard` (RFC4648)
- `:UrlSafe` (RFC4648_URLSAFE)

Standard Base64 uses: [A-Z], [a-z], [0-9], '+', '/' URL safe Base64 uses: [A-Z], [a-z], [0-9], '-', '_'

```
(str/encode-base64 (bytebuf [0 1 2 3 4 5 6]))
=> "AAECAwQFBg=="
```

```
(str/encode-base64 (bytebuf [0 1 2 3 4 5 6]) :Standard)
=> "AAECAwQFBg=="
```

```
(str/encode-base64 (bytebuf [0 1 2 3 4 5 6]) :UrlSafe)
=> "AAECAwQFBg=="
```

[section](#) [top](#)

str/encode-url

```
(str/encode-url s)
```

URL encode.

```
(str/encode-url "The string ü@foo-bar")
=> "The+string+%C3%BC%40foo-bar"
```

[section](#) [top](#)

str/ends-with?

```
(str/ends-with? s substr)
```

True if s ends with substr.

```
(str/ends-with? "abc" "bc")
=> true
```

[section](#) [top](#)

str/equals-ignore-case?

```
(str/equals-ignore-case? s1 s2)
```

Compares two strings ignoring case. True if both are equal.

```
(str/equals-ignore-case? "abc" "abC")  
=> true
```

[section](#)[top](#)

str/escape-html

```
(str/escape-html s)
```

HTML escape. Escapes `&`, `<`, `>`, `"`, `'`, and the non blocking space `U+00A0`

```
(str/escape-html "1 2 3 & < > \" ' \u00A0")  
=> "1 2 3 &amp; &lt; &gt; &quot; &apos; "
```

[section](#)[top](#)

str/escape-xml

```
(str/escape-xml s)
```

XML escape. Escapes `&`, `<`, `>`, `"`, `'`

```
(str/escape-xml "1 2 3 & < > \" '")  
=> "1 2 3 &amp; &lt; &gt; &quot; &apos;"
```

[section](#)[top](#)

str/expand

```
(str/expand s len fill mode*)
```

Expands a string to the max lenght len. Fills up with the fillstring if the string needs to be expanded. The fill string is added to the start or end of the string depending on the mode `:start`, `:end`. The mode defaults to `:end`

```
(str/expand "abcdefghij" 8 ".")  
=> "abcdefghij"
```

```
(str/expand "abcdefghij" 20 ".")  
=> "abcdefghij....."
```

```
(str/expand "abcdefghij" 20 "." :start)  
=> ".....abcdefghij"
```

```
(str/expand "abcdefghij" 20 "." :end)  
=> "abcdefghij....."
```

```
(str/expand "abcdefghij" 30 "1234" :start)  
=> "12341234123412341234abcdefghij"
```

```
(str/expand "abcdefghij" 30 "1234" :end)  
=> "abcdefghij12341234123412341234"
```

str/format

```
(str/format format args*)  
(str/format locale format args*)
```

Returns a formatted string using the specified format string and arguments.
Venice uses the Java format syntax.

JavaDoc: [Format Syntax](#)

```
(str/format "value: %.4f" 1.45)  
=> "value: 1.4500"  
  
(str/format (. :java.util.Locale :new "de" "DE") "value: %.4f" 1.45)  
=> "value: 1,4500"  
  
(str/format (. :java.util.Locale :GERMANY) "value: %.4f" 1.45)  
=> "value: 1,4500"  
  
(str/format (. :java.util.Locale :new "de" "CH") "value: %,d" 2345000)  
=> "value: 2'345'000"  
  
(str/format [ "de" ] "value: %,2f" 100000.45)  
=> "value: 100.000,45"  
  
(str/format [ "de" "DE" ] "value: %,2f" 100000.45)  
=> "value: 100.000,45"  
  
(str/format [ "de" "CH" ] "value: %,2f" 100000.45)  
=> "value: 100'000.45"  
  
(str/format [ "en" "US" ] "value: %,2f" 100000.45)  
=> "value: 100,000.45"  
  
(str/format [ "de" "DE" ] "value: %,d" 2345000)  
=> "value: 2.345.000"
```

str/format-bytebuf

```
(str/format-bytebuf data delimiter & options)
```

Formats a bytebuffer.

Options

:prefix0x prefix with 0x

```
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) nil)  
=> "002243E2FF"
```

```
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) "")  
=> "002243E2FF"  
  
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) ", ")  
=> "00, 22, 43, E2, FF"  
  
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) ", " :prefix0x)  
=> "0x00, 0x22, 0x43, 0xE2, 0xFF"
```

[section](#)[top](#)

str/hex-to-bytebuf

```
(str/hex-to-bytebuf hex)
```

Converts a hex string to a bytebuf

```
(str/hex-to-bytebuf "005E4AFF")  
=> [0 94 74 255]  
  
(str/hex-to-bytebuf "005e4aff")  
=> [0 94 74 255]
```

[section](#)[top](#)

str/hexdigit?

```
(str/hexdigit? s)
```

True if s is a char and the char is a hex digit.

```
(str/hexdigit? #\8)  
=> true  
  
(str/hexdigit? #\a)  
=> true  
  
(str/hexdigit? #\A)  
=> true  
  
(str/hexdigit? #\Y)  
=> false
```

[section](#)[top](#)

str/index-of

```
(str/index-of s value)  
(str/index-of s value from-index)
```

Return index of value (string or char) in s, optionally searching forward from from-index. Return nil if value not found.

```
(str/index-of "abcdefabc" "ab")  
=> 0
```

SEE ALSO

[str/index-of-char](#)

Return index of the first char of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return nil ...

[str/index-of-not-char](#)

Return index of the first char not of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return ...

[str/last-index-of](#)

Return last index of value (string or char) in s, optionally searching backward from from-index. Return nil if value not found.

[section](#)

[top](#)

str/index-of-char

```
(str/index-of-char s chars)  
(str/index-of-char s chars from-index)
```

Return index of the first char of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return nil if value not found.

```
(str/index-of-char "-+-123+-123" "012")  
=> 3
```

```
(str/index-of-char "-+-123+-123" [#\0 #\1 #\2])  
=> 3
```

```
(str/index-of-char "-+-123+-123" "012" 7)  
=> 9
```

SEE ALSO

[str/index-of-not-char](#)

Return index of the first char not of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return ...

[str/index-of](#)

Return index of value (string or char) in s, optionally searching forward from from-index. Return nil if value not found.

[str/last-index-of](#)

Return last index of value (string or char) in s, optionally searching backward from from-index. Return nil if value not found.

[section](#)

[top](#)

str/index-of-not-char

```
(str/index-of-not-char s chars)  
(str/index-of-not-char s chars from-index)
```

Return index of the first char not of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return nil if value not found.

```
(str/index-of-not-char "-+-123-+-123" "-+")
=> 3

(str/index-of-not-char "-+-123-+-123" [#\ - #\+])
=> 3

(str/index-of-not-char "-+-123-+-123" "-+" 7)
=> 9
```

SEE ALSO

[str/index-of-char](#)

Return index of the first char of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return nil ...

[str/index-of](#)

Return index of value (string or char) in s, optionally searching forward from from-index. Return nil if value not found.

[str/last-index-of](#)

Return last index of value (string or char) in s, optionally searching backward from from-index. Return nil if value not found.

[section](#)

[top](#)

str/join

```
(str/join coll)
(str/join separator coll)
```

Joins all elements in coll separated by an optional separator.

```
(str/join [1 2 3])
=> "123"

(str/join "-" [1 2 3])
=> "1-2-3"

(str/join "-" [(char "a") 1 "xyz" 2.56M])
=> "a-1-xyz-2.56"

(str/join #\ - [1 2 3])
=> "1-2-3"
```

[section](#)

[top](#)

str/last-index-of

```
(str/last-index-of s value)
(str/last-index-of s value from-index)
```

Return last index of value (string or char) in s, optionally searching backward from from-index. Return nil if value not found.

```
(str/last-index-of "abcdefabc" "ab")
=> 6

(str/last-index-of "abcdefabc" "de" 6)
=> 3
```

SEE ALSO

[str/index-of](#)

Return index of value (string or char) in s, optionally searching forward from from-index. Return nil if value not found.

[str/index-of-char](#)

Return index of the first char of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return nil ...

[str/index-of-not-char](#)

Return index of the first char not of chars (string or sequence of chars) in s, optionally searching forward from from-index. Return ...

[section](#)[top](#)

str/letter?

```
(str/letter? s)
```

True if s is a char and the char is a letter.

Defined by Java Character.isLetter(ch).

```
(str/letter? #\x)  
=> true
```

[section](#)[top](#)

str/levenshtein

```
(str/levenshtein s1 s2)
```

Returns the *Levenshtein* distance of two strings.

The *Damerau-Levenshtein* algorithm is an extension to the *Levenshtein* algorithm which solves the edit distance problem between a source string and a target string with the following operations:

- Character Insertion
- Character Deletion
- Character Replacement
- Adjacent Character Swap

Note that the adjacent character swap operation is an edit that may be applied when two adjacent characters in the source string match two adjacent characters in the target string, but in reverse order, rather than a general allowance for adjacent character swaps.

This implementation allows the client to specify the costs of the various edit operations with the restriction that the cost of two swap operations must not be less than the cost of a delete operation followed by an insert operation. This restriction is required to preclude two swaps involving the same character being required for optimality which, in turn, enables a fast dynamic programming solution.

The cost of the *Damerau-Levenshtein* algorithm is $O(n*m)$ where n is the length of the source string and m is the length of the target string. This implementation consumes $O(n*m)$ space.

```
(str/levenshtein "Tier" "Tor") ;; => 2  
=> 2
```

```
(str/levenshtein "Tier" "tor") ;; => 3  
=> 3
```

[section](#)[top](#)

str/linefeed?

```
(str/linefeed? s)
```

True if s is a char and the char is a linefeed.

```
(str/linefeed? #\newline)
=> true
```

```
(str/linefeed? (first "
"))
=> true
```

[section](#)[top](#)

str/lorem-ipsum

```
(str/lorem-ipsum & options)
```

Creates an arbitrary length Lorem Ipsum text.

Options:

:chars n returns n characters (limited to 1000000)

:paragraphs n returns n paragraphs (limited to 100)

```
(str/lorem-ipsum :chars 250)
=> "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ac iaculis turpis. Duis dictum id sem et
consectetur. Nullam lobortis, libero non consequat aliquet, lectus diam fringilla velit, finibus eleifend ipsum
urna at lacus. Phasellus sit am"
```

```
(str/lorem-ipsum :paragraphs 1)
=> "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ac iaculis turpis. Duis dictum id sem et
consectetur. Nullam lobortis, libero non consequat aliquet, lectus diam fringilla velit, finibus eleifend ipsum
urna at lacus. Phasellus sit amet nisl fringilla, cursus est in, mollis lacus. Proin dignissim rhoncus dolor.
Cras tellus odio, elementum sed erat sit amet, euismod tincidunt nisl. In hac habitasse platea dictumst. Duis
aliquam sollicitudin tempor. Sed gravida tincidunt felis at fringilla. Morbi tempor enim at commodo vulputate.
Aenean et ultrices lorem, placerat pretium augue. In hac habitasse platea dictumst. Cras fringilla ligula quis
interdum hendrerit. Etiam at massa tempor, facilisis lacus placerat, congue erat."
```

[section](#)[top](#)

str/lower-case

```
(str/lower-case s)
(str/lower-case locale s)
```

Converts s to lowercase.

Since case mappings are not always 1:1 character mappings when a locale is given, the resulting string may be a different length than the original!

```
(str/lower-case "aBcDeF")
=> "abcdef"
```



```
(str/lower-case #\A)
=> #\a

(str/lower-case (. :java.util.Locale :new "de" "DE") "aBcDeF")
=> "abcdef"

(str/lower-case (. :java.util.Locale :GERMANY) "aBcDeF")
=> "abcdef"

(str/lower-case (. :java.util.Locale :new "de" "CH") "aBcDeF")
=> "abcdef"

(str/lower-case [ "de" ] "aBcDeF")
=> "abcdef"

(str/lower-case [ "de" "DE" ] "aBcDeF")
=> "abcdef"

(str/lower-case [ "de" "DE" ] "aBcDeF")
=> "abcdef"
```

SEE ALSO

[str/upper-case](#)

Converts s to uppercase.

[section](#)

[top](#)

str/lower-case?

```
(str/lower-case? s)
```

True if s is a char and the char is a lower case char.

Defined by Java Character.isLowerCase(ch).

```
(str/lower-case? #\x)
=> true
```

```
(str/lower-case? #\X)
=> false
```

```
(str/lower-case? #\8)
=> false
```

[section](#)

[top](#)

str/markdown->ansi

```
(str/markdown->ansi markdown width)
```

Renders markdown formatted text to ansi text

```
(let [md ""
      #Title

      *italic*

      **bold**

      [Venice](https://github.com/jlangch/venice)

      List:
      * Part 1
      * Part 2
      * Part 3

      Table:

      | JAN | 1 |
      | FEB | 22 |
      | MAR | 333 |
      ""
      ]
  (-> (str/markdown->ansi md 60)
    (println)))
```

SEE ALSO

[str/markdown->text](#)

Renders markdown formatted text to raw text

[section](#)

[top](#)

str/markdown->text

```
(str/markdown->text markdown width)
```

Renders markdown formatted text to raw text

```
(let [md ""
      #Title

      *italic*

      **bold**

      [Venice](https://github.com/jlangch/venice)

      List:
      * Part 1
      * Part 2
      * Part 3

      Table:

      | JAN | 1 |
      | FEB | 22 |
      | MAR | 333 |
      ""
      ]
  (-> (str/markdown->text md 60)
    (println)))
```

SEE ALSO

[str/markdown->ansi](#)

Renders markdown formatted text to ansi text

[section](#)

[top](#)

str/nfirst

```
(str/nfirst s n)
```

Returns a string of the n first characters of s.

```
(str/nfirst "abcdef" 2)
=> "ab"
```

```
(str/nfirst "abcdef" 10)
=> "abcdef"
```

```
(str/nfirst "abcdef" 0)
=> ""
```

[section](#)

[top](#)

str/nlast

```
(str/nlast s n)
```

Returns a string of the n last characters of s.

```
(str/nlast "abcdef" 2)
=> "ef"
```

```
(str/nlast "abcdef" 10)
=> "abcdef"
```

```
(str/nlast "abcdef" 0)
=> ""
```

[section](#)

[top](#)

str/normalize-utf

```
(str/normalize-utf text form)
```

Normalizes an UTF string.

On MacOS umlauts like ä are just encoded as 'a' plus the combining diaeresis character. Therefore an 'ä' (`\u00FC`) and an 'ä' (`a + \u0308`) from a MacOS filename are different!

This function normalizes UTF strings to simplify processing.

The *form* argument is one of:

- :NFD Canonical decomposition

- :NFC Canonical decomposition, followed by canonical composition
- :NFKD Compatibility decomposition
- :NFKC Compatibility decomposition, followed by canonical composition

```
(load-module :hexdump ['hexdump :as 'h])

;; Even though printed the same these two strings are NOT equal
;; 1: "ü"           prints to "ü"
;; 2: "u\u0308"     prints to "ü"

;; ü represented as u with combining dieresis char: \u0308
(println "u\u0308")
;; => u" (actually prints as ü on a terminal)

;; ü: \u00FC
(println "\u00FC")
;; => ü

;; u with combining dieresis character
(h/dump (bytebuf-from-string "u\u0308"))
;; 00000000: 75cc 88                                u..

;; ü
(h/dump (bytebuf-from-string "ü"))
;; 00000000: c3bc                                ..

;; ü: \u00FC
(h/dump (bytebuf-from-string "\u00FC"))
;; 00000000: c3bc                                ..

;; u with combined dieresis character normalized to get a standard ü
(h/dump (bytebuf-from-string (str/normalize-utf "u\u0308" :NFC)))
;; 00000000: c3bc                                ..

;; the reverse (decomposition)
(h/dump (bytebuf-from-string (str/normalize-utf "\u00FC" :NFD)))
;; 00000000: 75cc 88                                u..
```

SEE ALSO

[io/file-normalize-utf](#)

Normalizes the UTF string of a file path.

[section](#)

[top](#)

str/not-blank?

```
(str/not-blank? s)
```

True if s contains at least one non whitespace char.

```
(str/not-blank? "abc")
=> true
```

```
(str/not-blank? " a ")
=> true
```

```
(str/not-blank? nil)
=> false
```

```
(str/not-blank? "")  
=> false
```

```
(str/not-blank? " ")  
=> false
```

SEE ALSO

[str/blank?](#)

True if s is nil, empty, or contains only whitespace.

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[not-empty?](#)

Returns true if x is not empty. Accepts strings, collections and bytebufs.

[nil?](#)

Returns true if x is nil, false otherwise

[section](#)

[top](#)

str/nrest

```
(str/nrest s n)
```

Returns a possibly empty string of the characters after the n first characters.

```
(str/nrest "abcdef" 3)  
=> "def"
```

[section](#)

[top](#)

str/platform-lf

```
(str/platform-lf)
```

Returns the platform linefeed: LF or CR-LF.

```
(str/platform-lf
```

[section](#)

[top](#)

str/pos

```
(str/pos s pos)
```

Returns the 0 based row/column position within a string based on absolute character position. Returns a map with the keys 'row' and 'col'.

Note: CR & LF count together as one each regarding the absolute position.

```
(str/pos "abcdefghij" 4)  
=> {:col 4 :row 0}
```

```
(str/pos "ab
cdefghij" 6)
=> {:col 3 :row 1}
```

[section](#)[top](#)

str/quote

```
(str/quote str q)
(str/quote str start end)
```

Quotes a string.

```
(str/quote "abc" "-")
=> "-abc-"
```

```
(str/quote "abc" "<" ">")
=> "<abc>"
```

[section](#)[top](#)

str/quoted?

```
(str/quoted? str q)
(str/quoted? str start end)
```

Returns true if the string is quoted.

```
(str/quoted? "-abc-" "-")
=> true
```

```
(str/quoted? "<abc>" "<" ">")
=> true
```

[section](#)[top](#)

str/repeat

```
(str/repeat s n)
(str/repeat s n sep)
```

Repeats s n times with an optional separator.

```
(str/repeat "abc" 0)
=> ""
```

```
(str/repeat "abc" 3)
=> "abcabcabc"
```

```
(str/repeat "abc" 3 "-")
=> "abc-abc-abc"
```

```
(str/repeat #\* 0)
=> ""

(str/repeat #\* 3)
=> "***"

(str/repeat #\* 3 #\-)
=> "*-*-*"
```

[section](#)[top](#)

str/replace-all

```
(str/replace-all s search replacement)
```

Replaces the all occurrences of search in s. The search arg may be a string or a regex pattern

```
(str/replace-all "abcdefabc" "ab" "__")
=> "__cdef__c"

(str/replace-all "a0b01c012d" (regex/pattern "[0-9]+") "_")
=> "a_b_c_d"

(str/replace-all "a0b01c012d" #"[0-9]+" "_")
=> "a_b_c_d"
```

SEE ALSO

[str/replace-first](#)

Replaces the first occurrence of search in s. The search arg may be a string or a regex pattern. If the search arg is of type string ...

[str/replace-last](#)

Replaces the last occurrence of search in s.

[section](#)[top](#)

str/replace-first

```
(str/replace-first s search replacement & options)
```

Replaces the first occurrence of search in s. The search arg may be a string or a regex pattern. If the search arg is of type string the options :ignore-case and :nfirst are supported.

Options:

:ignore-case b if true ignores case, defaults to false
:nfirst n e.g :nfirst 2, defaults to 1

```
(str/replace-first "ab-cd-ef-ab-cd" "ab" "XYZ")
=> "XYZ-cd-ef-ab-cd"

(str/replace-first "AB-CD-EF-AB-CD" "ab" "XYZ" :ignore-case true)
=> "XYZ-CD-EF-AB-CD"
```

```
(str/replace-first "ab-ab-cd-ab-ef-ab-cd" "ab" "XYZ" :nfirst 3)
=> "XYZ-XYZ-cd-XYZ-ef-ab-cd"
```

```
(str/replace-first "a0b01c012d" (regex/pattern "[0-9]+") "_")
=> "a_b01c012d"
```

```
(str/replace-first "a0b01c012d" #"[0-9]+" "_")
=> "a_b01c012d"
```

SEE ALSO

[str/replace-last](#)

Replaces the last occurrence of search in s.

[str/replace-all](#)

Replaces the all occurrences of search in s. The search arg may be a string or a regex pattern

[section](#)

[top](#)

str/replace-last

```
(str/replace-last s search replacement & options)
```

Replaces the last occurrence of search in s.

Options:

:ignore-case b if true ignores case, defaults to false

```
(str/replace-last "abcdefabc" "ab" "XYZ")
=> "abcdefXYZc"
```

```
(str/replace-last "foo.JPG" ".jpg" ".png" :ignore-case true)
=> "foo.png"
```

SEE ALSO

[str/replace-first](#)

Replaces the first occurrence of search in s. The search arg may be a string or a regex pattern. If the search arg is of type string ...

[str/replace-all](#)

Replaces the all occurrences of search in s. The search arg may be a string or a regex pattern

[section](#)

[top](#)

str/rest

```
(str/rest s)
```

Returns a possibly empty string of the characters after the first.

```
(str/rest "abcdef")
=> "bcdef"
```

[section](#)

[top](#)

str/reverse

```
(str/reverse s)
```

Reverses a string

```
(str/reverse "abcdef")  
=> "fedcba"
```

[section](#)[top](#)

str/split

```
(str/split s regex)  
(str/split s regex limit)
```

Splits string on a regular expression. Optional argument limit is the maximum number of splits. Returns a list of the splits.

```
(str/split "abc,def,ghi" ",")  
=> ("abc" "def" "ghi")  
  
(str/split "James Peter Robert" " " 2)  
=> ("James" "Peter Robert")  
  
(str/split "abc , def , ghi" " *, *")  
=> ("abc" "def" "ghi")  
  
(str/split "abc,def,ghi" "((?<=,)|(?=,))")  
=> ("abc" ",," "def" ",," "ghi")  
  
(str/split "q1w2e3r4t5y6u7i8o9p0" "#\\d+")  
=> ("q" "w" "e" "r" "t" "y" "u" "i" "o" "p")  
  
(str/split "q1w2e3r4t5y6u7i8o9p0" "#\\d+" 5)  
=> ("q" "w" "e" "r" "t5y6u7i8o9p0")  
  
(str/split "1234567890" "#(?:<=\\G.{4})")  
=> ("1234" "5678" "90")  
  
(str/split "1234567890" "#(?:=(.{4})+$)")  
=> ("12" "3456" "7890")  
  
(str/split " q1w2 " #"")  
=> (" " "q" "1" "w" "2" " ")  
  
(str/split nil ",")  
=> ()
```

SEE ALSO

[str/split-lines](#)

Splits s into lines.

[section](#)[top](#)

str/split-at

```
(str/split-at s pos)
```

Splits string at the given position. Returns a list of the splits.

```
(str/split-at nil 1)  
=> ("" "")
```

```
(str/split-at "" 1)  
=> ("" "")
```

```
(str/split-at "abc" 0)  
=> (" " "abc")
```

```
(str/split-at "abc" 1)  
=> ("a" "bc")
```

```
(str/split-at "abc" 2)  
=> ("ab" "c")
```

```
(str/split-at "abc" 3)  
=> ("abc" "")
```

SEE ALSO

[str/split-lines](#)

Splits s into lines.

[section](#)

[top](#)

str/split-columns

```
(str/split-columns s cols)
```

Splits a string into columns. The columns are given by their start positions.

```
(str/split-columns "1abc 2d 3gh" [0 6 12])  
=> ("1abc" "2d" "3gh")
```

SEE ALSO

[str/split](#)

Splits string on a regular expression. Optional argument limit is the maximum number of splits. Returns a list of the splits.

[section](#)

[top](#)

str/split-lines

```
(str/split-lines s)
```

Splits s into lines.

```
(str/split-lines "line1  
line2  
line3")  
=> ("line1" "line2" "line3")
```

SEE ALSO

[str/split](#)

Splits string on a regular expression. Optional argument limit is the maximum number of splits. Returns a list of the splits.

[io/slurp-lines](#)

Read all lines from f.

[section](#)

[top](#)

str/starts-with?

```
(str/starts-with? s substr)
```

True if s starts with substr.

```
(str/starts-with? "abc" "ab")  
=> true
```

[section](#)

[top](#)

str/strip-end

```
(str/strip-end s substr)
```

Removes a substr only if it is at the end of a s, otherwise returns s.

```
(str/strip-end "abcdef" "def")  
=> "abc"
```

```
(str/strip-end "abcdef" "abc")  
=> "abcdef"
```

[section](#)

[top](#)

str/strip-indent

```
(str/strip-indent s)
```

Strip the indent of a multi-line string. The first line's leading whitespaces define the indent.

```
(str/strip-indent "  line1  
    line2  
    line3")  
=> "line1\n  line2\n  line3"
```

str/strip-margin

```
(str/strip-margin s)
```

Strips leading whitespaces upto and including the margin '|' from each line in a multi-line string.

```
(str/strip-margin "line1
| line2
| line3")
=> "line1\n line2\n line3"
```

str/strip-start

```
(str/strip-start s substr)
```

Removes a substr only if it is at the beginning of a s, otherwise returns s.

```
(str/strip-start "abcdef" "abc")
=> "def"
```

```
(str/strip-start "abcdef" "def")
=> "abcdef"
```

str/subs

```
(str/subs s start)
(str/subs s start end)
```

Returns the substring of s beginning at start inclusive, and ending at end (defaults to length of string), exclusive.

```
(str/subs "abcdef" 2)
=> "cdef"
```

```
(str/subs "abcdef" 2 5)
=> "cde"
```

str/trim

```
(str/trim s)
```

Trims leading and trailing whitespaces from s.

```
(str/trim " abc ")  
=> "abc"
```

SEE ALSO

[str/trim-to-empty](#)

Trims leading and trailing whitespaces from s. Returns an empty string if s is nil.

[str/trim-to-nil](#)

Trims leading and trailing whitespaces from s. Returns nil if the resulting string is empty

[str/trim-left](#)

Trims leading whitespaces from s.

[str/trim-right](#)

Trims trailing whitespaces from s.

[section](#)

[top](#)

str/trim-left

```
(str/trim-left s)
```

Trims leading whitespaces from s.

```
(str/trim-left " abc ")  
=> "abc "
```

SEE ALSO

[str/trim-right](#)

Trims trailing whitespaces from s.

[str/trim](#)

Trims leading and trailing whitespaces from s.

[str/trim-to-nil](#)

Trims leading and trailing whitespaces from s. Returns nil if the resulting string is empty

[section](#)

[top](#)

str/trim-right

```
(str/trim-right s)
```

Trims trailing whitespaces from s.

```
(str/trim-right " abc ")  
=> " abc"
```

SEE ALSO

[str/trim-left](#)

Trims leading whitespaces from s.

[str/trim](#)

Trims leading and trailing whitespaces from s.

[str/trim-to-nil](#)

Trims leading and trailing whitespaces from s. Returns nil if the resulting string is empty

[section](#)[top](#)

str/trim-to-empty

```
(str/trim-to-empty s)
```

Trims leading and trailing whitespaces from s. Returns an empty string if s is nil.

```
(str/trim-to-empty "")  
=> ""
```

```
(str/trim-to-empty "  ")  
=> ""
```

```
(str/trim-to-empty nil)  
=> ""
```

```
(str/trim-to-empty " abc ")  
=> "abc"
```

SEE ALSO

[str/trim](#)

Trims leading and trailing whitespaces from s.

[str/trim-left](#)

Trims leading whitespaces from s.

[str/trim-right](#)

Trims trailing whitespaces from s.

[section](#)[top](#)

str/trim-to-nil

```
(str/trim-to-nil s)
```

Trims leading and trailing whitespaces from s. Returns nil if the resulting string is empty

```
(str/trim-to-nil "")  
=> nil
```

```
(str/trim-to-nil "  ")  
=> nil
```

```
(str/trim-to-nil nil)  
=> nil
```

```
(str/trim-to-nil " abc ")  
=> "abc"
```

SEE ALSO

[str/trim-to-empty](#)

Trims leading and trailing whitespaces from s. Returns an empty string if s is nil.

[str/trim](#)

Trims leading and trailing whitespaces from s.

[str/trim-left](#)

Trims leading whitespaces from s.

[str/trim-right](#)

Trims trailing whitespaces from s.

[section](#)[top](#)

str/truncate

```
(str/truncate s maxlen marker mode*)
```

Truncates a string to the max length maxlen and adds the marker if the string needs to be truncated. The marker is added to the start, middle, or end of the string depending on the mode :start, :middle, :end. The mode defaults to :end

```
(str/truncate "abcdefghij" 20 "...")  
=> "abcdefghij"
```

```
(str/truncate "abcdefghij" 9 "...")  
=> "abcdef..."
```

```
(str/truncate "abcdefghij" 4 "...")  
=> "a..."
```

```
(str/truncate "abcdefghij" 7 "...":start)  
=> "...ghij"
```

```
(str/truncate "abcdefghij" 7 "...":middle)  
=> "ab...ij"
```

```
(str/truncate "abcdefghij" 7 "...":end)  
=> "abcd..."
```

[section](#)[top](#)

str/upper-case

```
(str/upper-case s)  
(str/upper-case locale s)
```

Converts s to uppercase.

Since case mappings are not always 1:1 character mappings when a locale is given, the resulting string may be a different length than the original!

```
(str/upper-case "aBcDeF")  
=> "ABCDEF"
```

```
(str/upper-case #\a)  
=> #\A
```

```
(str/upper-case (. :java.util.Locale :new "de" "DE") "aBcDeF")  
=> "ABCDEF"
```

```
(str/upper-case (. :java.util.Locale :GERMANY) "aBcDeF")  
=> "ABCDEF"
```

```
(str/upper-case (. :java.util.Locale :new "de" "CH") "aBcDeF")  
=> "ABCDEF"
```

```
(str/upper-case [ "de" ] "aBcDeF")  
=> "ABCDEF"
```

```
(str/upper-case [ "de" "DE" ] "aBcDeF")  
=> "ABCDEF"
```

```
(str/upper-case [ "de" "DE" ] "aBcDeF")  
=> "ABCDEF"
```

SEE ALSO

[str/lower-case](#)

Converts s to lowercase.

[section](#)

[top](#)

str/upper-case?

```
(str/upper-case? s)
```

True if s is a char and the char is an upper case char.

Defined by Java Character.isUpperCase(ch).

```
(str/upper-case? #\x)  
=> false
```

```
(str/upper-case? #\X)  
=> true
```

```
(str/upper-case? #\8)  
=> false
```

[section](#)

[top](#)

str/valid-email-addr?

```
(str/valid-email-addr? e)
```

Returns true if e is a valid email address according to RFC5322, else returns false

```
(str/valid-email-addr? "user@domain.com")  
=> true
```



```
(str/valid-email-addr? "user@domain.co.in")
=> true

(str/valid-email-addr? "user.name@domain.com")
=> true

(str/valid-email-addr? "user_name@domain.com")
=> true

(str/valid-email-addr? "username@yahoo.corporate.in")
=> true
```

[section](#)[top](#)

str/whitespace?

```
(str/whitespace? s)
```

True if s is char and the char is a whitespace.

Defined by Java Character.isWhitespace(ch).

```
(str/whitespace? #\space)
=> true
```

[section](#)[top](#)

str/wrap

```
(str/wrap text & options)
```

Wraps ascii text to lines with a length of maxlen characters .

Options:

:maxlen n the max len of line (default 80)

:line-wrap controls the line wrap

{:

anywhere,

:break-

word}

```
(-> (str/lorem-ipsum :paragraphs 1)
    (str/wrap :maxlen 80 :line-wrap :break-word))
=> "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ac iaculis\nturpis. Duis dictum id sem et
consectetur. Nullam lobortis, libero non consequat\naliquet, lectus diam fringilla velit, finibus eleifend
ipsum urna at lacus.\nPhasellus sit amet nisl fringilla, cursus est in, mollis lacus. Proin dignissim\nrhoncus
dolor. Cras tellus odio, elementum sed erat sit amet, euismod tincidunt\nnisl. In hac habitasse platea
dictumst. Duis aliquam sollicitudin tempor. Sed\ngravida tincidunt felis at fringilla. Morbi tempor enim at
commodo vulputate.\nAenean et ultrices lorem, placerat pretium augue. In hac habitasse platea\ndictumst. Cras
fringilla ligula quis interdum hendrerit. Etiam at massa tempor,\nfacilisis lacus placerat, congue erat."
```

[section](#)[top](#)

string-array

```
(string-array coll)
(string-array len)
(string-array len init-val)
```

Returns an array of Java strings containing the contents of coll or returns an array with the given length and optional init value

```
(string-array '("1" "2" "3"))
=> [1, 2, 3]

(string-array 10)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]

(string-array 10 "42")
=> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

SEE ALSO

[java-string-list](#)

Converts a Venice list/vector to a Java String list

[section](#)

[top](#)

string?

```
(string? x)
```

Returns true if x is a string

```
(string? "abc")
=> true

(string? 1)
=> false

(string? nil)
=> false
```

[section](#)

[top](#)

sublist

```
(sublist l start) (sublist l start end)
```

Returns a list of the items in list from start (inclusive) to end (exclusive). If end is not supplied, defaults to (count list).

`sublist` accepts a lazy-seq if both start and end is given.

```
(sublist '(1 2 3 4 5 6) 2)
=> (3 4 5 6)
```

```
(sublist '(1 2 3 4 5 6) 2 3)
=> (3)
```

```
(doall (sublist (lazy-seq 1 inc) 3 7))
=> (4 5 6 7)
```

SEE ALSO

[subvec](#)

Returns a vector of the items in vector from start (inclusive) to end (exclusive). If end is not supplied, defaults to (count vector)

[section](#)

[top](#)

subset?

```
(subset? set1 set2)
```

Return true if set1 is a subset of set2

```
(subset? #{2 3} #{1 2 3 4})
=> true
```

```
(subset? #{2 5} #{1 2 3 4})
=> false
```

SEE ALSO

[set](#)

Creates a new set containing the items.

[superset?](#)

Return true if set1 is a superset of set2

[union](#)

Return a set that is the union of the input sets

[difference](#)

Return a set that is the first set without elements of the remaining sets

[intersection](#)

Return a set that is the intersection of the input sets

[section](#)

[top](#)

subvec

```
(subvec v start) (subvec v start end)
```

Returns a vector of the items in vector from start (inclusive) to end (exclusive). If end is not supplied, defaults to (count vector)

```
(subvec [1 2 3 4 5 6] 2)
=> [3 4 5 6]
```

```
(subvec [1 2 3 4 5 6] 2 3)
=> [3]
```

SEE ALSO

[sublist](#)

Returns a list of the items in list from start (inclusive) to end (exclusive). If end is not supplied, defaults to (count list).

[section](#)

[top](#)

supers

(supers class)

Returns the immediate and indirect superclasses and interfaces of class, if any.

```
(supers :java.util.ArrayList)
=> (:java.util.ArrayList :java.util.AbstractCollection :java.util.List :java.util.Collection :java.lang.
Iterable)
```

[section](#)

[top](#)

superset?

(superset? set1 set2)

Return true if set1 is a superset of set2

```
(superset? #{1 2 3 4} #{2 3} )
=> true
```

```
(superset? #{1 2 3 4} #{2 5})
=> false
```

SEE ALSO

[set](#)

Creates a new set containing the items.

[subset?](#)

Return true if set1 is a subset of set2

[union](#)

Return a set that is the union of the input sets

[difference](#)

Return a set that is the first set without elements of the remaining sets

[intersection](#)

Return a set that is the intersection of the input sets

[section](#)

[top](#)

supertype

(supertype x)

Returns the super type of x.

```
(supertype 5)
=> :core/number

(supertype [1 2])
=> :core/sequence

(supertype (. :java.math.BigInteger :valueOf 100))
=> :java.lang.Number
```

SEE ALSO

[type](#)

Returns the type of x.

[supertypes](#)

Returns the super types of x.

[instance-of?](#)

Returns true if x is an instance of the given type

[section](#)

[top](#)

supertypes

```
(supertypes x)
```

Returns the super types of x.

```
(supertypes 5)
=> (:core/number :core/val)

(supertypes [1 2])
=> (:core/sequence :core/collection :core/val)

(supertypes (. :java.math.BigInteger :valueOf 100))
=> (:java.lang.Number :java.lang.Object)
```

SEE ALSO

[type](#)

Returns the type of x.

[supertype](#)

Returns the super type of x.

[instance-of?](#)

Returns true if x is an instance of the given type

[section](#)

[top](#)

swap!

```
(swap! box f & args)
```

Atomically swaps the value of an atom or a volatile to be: `(apply f current-value-of-box args)` . Note that `f` may be called multiple times, and thus should be free of side effects. Returns the value that was swapped in.

```
(do
  (def counter (atom 0))
  (swap! counter inc))
=> 1

(do
  (def counter (atom 0))
  (swap! counter inc)
  (swap! counter + 1)
  (swap! counter #(inc %))
  (swap! counter (fn [x] (inc x))))
@counter
=> 4

(do
  (def fruits (atom ()))
  (swap! fruits conj :apple)
  (swap! fruits conj :mango)
  @fruits)
=> (:apple :mango)

(do
  (def counter (volatile 0))
  (swap! counter (partial + 6))
  @counter)
=> 6
```

SEE ALSO

[swap-vals!](#)

Atomically swaps the value of an atom to be: `(apply f current-value-of-atom args)`. Note that `f` may be called multiple times, and thus ...

[reset!](#)

Sets the value of an atom or a volatile to `newval` without regard for the current value. Returns `newval`.

[compare-and-set!](#)

Atomically sets the value of atom to `newval` if and only if the current value of the atom is identical to `oldval`. Returns true if set ...

[atom](#)

Creates an atom with the initial value `x`.

[volatile](#)

Creates a volatile with the initial value `x`

[section](#)

[top](#)

swap-vals!

```
(swap-vals! atom f & args)
```

Atomically swaps the value of an atom to be: `(apply f current-value-of-atom args)` . Note that `f` may be called multiple times, and thus should be free of side effects. Returns `[old new]`, the value of the atom before and after the swap.

```
(do
  (def queue (atom '(1 2 3)))
  (swap-vals! queue pop))
=> [(1 2 3) (2 3)]
```

SEE ALSO

[swap!](#)

Atomically swaps the value of an atom or a volatile to be: (apply f current-value-of-box args). Note that f may be called multiple ...

[reset!](#)

Sets the value of an atom or a volatile to newval without regard for the current value. Returns newval.

[compare-and-set!](#)

Atomically sets the value of atom to newval if and only if the current value of the atom is identical to oldval. Returns true if set ...

[atom](#)

Creates an atom with the initial value x.

[volatile](#)

Creates a volatile with the initial value x

[section](#)[top](#)

swiss-cities/close

```
(close)
```

Close the service and free all allocated resources.

```
(do
  (load-module :swiss-cities)

  (swiss-cities/swiss-cities/download-city-data)

  (swiss-cities/close))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/open?](#)

Returns true if the the LV95 Swiss geo data have been successfully loaded else false.

[section](#)[top](#)

swiss-cities/distance

```
(distance location-1 location-2)
```

Calculate the distance in kilometers between to Swiss locations.

Returns the distance as a rounded long value

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data))
```

```
(swiss-cities/distance "Basel" "Bern"))

(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/distance (swiss-cities/find-location "Basel")
    (swiss-cities/find-location "Bern")))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/distance](#)

Calculate the distance in kilometers between two Swiss locations.

[section](#)[top](#)

swiss-cities/download-city-data

```
(download-city-data)
```

Downloads the LV95 Swiss Geo data from [Swiss Geo Admin](#), parses it and keeps it in the module for subsequent use.

The LV95 Swiss Geo data can be renewed any time by downloading it again.

Alternatively use `swiss-cities/load-city-data-from-classpath` or `swiss-cities/slurp-city-data`.

[Swiss Topo Cities](#)

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/gemeinden)
  (swiss-cities/ortschaften))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/close](#)

Close the service and free all allocated resources.

[swiss-cities/open?](#)

Returns true if the LV95 Swiss geo data have been successfully loaded else false.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the specified plz.

[swiss-cities/distance](#)

Calculate the distance in kilometers between two Swiss locations.

[swiss-cities/locations](#)

Returns a list of all locations.

[swiss-cities/ortschaften](#)

List all Ortschaften alphabetically.

[swiss-cities/gemeinden](#)

List all Gemeinden alphabetically.

[section](#)

[top](#)

swiss-cities/find-by-gemeinde

```
(find-by-gemeinde gemeinde)
(find-by-gemeinde gemeinde kanton)
```

Find all locations by a *Gemeinde* and optionally by a canton.

The argument 'gemeinde' may be a full name or a regular expression to specify a political *Gemeinde*.

Returns a list of the matching locations.

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-by-gemeinde "Maur")
  (swiss-cities/find-by-gemeinde #"Mau.*")

  ;; in canton 'ZH' only
  (swiss-cities/find-by-gemeinde "Maur" "ZH")
  (swiss-cities/find-by-gemeinde #"Mau.*" "ZH"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the spcified plz.

[section](#)

[top](#)

swiss-cities/find-by-ortschaft

```
(find-by-ortschaft ortschaft)
(find-by-ortschaft ortschaft kanton)
```

Find all locations by an *Ortschaft* and optionally by a canton.

The argument 'ortschaft' may be a full name or a regular expression to specify an *Ortschaft*.

Returns a list of the matching locations.

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-by-ortschaft "Maur")
  (swiss-cities/find-by-ortschaft #"Mau.*"))

;; in canton 'ZH' only
(swiss-cities/find-by-ortschaft "Maur" "ZH")
(swiss-cities/find-by-ortschaft #"Mau.*" "ZH"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the spcified plz.

swiss-cities/find-by-ortschaft-and-gemeinde

```
(find-by-ortschaft-and-gemeinde name)
(find-by-ortschaft-and-gemeinde name kanton)
```

Find all locations by an *Ortschaft* **and** a *Gemeinde* and optionally by a canton.

The argument 'name' may be a full name or a regular expression to specify an *Ortschaft* or a political *Gemeinde*.

Returns a list of the matching locations.

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-by-ortschaft-or-gemeinde "Maur")
  (swiss-cities/find-by-ortschaft-or-gemeinde #"Mau.*")

  ;; in canton 'ZH' only
  (swiss-cities/find-by-ortschaft-or-gemeinde "Maur" "ZH")
  (swiss-cities/find-by-ortschaft-or-gemeinde #"Mau.*" "ZH"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the specified plz.

swiss-cities/find-by-ortschaft-or-gemeinde

```
(find-by-ortschaft-or-gemeinde name)
(find-by-ortschaft-or-gemeinde name kanton)
```

Find all locations by an *Ortschaft* **or** a *Gemeinde* and optionally by a canton.

The argument 'name' may be a full name or a regular expression to specify an *Ortschaft* or a political *Gemeinde*.

Returns a list of the matching locations.

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-by-ortschaft-or-gemeinde "Maur")
  (swiss-cities/find-by-ortschaft-or-gemeinde #"Mau.*")

  ;; in canton 'ZH' only
  (swiss-cities/find-by-ortschaft-or-gemeinde "Maur" "ZH")
  (swiss-cities/find-by-ortschaft-or-gemeinde #"Mau.*" "ZH"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the spcified plz.

[section](#)[top](#)

swiss-cities/find-by-plz

```
(find-by-plz plz)
```

Find all locations with the spcified plz.

Returns a list of the matching locations.

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-by-plz 7132)
  (swiss-cities/find-by-plz "7132"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `:java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/plz?](#)

Return true if the 'plz' is valid else false

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[section](#)

[top](#)

swiss-cities/find-location

(find-location name)

Find a location (*Ortschaft* or *Gemeinde*) by name.

Returns the location's geo data or `nil` if the location has not been found in the data.

Example location data:

```
{ :ortschaft "Basel"
  :plz "4001"
  :gemeinde "Basel"
  :kanton "BS"
  :east 2611314.0
  :north 1267295.0
  :validity 2008-07-01
}
```

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-location "Basel"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `:java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the specified plz.

[section](#)

[top](#)

swiss-cities/gemeinden

(gemeinden)

List all *Gemeinden* alphabetically.

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/gemeinden))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/locations](#)

Returns a list of all locations.

[swiss-cities/ortschaften](#)

List all Ortschaften alphabetically.

[section](#)

[top](#)

swiss-cities/load-city-data-from-classpath

(load-city-data-from-classpath resource)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

The LV95 Swiss geo data can be renewed any time by reloading it again.

Alternatively use `swiss-cities/download-city-data` or `swiss-cities/slurp-city-data`.

```
(do
  (load-module :swiss-cities)

  (def resource "com/github/jlangch/venice/data/ortschaftenverzeichnis_plz_2056.csv.zip")
  (swiss-cities/load-city-data-from-classpath resource)

  (swiss-cities/gemeinden)
  (swiss-cities/ortschaften))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/close](#)

Close the service and free all allocated resources.

[swiss-cities/open?](#)

Returns true if the the LV95 Swiss geo data have been successfully loaded else false.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the spcified plz.

[swiss-cities/distance](#)

Calculate the distance in kilometers between to Swiss locations.

[swiss-cities/locations](#)

Returns a list of all locations.

[swiss-cities/ortschaften](#)

List all Ortschaften alphabetically.

[swiss-cities/gemeinden](#)

List all Gemeinden alphabetically.

[section](#)[top](#)

swiss-cities/locations

(locations)

Returns a list of all locations.

Example location data:

```
{ :ortschaft "Basel"
  :plz "4001"
  :gemeinde "Basel"
  :kanton "BS"
  :east 2611314.0
  :north 1267295.0
  :validity 2008-07-01
}
```

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/find-location "Basel"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/ortschaften](#)

List all *Ortschaften* alphabetically.

[swiss-cities/gemeinden](#)

List all *Gemeinden* alphabetically.

[section](#)[top](#)

swiss-cities/open?

```
(open?)
```

Returns `true` if the the LV95 Swiss geo data have been successfully loaded else `false` .

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/open?))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/close](#)

Close the service and free all allocated resources.

[section](#)[top](#)

swiss-cities/ortschaften

```
(ortschaften)
```

List all *Ortschaften* alphabetically.


```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/ortschaften))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java :java.io.InputStream, parses it and keeps it in the module for subsequent use.

[swiss-cities/locations](#)

Returns a list of all locations.

[swiss-cities/gemeinden](#)

List all Gemeinden alphabetically.

[section](#)

[top](#)

swiss-cities/plz?

```
(plz? plz)
```

Return `true` if the 'plz' is valid else `false`

```
(do
  (load-module :swiss-cities)
  (swiss-cities/download-city-data)

  (swiss-cities/plz? 7132)
  (swiss-cities/plz? "7132"))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java :java.io.InputStream, parses it and keeps it in the module for subsequent use.

[swiss-cities/find-by-plz](#)

Find all locations with the spcified plz.

[swiss-cities/locations](#)

Returns a list of all locations.

[swiss-cities/ortschaften](#)

List all Ortschaften alphabetically.

[section](#)

[top](#)

swiss-cities/slurp-city-data

```
(slurp-city-data f)
```

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `:java.io.InputStream`, parses it and keeps it in the module for subsequent use.

The LV95 Swiss geo data can be renewed any time by reloading it again.

Alternatively use `swiss-cities/download-city-data` or `swiss-cities/load-city-data-from-classpath`.

```
(do
  (load-module :swiss-cities)

  (def csv "/data/ortschaftenverzeichnis_plz_2056.csv.zip")
  (swiss-cities/slurp-city-data (io/file csv))

  (swiss-cities/gemeinden)
  (swiss-cities/ortschaften))
```

SEE ALSO

[swiss-cities/download-city-data](#)

Downloads the LV95 Swiss Geo data from Swiss Geo Admin (<https://data.geo.admin.ch>), parses it and keeps it in the module for subsequent use.

[swiss-cities/load-city-data-from-classpath](#)

Loads the LV95 Swiss geo data (ZIP file) from a classpath resource, parses it and keeps it in the module for subsequent use.

[swiss-cities/slurp-city-data](#)

Loads the LV95 Swiss geo data (ZIP file) from a file or a Java `:java.io.InputStream`, parses it and keeps it in the module for subsequent use.

[swiss-cities/close](#)

Close the service and free all allocated resources.

[swiss-cities/open?](#)

Returns true if the the LV95 Swiss geo data have been successfully loaded else false.

[swiss-cities/find-location](#)

Find a location (Ortschaft or Gemeinde) by name.

[swiss-cities/find-by-ortschaft](#)

Find all locations by an Ortschaft and optionally by a canton.

[swiss-cities/find-by-gemeinde](#)

Find all locations by a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-or-gemeinde](#)

Find all locations by an Ortschaft or a Gemeinde and optionally by a canton.

[swiss-cities/find-by-ortschaft-and-gemeinde](#)

Find all locations by an Ortschaft and a Gemeinde and optionally by a canton.

[swiss-cities/find-by-plz](#)

Find all locations with the spcified plz.

[swiss-cities/distance](#)

Calculate the distance in kilometers between to Swiss locations.

[swiss-cities/locations](#)

Returns a list of all locations.

[swiss-cities/ortschaften](#)

List all Ortschaften alphabetically.

[swiss-cities/gemeinden](#)

List all Gemeinden alphabetically.

symbol

```
(symbol name)
(symbol ns name)
```

Returns a symbol from the given name

```
(symbol "a")
=> a

(symbol "foo" "a")
=> foo/a

(symbol *ns* "a")
=> user/a

(symbol 'a)
=> a

((resolve (symbol "core" "+")) 1 2)
=> 3

(name str/reverse)
=> "reverse"

(namespace str/reverse)
=> "str"
```

SEE ALSO

[resolve](#)

Resolves a symbol.

[name](#)

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[section](#)

[top](#)

symbol?

```
(symbol? x)
```

Returns true if x is a symbol

```
(symbol? 'a)
=> true

(symbol? (symbol "a"))
=> true

(symbol? nil)
=> false
```

```
(symbol? :a)
=> false
```

[section](#)[top](#)

system-env

```
(system-env)
(system-env name)
(system-env name default-val)
```

Returns the system env variable with the given name. Returns the default-val if the variable does not exist or it's value is nil.

Without arguments returns all system env variables authorized by the configured sandbox.

```
(system-env :SHELL)
=> "/bin/zsh"

(system-env :FOO "test")
=> "test"

(system-env "SHELL")
=> "/bin/zsh"
```

SEE ALSO

[system-prop](#)

Returns the system property with the given name. Returns the default-val if the property does not exist or it's value is nil.

[section](#)[top](#)

system-exit-code

```
(system-exit-code code)
```

Defines the exit code that is used if the Java VM exits. Defaults to 0.

Note:

The exit code is only used when the Venice launcher has been used to run a script file, a command line script, a Venice app archive, or the REPL.

```
(system-exit-code 0)
```

[section](#)[top](#)

system-prop

```
(system-prop)
(system-prop name)
(system-prop name default-val)
```

Returns the system property with the given name. Returns the default-val if the property does not exist or it's value is nil.

Without arguments returns all system properties authorized by the configured sandbox.

```
(system-prop :os.name)
=> "Mac OS X"

(system-prop :foo.org "abc")
=> "abc"

(system-prop "os.name")
=> "Mac OS X"
```

SEE ALSO

[system-env](#)

Returns the system env variable with the given name. Returns the default-val if the variable does not exist or it's value is nil.

[section](#)

[top](#)

tail-pos

```
(tail-pos)
(tail-pos name)
```

Throws a `NotInTailPositionException` if the expr is not in tail position otherwise returns nil.

Definition:

The tail position is a position which an expression would return a value from. There are no more forms evaluated after the form in the tail position is evaluated.

```
;; in tail position
(do 1 (tail-pos))
=> nil

;; not in tail position
(do (tail-pos) 1)
=> NotInTailPositionException: Not in tail position
```

[section](#)

[top](#)

take

```
(take n coll)
```

Returns a collection of the first n items in coll, or all items if there are fewer than n.

Returns a stateful transducer when no collection is provided. Returns a lazy sequence if coll is a lazy sequence.

```
(take 3 [1 2 3 4 5])
=> [1 2 3]

(take 10 [1 2 3 4 5])
=> [1 2 3 4 5]

(doall (take 4 (repeat 3)))
=> (3 3 3 3)

(doall (take 10 (cycle (range 0 3))))
=> (0 1 2 0 1 2 0 1 2 0)
```

take!

```
(take! queue)
```

Retrieves and removes the head value of the queue or the deque, waiting if necessary until a value becomes available.

```
(let [q (queue)]
  (put! q 1)
  (take! q)
  q)
=> ()
```

SEE ALSO

[queue](#)

Creates a new mutable threadsafe bounded or unbounded queue.

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

take-last

```
(take-last n coll)
```

Return a sequence of the last n items in coll.

Returns a stateful transducer when no collection is provided.

```
(take-last 3 [1 2 3 4 5])
=> [3 4 5]
```

```
(take-last 10 [1 2 3 4 5])
=> [1 2 3 4 5]
```

take-tail!

```
(take-tail! deque)
```

Retrieves and removes the tail value of the deque, waiting if necessary until a value becomes available.

```
(let [q (deque)]
  (put! q 1)
  (put! q 2)
  (put! q 3)
  (put! q 4)
  (println q)
  (println (take-tail! q))
  (println (take! q))
  (println q))
(1 2 3 4)
4
1
(2 3)
=> nil
```

SEE ALSO

[deque](#)

Creates a new mutable threadsafe bounded or unbounded deque.

[put!](#)

Puts an item to a queue. The operation is synchronous, it waits indefinitely until the value can be placed on the queue. Returns always nil.

[offer!](#)

Offers an item to tail of the a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[put-head!](#)

Puts an item to the head of a deque. The operation is synchronous, it waits indefinitely until the value can be placed on the queue.

[offer-head!](#)

Offers an item to the head of a deque with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait ...

[poll-tail!](#)

Polls an item from the tail of a deque with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element (or nil if the stack is empty), for a queue the ...

[empty?](#)

Returns true if x is empty. Accepts strings, collections and bytebufs.

[count](#)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

[section](#)

[top](#)

take-while

```
(take-while predicate coll)
```

Returns a list of successive items from coll while (predicate item) returns logical true.

Returns a transducer when no collection is provided.

```
(take-while neg? [-2 -1 0 1 2 3])
=> [-2 -1]
```

tap>

```
(tap> x)
```

Sends x to any taps. Will not block. Returns true if there was room in the queue, false if not (x is dropped).

```
(do
  (add-tap prn)
  (tap> {:foo "hello" :bar 34.5}))
=> true
```

SEE ALSO

[add-tap](#)

adds f, a fn of one argument, to the tap set. This function will be called with anything sent via tap>.

[remove-tap](#)

Remove f from the tap set.

[clear-taps](#)

Removes all tap sets.

tee

```
(tee f coll)
```

Applies f to every item of each the coll and returns the unaltered coll.
Returns a transducer when no collection is provided.

```
(tee println [1 2 3 4])
1
2
3
4
=> (1 2 3 4)
```

```
(->> [1 2 3 4]
  (tee println)
  (map inc))
1
2
3
4
=> (2 3 4 5)
```

SEE ALSO

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[filter](#)

Returns a collection of the items in coll for which (predicate item) returns logical true.

reduce

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

map-indexed

Returns a collection of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item of coll, etc. until ...

[section](#)[top](#)

test/deftest

```
(deftest name & body)
```

Defines a test function with no arguments.

All assertion macros are available for test assertions within the test function body:

- `assert`
- `assert-false`
- `assert-eq`
- `assert-ne`
- `assert-throws`
- `assert-does-not-throw`
- `assert-throws-with-msg`

It's recommended to use dedicated test namespaces for the tests and to group tests by namespaces.

Note: Actually, the test body goes in the `:test` metadata on the var, and the real function (the value of the var) calls `test-var` on itself.

```
(do
  (load-module :test)

  (ns foo-test)

  (test/deftest add-test []
    (assert-eq 0 (+ 0 0))
    (assert-eq 3 (+ 1 2)))

  (test/deftest mul-test []
    (assert-eq 6 (* 2 3)))

  (ns bar)
  (test/run-tests 'foo-test))
```

Testing namespace 'foo-test

PASS foo-test/add-test

PASS foo-test/mul-test

Ran 2 tests with 3 assertions

0 failures, 0 errors.

```
=> {:assert 3 :error 0 :pass 2 :test 2 :type :summary :fail 0}
```

;; Explicit setup/teardown

```
(do
  (ns foo-test)
  (load-module :test)

  (test/deftest sum-test []
    (let [f (io/temp-file "test-", ".txt")]
      (try
```

```
(io/spit f "1234" :append true)
(assert-eq "1234" (io/slurp f :binary false))
(finally
  (io/delete-file f))))
```

```
(test/run-tests *ns*)
```

Testing namespace 'foo-test

PASS foo-test/sum-test

Ran 1 tests with 1 assertions

0 failures, 0 errors.

```
=> {:assert 1 :error 0 :pass 1 :test 1 :type :summary :fail 0}
```

SEE ALSO

[test/run-tests](#)

Runs all tests in the given namespaces; prints results. The tests are run grouped the namespace.

[test/run-test-var](#)

Runs a single test; prints results. Returns a map summarizing the test results.

[test/use-fixtures](#)

Wrap test runs in a fixture function to perform setup and teardown. Fixtures are always bound to a namespace, hence tests from different ...

[test/successful?](#)

Returns true if the given test summary indicates all tests were successful, false otherwise.

[assert](#)

Evaluates expr and throws an :AssertionException exception if it does not evaluate to logical true.

[assert-false](#)

Evaluates expr and throws an :AssertionException exception if it does not evaluate to logical false.

[assert-eq](#)

Assert that expected and actual are equal. Throws an :AssertionException exception if they are not equal.

[assert-ne](#)

Assert that unexpected and actual are not equal. Throws an :AssertionException exception if they are equal.

[assert-throws](#)

Evaluates expr and throws an :AssertionException exception if it does not throw the expected exception of type ex-type.

[assert-does-not-throw](#)

Evaluates expr and throws an :AssertionException exception if it does throw any kind of exception.

[section](#)

[top](#)

test/run-test-var

```
(run-test-var v)
```

Runs a single test; prints results. Returns a map summarizing the test results.

```
(do
  (ns foo-test)
  (load-module :test)

  (test/deftest plus-test []
    (assert-eq 3 (+ 1 2)))

  (test/run-test-var plus-test))
```

```
Testing namespace 'foo-test
```

```
PASS foo-test/plus-test
```

```
Ran 1 tests with 1 assertions
```

```
0 failures, 0 errors.
```

```
=> {:assert 1 :error 0 :pass 1 :test 1 :type :summary :fail 0}
```

SEE ALSO

[test/deftest](#)

Defines a test function with no arguments.

[test/run-tests](#)

Runs all tests in the given namespaces; prints results. The tests are run grouped the namespace.

[test/use-fixtures](#)

Wrap test runs in a fixture function to perform setup and teardown. Fixtures are always bound to a namespace, hence tests from different ...

[section](#)

[top](#)

test/run-tests

```
(run-tests & namespaces)
```

Runs all tests in the given namespaces; prints results. The tests are run grouped the namespace.

Returns a map summarizing test results.

```
(do
  (load-module :test)

  (ns foo-test)
  (test/deftest add-test []
    (assert-eq 3 (+ 1 2)))
  (test/deftest sub-test []
    (assert-eq 1 (- 2 1)))

  (ns bar-test)
  (test/deftest mul-test []
    (assert-eq 2 (* 1 2)))

  (test/run-tests 'foo-test 'bar-test))
```

```
Testing namespace 'foo-test
```

```
PASS foo-test/add-test
```

```
PASS foo-test/sub-test
```

```
Testing namespace 'bar-test
```

```
PASS bar-test/mul-test
```

```
Ran 3 tests with 3 assertions
```

```
0 failures, 0 errors.
```

```
=> {:assert 3 :error 0 :pass 3 :test 3 :type :summary :fail 0}
```

SEE ALSO

[test/deftest](#)

Defines a test function with no arguments.

[test/run-test-var](#)

Runs a single test; prints results. Returns a map summarizing the test results.

[test/use-fixtures](#)

Wrap test runs in a fixture function to perform setup and teardown. Fixtures are always bound to a namespace, hence tests from different ...

[section](#)[top](#)

test/successful?

```
(successful? summary)
```

Returns true if the given test summary indicates all tests were successful, false otherwise.

```
(do
  (ns foo-test)
  (load-module :test)

  (test/deftest plus-test []
    (assert-eq 3 (+ 1 2)))

  (let [summary (test/run-tests 'foo-test)]
    (test/successful? summary)))
```

Testing namespace 'foo-test

PASS foo-test/plus-test

Ran 1 tests with 1 assertions
0 failures, 0 errors.
=> true

SEE ALSO

[test/deftest](#)

Defines a test function with no arguments.

[test/run-tests](#)

Runs all tests in the given namespaces; prints results. The tests are run grouped the namespace.

[test/run-test-var](#)

Runs a single test; prints results. Returns a map summarizing the test results.

[test/use-fixtures](#)

Wrap test runs in a fixture function to perform setup and teardown. Fixtures are always bound to a namespace, hence tests from different ...

[section](#)[top](#)

test/use-fixtures

```
(use-fixtures ns fixture-type & fixture-fns)
```

Wrap test runs in a fixture function to perform setup and teardown. Fixtures are always bound to a namespace, hence tests from different namespaces have different fixtures.

A fixture of type `:each` is called before and after each test in the fixture's namespace.

A fixture of type `:once` is called before the first and after the last test in the fixture's namespace serving as an initial setup and final teardown.

To pass a value from a fixture to the tests dynamic vars can be used. See the 3rd example below.

```
;; Fixtures :each
;; Adds logic for a setup and teardown method that will be called
;; before and after each test
(do
  (load-module :test)

  (defn each-time-setup []
    (println "FIXTURE each time setup"))

  (defn each-time-teardown []
    (println "FIXTURE each time teardown"))

  (defn each-fixture [f]
    (each-time-setup)
    (try
      (f)
      (finally (each-time-teardown))))

  ;; register as an each-time callback
  (test/use-fixtures *ns* :each each-fixture)

  (test/deftest add-test []
    (assert-eq 3 (+ 1 2)))

  (test/deftest sub-test []
    (assert-eq 3 (- 4 1)))

  (test/run-tests *ns*))
```

Testing namespace 'user

```
FIXTURE each time setup
PASS user/add-test
FIXTURE each time teardown
FIXTURE each time setup
PASS user/sub-test
FIXTURE each time teardown
```

Ran 2 tests with 2 assertions

0 failures, 0 errors.

```
=> {:assert 2 :error 0 :pass 2 :test 2 :type :summary :fail 0}
```

```
;; Fixtures :once
;; Adds logic for a setup and teardown method that will be called
;; before the first and after the last test as an initial setup
;; and final teardown
(do
  (load-module :test)

  (defn one-time-setup []
    (println "FIXTURE one time setup"))

  (defn one-time-teardown []
    (println "FIXTURE one time teardown"))

  (defn one-fixture [f]
    (one-time-setup)
    (try
      (f)
      (finally (one-time-teardown))))
```

```
;; register as a one-time callback
(test/use-fixtures *ns* :once one-fixture)

(test/deftest add-test []
  (assert-eq 3 (+ 1 2)))

(test/deftest sub-test []
  (assert-eq 3 (- 4 1)))

(test/run-tests *ns*)

Testing namespace 'user

FIXTURE one time setup
PASS user/add-test
PASS user/sub-test
FIXTURE one time teardown

Ran 2 tests with 2 assertions
0 failures, 0 errors.
=> {:assert 2 :error 0 :pass 2 :test 2 :type :summary :fail 0}
```

;; Passing a value from a setup fixture to the tests

```
(do
  (load-module :test)

  (def-dynamic *state* 0)

  (defn one-time-setup []
    (println "FIXTURE one-time setup")
    100)

  (defn one-time-teardown []
    (println "FIXTURE one-time teardown"))

  (defn one-fixture [f]
    (binding [*state* (one-time-setup)]
      (try
        (f)
        (finally (one-time-teardown))))))

;; register as a one-time callback
(test/use-fixtures *ns* :once one-fixture)

(test/deftest add-test []
  (println "state user/add-test:" *state*)
  (assert-eq 3 (+ 1 2)))

(test/deftest sub-test []
  (println "state user/sub-test:" *state*)
  (assert-eq 3 (- 4 1)))

(test/run-tests *ns*)
```

Testing namespace 'user

```
FIXTURE one-time setup
state user/add-test: 100
PASS user/add-test
state user/sub-test: 100
PASS user/sub-test
FIXTURE one-time teardown
```

Ran 2 tests with 2 assertions

```
0 failures, 0 errors.  
=> {:assert 2 :error 0 :pass 2 :test 2 :type :summary :fail 0}
```

SEE ALSO

[test/deftest](#)

Defines a test function with no arguments.

[test/run-tests](#)

Runs all tests in the given namespaces; prints results. The tests are run grouped the namespace.

[test/run-test-var](#)

Runs a single test; prints results. Returns a map summarizing the test results.

[section](#)[top](#)

then-accept

```
(then-accept p f)
```

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

```
(-> (promise (fn [] "the quick brown fox"))  
    (then-accept (fn [v] (println (pr-str v)))))  
    (deref))  
"the quick brown fox"  
=> nil  
  
(let [result (promise)  
      p      (promise)]  
  (thread #(deliver p 5))  
  (then-accept p (fn [v] (deliver result (+ v 2)))))  
  [@p @result]))  
=> [5 7]
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[then-compose](#)

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completest normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

then-accept-both

```
(then-accept-both p p-other f)
```

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two results as arguments.

```
(-> (promise (fn [] (sleep 200) "The quick brown fox"))
    (then-accept-both (promise (fn [] (sleep 100) "jumps over the lazy dog"))
                      (fn [u v] (println (pr-str (str u " " v))))))
    (deref))
"The quick brown fox jumps over the lazy dog"
=> nil
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[then-compose](#)

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

then-apply

```
(then-apply p f)
```

Applies a function `f` on the result of the previous stage of the promise `p`.


```
(-> (promise (fn [] "the quick brown fox"))
    (then-apply str/upper-case)
    (then-apply #(str % " jumps over the lazy dog"))
    (deref))
=> "THE QUICK BROWN FOX jumps over the lazy dog"
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[then-compose](#)

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

then-combine

```
(then-combine p p-other f)
```

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

```
(-> (promise (fn [] "The Quick Brown Fox"))
    (then-apply str/upper-case)
    (then-combine (-> (promise (fn [] "Jumps Over The Lazy Dog"))
                      (then-apply str/lower-case))
                  #(str %1 " " %2))
    (deref))
=> "THE QUICK BROWN FOX jumps over the lazy dog"
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[then-compose](#)

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

then-compose

```
(then-compose p f)
```

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value with this promise.

```
(-> (promise (fn [] "The Quick Brown Fox"))
    (then-apply str/upper-case)
    (then-compose (fn [x] (-> (promise (fn [] "Jumps Over The Lazy Dog"))
                              (then-apply str/lower-case)
                              (then-apply #(str x " " %1)))))
    (deref))
=> "THE QUICK BROWN FOX jumps over the lazy dog"
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

`complete-on-timeout`

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

third

```
(third coll)
```

Returns the third element of coll.

```
(third nil)
=> nil
```

```
(third [])
=> nil
```

```
(third [1 2 3])
=> 3
```

```
(third '())
=> nil
```

```
(third '(1 2 3))
=> 3
```

[section](#)[top](#)

thread

```
(thread f)
(thread f name)
(thread f name type)
```

Executes the function `f` in another thread, returning immediately to the calling thread. Returns a `promise` which will receive the result of calling the function `f` when completed. Optionally a name can be assigned to the spawned thread.

The thread can be given a name by passing the *name* argument. By default the thread name is set to "venice-thread". For each thread spawned on a name the thread's name will be suffixed with an incrementing index starting from 1.

The thread type (*daemon* or *user*) can be controlled by the *type* argument that must be one of `{:daemon, :user}`. By default a daemon thread is spawned.

Note: Each call to `thread` creates a new expensive system thread. Consider to use futures or promises that use an *ExecutorService* to deal efficiently with threads.

```
@(thread #(do (sleep 100) 1))
=> 1
```

```
@(thread #(do (sleep 100) (thread-name)))
=> "venice-thread-3"
```

```
@(thread #(do (sleep 100) (thread-name)) "job")
=> "job-1"
```

```
@(thread #(do (sleep 100) (thread-name)) "job" :daemon)
=> "job-2"

;; consumer / producer
(do
  (defn produce [q]
    (doseq [x (range 4)] (put! q x) (sleep 100))
    (put! q nil))
  (defn consume [q]
    (transduce (map println) (constantly nil) q))
  (let [q (queue 10)]
    (thread #(produce q))
    @(thread #(consume q))))

0
1
2
3
=> nil
```

SEE ALSO

[future](#)

Takes a function without arguments and yields a future object that will invoke the function in another thread, and will cache the result ...

[promise](#)

Returns a promise object that can be read with `deref`, and `set`, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[agent](#)

Creates and returns an agent with an initial value of state and zero or more options.

[section](#)

[top](#)

thread-daemon?

```
(thread-daemon?)
```

Returns true if this Thread is a daemon thread else false.

```
(thread-daemon?)
=> false
```

SEE ALSO

[thread-name](#)

Returns this thread's name.

[section](#)

[top](#)

thread-id

```
(thread-id)
```

Returns the identifier of this Thread. The thread ID is a positive number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime. When a thread is terminated, this thread ID may be reused.

```
(thread-id)  
=> 1
```

SEE ALSO

[thread-name](#)

Returns this thread's name.

[section](#)

[top](#)

thread-interrupted

```
(thread-interrupted)
```

Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it).

Returns true if the current thread has been interrupted else false.

```
(thread-interrupted)  
=> false
```

SEE ALSO

[thread-interrupted?](#)

Tests whether this thread has been interrupted. The interrupted status of the thread is unaffected by this method. Returns true if ...

[section](#)

[top](#)

thread-interrupted?

```
(thread-interrupted?)
```

Tests whether this thread has been interrupted. The interrupted status of the thread is unaffected by this method. Returns true if the current thread has been interrupted else false.

```
(thread-interrupted?)  
=> false
```

SEE ALSO

[thread-interrupted](#)

Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method. In other words, ...

[section](#)

[top](#)

thread-local

```
(thread-local)
```

Creates a new thread-local accessor

```
(do
  (assoc! (thread-local) :a 1)
  (get (thread-local) :a))
=> 1

(do
  (assoc! (thread-local) :a 1)
  (get (thread-local) :b 999))
=> 999

(do
  (thread-local :a 1 :b 2)
  (get (thread-local) :a))
=> 1

(do
  (thread-local { :a 1 :b 2 })
  (get (thread-local) :a))
=> 1

(do
  (thread-local-clear)
  (assoc! (thread-local) :a 1 :b 2)
  (dissoc! (thread-local) :a)
  (get (thread-local) :a 999))
=> 999
```

SEE ALSO

[thread-local-clear](#)

Removes all thread local vars

[thread-local-map](#)

Returns a snapshot of the thread local vars as a map.

[assoc!](#)

Associates key/vals with a mutable map, returns the map

[dissoc!](#)

Dissociates keys from a mutable map, returns the map

[get](#)

Returns the value mapped to key, not-found or nil if key not present.

[section](#)

[top](#)

thread-local-clear

```
(thread-local-clear)
```

Removes all thread local vars

```
(thread-local-clear)
=> thread-local-clear
```

SEE ALSO

[thread-local](#)

Creates a new thread-local accessor

[dissoc!](#)

Dissociates keys from a mutable map, returns the map

[section](#)

[top](#)

thread-local-map

```
(thread-local-map)
```

Returns a snapshot of the thread local vars as a map.

Note:

The returned map is a copy of the current thread local vars. Thus modifying this map is not modifying the thread local vars! Use `assoc!` and `dissoc!` for that purpose!

```
(do
  (thread-local-clear)
  (thread-local :a 1 :b 2)
  (thread-local-map))
=> {:a 1 :b 2 :*assertions* (0)}
```

SEE ALSO

[thread-local](#)

Creates a new thread-local accessor

[get](#)

Returns the value mapped to key, not-found or nil if key not present.

[assoc!](#)

Associates key/vals with a mutable map, returns the map

[dissoc!](#)

Dissociates keys from a mutable map, returns the map

[section](#)

[top](#)

thread-local?

```
(thread-local? x)
```

Returns true if x is a thread-local, otherwise false

```
(do
  (def x (thread-local))
  (thread-local? x))
=> true
```

SEE ALSO

[thread-local](#)

Creates a new thread-local accessor

[section](#)

[top](#)

thread-name

```
(thread-name)
```

Returns this thread's name.

```
(thread-name)
=> "main"
```

SEE ALSO

[thread-id](#)

Returns the identifier of this Thread. The thread ID is a positive number generated when this thread was created. The thread ID is ...

[section](#)

[top](#)

throw

```
(throw)
(throw val)
(throw ex)
```

Throws an exception.

```
(throw)
```

Throws a `:ValueException` with `nil` as its value.

```
(throw val)
```

With `val` as a Venice value throws a `:ValueException` with `val` as its value.

E.g: `(throw [1 2 3])`

```
(throw ex)
```

With a `ex` as an exception type throws the exception.

E.g: `(throw (ex :VncException "invalid data"))`

```
(try
  (+ 100 200)
  (catch :Exception e
    "caught ~(ex-message e)"))
=> 300
```

```
(try
  (+ 100 200)
  (throw)
  (catch :ValueException e
    "caught ~(pr-str (ex-value e))"))
=> "caught nil"
```

```
(try
  (+ 100 200)
  (throw 100)
  (catch :ValueException e
    "caught ~(ex-value e)"))
=> "caught 100"
```

;; The finally block is just for side effects, like
;; closing resources. It never returns a value!

```
(try
  (+ 100 200)
  (throw [100 {:a 3}])
  (catch :ValueException e
```



```

        "caught ~(ex-value e)")
    (finally (println "#finally")
              :finally))
#finally
=> "caught [100 {:a 3}]"

(try
  (throw (ex :RuntimeException "#test")))
(catch :RuntimeException e
      "caught ~(ex-message e)")
=> "caught #test"

;; Venice wraps thrown checked exceptions with a RuntimeException!
(do
  (import :java.io.IOException)
  (try
    (throw (ex :IOException "#test"))
    (catch :RuntimeException e
          "caught ~(ex-message (ex-cause e))")))
=> "caught #test"

```

SEE ALSO

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of `:java.lang.Exception`

[try](#)

Exception handling: try - catch - finally

[try-with](#)

try-with-resources allows the declaration of resources to be used in a try block with the assurance that the resources will be closed ...

[section](#)

[top](#)

time

```
(time expr)
```

Evaluates `expr` and prints the time it took. Returns the value of `expr`.

```

(time (+ 100 200))
Elapsed time: 4.54µs
=> 300

```

SEE ALSO

[dorun](#)

Runs the `expr` count times in the most effective way. It's main purpose is supporting benchmark tests. Returns the expression result ...

[section](#)

[top](#)

time/after?

```

(time/after? date1 date2)
(time/after? date1 date2 & more)

```

Returns true if all dates are ordered from the latest to the earliest (same semantics as `>`)

```
(time/after? (time/local-date 2019 1 1)
             (time/local-date 2018 1 1))
=> true

(time/after? (time/local-date-time "2019-01-01T10:00:00.000")
             (time/local-date-time "2018-01-01T10:00:00.000"))
=> true

(time/after? (time/zoned-date-time "2019-01-01T10:00:00.000+01:00")
             (time/zoned-date-time "2018-01-01T10:00:00.000+01:00"))
=> true
```

SEE ALSO

[time/before?](#)

Returns true if all dates are ordered from the earliest to the latest (same semantics as <)

[time/not-after?](#)

Returns true if date1 is not-after date2 else false (same semantics as <=)

[time/not-before?](#)

Returns true if date1 is not-before date2 else false (same semantics as >=)

[section](#)

[top](#)

time/before?

```
(time/before? date1 date2)
(time/before? date1 date2 & more)
```

Returns true if all dates are ordered from the earliest to the latest (same semantics as )

```
(time/before? (time/local-date 2018 1 1)
              (time/local-date 2019 1 1))
=> true

(time/before? (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-01-01T10:00:00.000"))
=> true

(time/before? (time/zoned-date-time "2018-01-01T10:00:00.000+01:00")
              (time/zoned-date-time "2019-01-01T10:00:00.000+01:00"))
=> true
```

SEE ALSO

[time/after?](#)

Returns true if all dates are ordered from the latest to the earliest (same semantics as >)

[time/not-after?](#)

Returns true if date1 is not-after date2 else false (same semantics as <=)

[time/not-before?](#)

Returns true if date1 is not-before date2 else false (same semantics as >=)

[section](#)

[top](#)

time/between

```
(time/between date1 date2 unit)
```

Calculates the amount of time between two date/time values. Unit is one of :millis, :seconds, :minutes, :hours, :days, :weeks, :months, :years.

Note: the units :millis, :seconds, :minutes, and :hours are not supported for local-date type.

```
(time/between (time/local-date 2018 1 1)
              (time/local-date 2019 1 1)
              :days)
=> 365

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :seconds)
=> 36633600

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :minutes)
=> 610560

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :hours)
=> 10176

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :days)
=> 424

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :weeks)
=> 60

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :months)
=> 14

(time/between (time/local-date-time "2018-01-01T10:00:00.000")
              (time/local-date-time "2019-03-01T10:00:00.000")
              :years)
=> 1

(time/between (time/zoned-date-time "2018-01-01T10:00:00.000+01:00")
              (time/zoned-date-time "2019-03-01T10:00:00.000+01:00")
              :months)
=> 14
```

SEE ALSO

[time/after?](#)

Returns true if all dates are ordered from the latest to the earliest (same semantics as >)

[time/before?](#)

Returns true if all dates are ordered from the earliest to the latest (same semantics as <)

[time/not-after?](#)

Returns true if date1 is not-after date2 else false (same semantics as <=)

[time/not-before?](#)

Returns true if date1 is not-before date2 else false (same semantics as >=)

[section](#)

[top](#)

time/date

```
(time/date)
(time/date x)
```

Creates a new date of type 'java.util.Date'. x can be a long representing milliseconds since the epoch, a 'java.time.LocalDate', a 'java.time.LocalDateTime', or a 'java.time.ZonedDateTime'

```
(time/date)
=> Thu Jun 04 17:21:08 CEST 2026
```

[section](#)

[top](#)

time/date?

```
(time/date? date)
```

Returns true if date is a 'java.util.Date' else false

```
(time/date? (time/date))
=> true
```

[section](#)

[top](#)

time/day-of-month

```
(time/day-of-month date)
```

Returns the day of the month (1..31)

```
(time/day-of-month (time/local-date))
=> 4
```

```
(time/day-of-month (time/local-date-time))
=> 4
```

```
(time/day-of-month (time/zoned-date-time))
=> 4
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/last-day-of-month](#)

Returns the last day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

[section](#)

[top](#)

time/day-of-week

```
(time/day-of-week date)
```

Returns the day of the week (:MONDAY ... :SUNDAY)

```
(time/day-of-week (time/local-date))  
=> :THURSDAY
```

```
(time/day-of-week (time/local-date-time))  
=> :THURSDAY
```

```
(time/day-of-week (time/zoned-date-time))  
=> :THURSDAY
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/last-day-of-month](#)

Returns the last day of a month as a local-date.

[section](#)

[top](#)

time/day-of-year

```
(time/day-of-year date)
```

Returns the day of the year (1..366)

```
(time/day-of-year (time/local-date))  
=> 155  
  
(time/day-of-year (time/local-date-time))  
=> 155  
  
(time/day-of-year (time/zoned-date-time))  
=> 155
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/last-day-of-month](#)

Returns the last day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

[section](#)

[top](#)

time/earliest

```
(time/earliest coll)
```

Returns the earliest date from a collection of dates. All dates must be of equal type. The coll may be empty or nil.

```
(time/earliest [(time/local-date 2018 8 4) (time/local-date 2018 8 3)])  
=> 2018-08-03
```

[section](#)

[top](#)

time/first-day-of-month

```
(time/first-day-of-month date)
```

Returns the first day of a month as a local-date.

```
(time/first-day-of-month (time/local-date))  
=> 2026-06-01
```

```
(time/first-day-of-month (time/local-date-time))  
=> 2026-06-01
```

```
(time/first-day-of-month (time/zoned-date-time))  
=> 2026-06-01
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/last-day-of-month](#)

Returns the last day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

[section](#)

[top](#)

time/first-day-of-month?

```
(time/first-day-of-month? date)
```

Returns `true` if the date is the first day of a month otherwise `false`.

```
(time/first-day-of-month? (time/local-date))
```

```
=> false
```

```
(time/first-day-of-month? (time/local-date-time))
```

```
=> false
```

```
(time/first-day-of-month? (time/zoned-date-time))
```

```
=> false
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

[section](#)

[top](#)

time/format

```
(time/format date format)
(time/format date format locale)
(time/format date formatter)
(time/format date formatter locale)
```

Formats a date with a format.

To format a large number of dates a pre instantiated formatter delivers best performance:

```
(let [fmt (time/formatter "yyyy-MM-dd'T'HH:mm:ss")]
  (dotimes [n 100] (time/format (time/local-date-time) fmt)))
```

```
(time/format (time/local-date) "dd-MM-yyyy")
=> "04-06-2026"
```

```
(time/format (time/local-date) (time/formatter "dd-MM-yyyy"))
=> "04-06-2026"
```

```
(time/format (time/local-date) :iso)
=> "2026-06-04"
```

```
(time/format (time/local-date-time) "yyyy-MM-dd'T'HH:mm:ss")
=> "2026-06-04T17:21:11"
```

```
(time/format (time/local-date-time) (time/formatter "yyyy-MM-dd'T'HH:mm:ss"))
=> "2026-06-04T17:21:11"
```

```
(time/format (time/local-date-time) :iso)
=> "2026-06-04T17:21:11.866"
```

```
(time/format (time/zoned-date-time) "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> "2026-06-04T17:21:11.901CEST"
```

```
(time/format (time/zoned-date-time) :iso)
=> "2026-06-04T17:21:11.936+02:00"
```

```
(time/format (time/zoned-date-time) (time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz"))
=> "2026-06-04T17:21:12.003CEST"
```

SEE ALSO

[time/formatter](#)

Creates a formatter

[section](#)

[top](#)

time/formatter

```
(time/formatter format)
(time/formatter format locale)
```

Creates a formatter

```
(time/formatter "dd-MM-yyyy")
```

```
(time/formatter "dd-MM-yyyy" :en_EN)
```



```
(time/formatter "dd-MM-yyyy" "en_EN")
```

```
(time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz")
```

```
(time/formatter :ISO_OFFSET_DATE_TIME)
```

SEE ALSO

[time/format](#)

Formats a date with a format.

[section](#)

[top](#)

time/hour

```
(time/hour date)
```

Returns the hour of the date 0..23

```
(time/hour (time/local-date))
```

```
=> 0
```

```
(time/hour (time/local-date-time))
```

```
=> 17
```

```
(time/hour (time/zoned-date-time))
```

```
=> 17
```

SEE ALSO

[time/minute](#)

Returns the minute of the date 0..59

[time/second](#)

Returns the second of the date 0..59

[time/milli](#)

Returns the millis of the date 0..999

[section](#)

[top](#)

time/last-day-of-month

```
(time/last-day-of-month date)
```

Returns the last day of a month as a local-date.

```
(time/last-day-of-month (time/local-date))
```

```
=> 2026-06-30
```

```
(time/last-day-of-month (time/local-date-time))
```

```
=> 2026-06-30
```

```
(time/last-day-of-month (time/zoned-date-time))
```

```
=> 2026-06-30
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

[section](#)

[top](#)

time/last-day-of-month?

```
(time/last-day-of-month? date)
```

Returns `true` if the date is the last day of a month otherwise `false`.

```
(time/last-day-of-month? (time/local-date))  
=> false
```

```
(time/last-day-of-month? (time/local-date-time))  
=> false
```

```
(time/last-day-of-month? (time/zoned-date-time))  
=> false
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

[section](#)

[top](#)

time/latest

```
(time/latest coll)
```

Returns the latest date from a collection of dates. All dates must be of equal type. The coll may be empty or nil.

```
(time/latest [(time/local-date 2018 8 1) (time/local-date 2018 8 3)])  
=> 2018-08-03
```

[section](#)[top](#)

time/leap-year?

```
(time/leap-year? date)
```

Checks if the year is a leap year.

```
(time/leap-year? 2000)  
=> true  
  
(time/leap-year? (time/local-date 2000 1 1))  
=> true  
  
(time/leap-year? (time/local-date-time))  
=> false  
  
(time/leap-year? (time/zoned-date-time))  
=> false
```

SEE ALSO

[time/length-of-year](#)

Returns the length of the year represented by this date.

[time/length-of-month](#)

Returns the length of the month represented by this date.

[section](#)[top](#)

time/length-of-month

```
(time/length-of-month date)
```

Returns the length of the month represented by this date.

This returns the length of the month in days. For example, a date in January would return 31.

```
(time/length-of-month (time/local-date 2000 2 1))  
=> 29  
  
(time/length-of-month (time/local-date 2001 2 1))  
=> 28  
  
(time/length-of-month (time/local-date-time))  
=> 30
```

```
(time/length-of-month (time/zoned-date-time))  
=> 30
```

SEE ALSO

[time/length-of-year](#)

Returns the length of the year represented by this date.

[time/leap-year?](#)

Checks if the year is a leap year.

[section](#)

[top](#)

time/length-of-year

```
(time/length-of-year date)
```

Returns the length of the year represented by this date.

This returns the length of the year in days, either 365 or 366.

```
(time/length-of-year (time/local-date 2000 1 1))  
=> 366
```

```
(time/length-of-year (time/local-date 2001 1 1))  
=> 365
```

```
(time/length-of-year (time/local-date-time))  
=> 365
```

```
(time/length-of-year (time/zoned-date-time))  
=> 365
```

SEE ALSO

[time/length-of-month](#)

Returns the length of the month represented by this date.

[time/leap-year?](#)

Checks if the year is a leap year.

[section](#)

[top](#)

time/local-date

```
(time/local-date)  
(time/local-date year month day)  
(time/local-date date)
```

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

```
(time/local-date)  
=> 2026-06-04
```

```
(time/local-date 2018 8 1)  
=> 2018-08-01
```

```
(time/local-date "2018-08-01")
=> 2018-08-01

(time/local-date (time/local-date-time 2018 8 1 14 20 10))
=> 2018-08-01

(time/local-date 1375315200000)
=> 2013-08-01

(time/local-date (. :java.util.Date :new))
=> 2026-06-04
```

SEE ALSO

[time/local-date-time](#)

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

[time/zoned-date-time](#)

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

[section](#)[top](#)

time/local-date-parse

```
(time/local-date-parse str format
(time/local-date-parse str format locale
```

Parses a local-date.

To parse a large number of dates a pre instantiated formatter delivers best performance:

```
(let [fmt (time/formatter "yyyy-MM-dd")]
(dotimes [n 100] (time/local-date-parse "2018-12-01" fmt)))
```

```
(time/local-date-parse "2018-12-01" "yyyy-MM-dd")
=> 2018-12-01
```

```
(time/local-date-parse "2018-Dec-01" "yyyy-MMM-dd" :ENGLISH)
=> 2018-12-01
```

```
(time/local-date-parse "2018-12-01" :iso)
=> 2018-12-01
```

[section](#)[top](#)

time/local-date-time

```
(time/local-date-time)
(time/local-date-time year month day)
(time/local-date-time year month day hour minute second)
(time/local-date-time year month day hour minute second millis)
(time/local-date-time date)
```

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

```

(time/local-date-time)
=> 2026-06-04T17:21:08.564

(time/local-date-time 2018 8 1)
=> 2018-08-01T00:00

(time/local-date-time 2018 8 1 14 20 10)
=> 2018-08-01T14:20:10

(time/local-date-time 2018 8 1 14 20 10 200)
=> 2018-08-01T14:20:10.200

(time/local-date-time "2018-08-01T14:20:10.200")
=> 2018-08-01T14:20:10.200

(time/local-date-time (time/local-date 2018 8 1))
=> 2018-08-01T00:00

(time/local-date-time 1375315200000)
=> 2013-08-01T02:00

(time/local-date-time (. :java.util.Date :new))
=> 2026-06-04T17:21:08.807

```

SEE ALSO

[time/local-date](#)

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

[time/zoned-date-time](#)

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

[section](#)

[top](#)

time/local-date-time-parse

```

(time/local-date-time-parse str format
(time/local-date-time-parse str format locale

```

Parses a local-date-time.

To parse a large number of dates a pre instantiated formatter delivers best performance:

```

(let [fmt (time/formatter "yyyy-MM-dd HH:mm:ss")]
  (dotimes [n 100] (time/local-date-time-parse "2018-12-01 14:20:01" fmt)))

```

```

(time/local-date-time-parse "2018-08-01 14:20" "yyyy-MM-dd HH:mm")
=> 2018-08-01T14:20

```

```

(time/local-date-time-parse "2018-08-01 14:20:01.231" "yyyy-MM-dd HH:mm:ss.SSS")
=> 2018-08-01T14:20:01.231

```

```

(time/local-date-time-parse "2018-08-01T14:20:01.231" :iso)
=> 2018-08-01T14:20:01.231

```

[section](#)

[top](#)

time/local-date-time?

```
(time/local-date-time? date)
```

Returns true if date is a local-date-time ('java.time.LocalDateTime') else false

```
(time/local-date-time? (time/local-date-time))  
=> true
```

[section](#)[top](#)

time/local-date?

```
(time/local-date? date)
```

Returns true if date is a locale date ('java.time.LocalDate') else false

```
(time/local-date? (time/local-date))  
=> true
```

[section](#)[top](#)

time/milli

```
(time/milli date)
```

Returns the millis of the date 0..999

```
(time/milli (time/local-date))  
=> 0
```

```
(time/milli (time/local-date-time))  
=> 34
```

```
(time/milli (time/zoned-date-time))  
=> 68
```

SEE ALSO

[time/hour](#)

Returns the hour of the date 0..23

[time/minute](#)

Returns the minute of the date 0..59

[time/second](#)

Returns the second of the date 0..59

[section](#)[top](#)

time/minus

```
(time/minus date unit n)
(time/minus date temporal)
```

Subtracts the n units from the date. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

In the two argument version subtracts a java.time.Temporal (Period, Duration) from the date.

```
(time/minus (time/local-date) :days 2)
=> 2026-06-02
```

```
(time/minus (time/local-date-time) :days 2)
=> 2026-06-02T17:21:13.659
```

```
(time/minus (time/zoned-date-time) :days 2)
=> 2026-06-02T17:21:13.694+02:00[Europe/Zurich]
```

```
(time/minus (time/local-date) (. :java.time.Period :ofDays 2))
=> 2026-06-02
```

```
(time/minus (time/local-date-time) (. :java.time.Period :ofDays 2))
=> 2026-06-02T17:21:13.763
```

```
(time/minus (time/zoned-date-time) (. :java.time.Period :ofDays 2))
=> 2026-06-02T17:21:13.798+02:00[Europe/Zurich]
```

SEE ALSO

[time/plus](#)

Adds the n units to the date. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

[section](#)

[top](#)

time/minute

```
(time/minute date)
```

Returns the minute of the date 0..59

```
(time/minute (time/local-date))
=> 0
```

```
(time/minute (time/local-date-time))
=> 21
```

```
(time/minute (time/zoned-date-time))
=> 21
```

SEE ALSO

[time/hour](#)

Returns the hour of the date 0..23

[time/second](#)

Returns the second of the date 0..59

[time/milli](#)

Returns the millis of the date 0..999

time/month

```
(time/month date)
```

Returns the month of the date 1..12

```
(time/month (time/local-date))
```

```
=> 6
```

```
(time/month (time/local-date-time))
```

```
=> 6
```

```
(time/month (time/zoned-date-time))
```

```
=> 6
```

SEE ALSO

[time/year](#)

Returns the year of the date

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/last-day-of-month](#)

Returns the last day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

time/not-after?

```
(time/not-after? date1 date2)
```

Returns true if date1 is not-after date2 else false (same semantics as `<=`)

```
(time/not-after? (time/local-date 2018 1 1)
                 (time/local-date 2019 1 1))
```

```
=> true
```

```
(time/not-after? (time/local-date-time "2018-01-01T10:00:00.000")
                 (time/local-date-time "2019-01-01T10:00:00.000"))
```

```
=> true
```

```
(time/not-after? (time/zoned-date-time "2018-01-01T10:00:00.000+01:00")
                 (time/zoned-date-time "2019-01-01T10:00:00.000+01:00"))
```

```
=> true
```

SEE ALSO

[time/after?](#)

Returns true if all dates are ordered from the latest to the earliest (same semantics as >)

[time/before?](#)

Returns true if all dates are ordered from the earliest to the latest (same semantics as <)

[time/not-before?](#)

Returns true if date1 is not-before date2 else false (same semantics as >=)

[section](#)[top](#)

time/not-before?

```
(time/not-before? date1 date2)
```

Returns true if date1 is not-before date2 else false (same semantics as >=)

```
(time/not-before? (time/local-date 2019 1 1)
                  (time/local-date 2019 1 1))
```

```
=> true
```

```
(time/not-before? (time/local-date-time "2019-01-01T10:00:00.000")
                  (time/local-date-time "2018-01-01T10:00:00.000"))
```

```
=> true
```

```
(time/not-before? (time/zoned-date-time "2019-01-01T10:00:00.000+01:00")
                  (time/zoned-date-time "2018-01-01T10:00:00.000+01:00"))
```

```
=> true
```

SEE ALSO

[time/after?](#)

Returns true if all dates are ordered from the latest to the earliest (same semantics as >)

[time/before?](#)

Returns true if all dates are ordered from the earliest to the latest (same semantics as <)

[time/not-after?](#)

Returns true if date1 is not-after date2 else false (same semantics as <=)

[section](#)[top](#)

time/period

```
(time/period from to unit)
```

Returns the period interval of two dates in the specified unit.

Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(time/period (time/local-date) (time/plus (time/local-date) :days 3) :days)
```

```
=> 3
```

```
(time/period (time/local-date-time) (time/plus (time/local-date-time) :days 3) :days)
```

```
=> 3
```

```
(time/period (time/zoned-date-time) (time/plus (time/zoned-date-time) :days 3) :days)
=> 3
```

SEE ALSO

[time/local-date](#)

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

[time/local-date-time](#)

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

[time/zoned-date-time](#)

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

[section](#)[top](#)

time/plus

```
(time/plus date unit n)
(time/minus plus temporal)
```

Adds the n units to the date. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

In the two argument version add a java.time.Temporal (Period, Duration) to the date.

```
(time/plus (time/local-date) :days 2)
=> 2026-06-06
```

```
(time/plus (time/local-date-time) :days 2)
=> 2026-06-06T17:21:13.448
```

```
(time/plus (time/zoned-date-time) :days 2)
=> 2026-06-06T17:21:13.483+02:00[Europe/Zurich]
```

```
(time/plus (time/local-date) (. :java.time.Period :ofDays 2))
=> 2026-06-06
```

```
(time/plus (time/local-date-time) (. :java.time.Period :ofDays 2))
=> 2026-06-06T17:21:13.553
```

```
(time/plus (time/zoned-date-time) (. :java.time.Period :ofDays 2))
=> 2026-06-06T17:21:13.587+02:00[Europe/Zurich]
```

SEE ALSO

[time/minus](#)

Subtracts the n units from the date. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

[section](#)[top](#)

time/second

```
(time/second date)
```

Returns the second of the date 0..59

```
(time/second (time/local-date))  
=> 0
```

```
(time/second (time/local-date-time))  
=> 10
```

```
(time/second (time/zoned-date-time))  
=> 10
```

SEE ALSO

[time/hour](#)

Returns the hour of the date 0..23

[time/minute](#)

Returns the minute of the date 0..59

[time/milli](#)

Returns the millis of the date 0..999

[section](#)

[top](#)

time/to-millis

```
(time/to-millis date)
```

Converts the passed date to milliseconds since epoch

```
(time/to-millis (time/date))  
=> 1780586474062
```

```
(time/to-millis (time/local-date))  
=> 1780524000000
```

```
(time/to-millis (time/local-date-time))  
=> 1780586474131
```

```
(time/to-millis (time/zoned-date-time))  
=> 1780586474166
```

[section](#)

[top](#)

time/unix-timestamp

```
(time/unix-timestamp)  
(time/unix-timestamp year month day)  
(time/unix-timestamp year month day hour minute second)  
(time/unix-timestamp year month day hour minute second millis)  
(time/unix-timestamp date)
```

Returns a unix timestamp. Seconds since Jan 01 1970 (UTC).

See: [Unix Timestamp](#)

```
(time/unix-timestamp)  
=> 1780593669
```

```
(time/unix-timestamp 2018 8 1)
=> 1533081600

(time/unix-timestamp 2018 8 1 14 20 10)
=> 1533133210

(time/unix-timestamp 2018 8 1 14 20 10 200)
=> 1533133210

(time/unix-timestamp "2018-08-01T14:20:10.200")
=> 2018-08-01T14:20:10.200

(time/unix-timestamp (time/local-date-time))
=> 1780593669

(time/unix-timestamp (time/local-date 2018 8 1))
=> 1533081600

(time/unix-timestamp (. :java.util.Date :new))
=> 1780593670
```

SEE ALSO

[time/unix-timestamp-to-local-date-time](#)

Converts a unix timestamp (seconds since Jan 01 1970 (UTC)) to a java :LocalDateTime.

[time/local-date-time](#)

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

[time/local-date](#)

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

[time/zoned-date-time](#)

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

[section](#)

[top](#)

time/unix-timestamp-to-local-date-time

```
(time/unix-timestamp-to-local-date-time seconds-since-epoch)
```

Converts a unix timestamp (seconds since Jan 01 1970 (UTC)) to a java :LocalDateTime.

See: [Unix Timestamp](#)

```
(time/unix-timestamp-to-local-date-time (time/unix-timestamp))
=> 2026-06-04T17:21:10
```

SEE ALSO

[time/unix-timestamp](#)

Returns a unix timestamp. Seconds since Jan 01 1970 (UTC).

[section](#)

[top](#)

time/with-time

```
(time/with-time date hour minute second)
(time/with-time date hour minute second millis)
```

Sets the time of a date. Returns a new date

```
(time/with-time (time/local-date) 22 00 15 333)
=> 2026-06-04T22:00:15.333
```

```
(time/with-time (time/local-date-time) 22 00 15 333)
=> 2026-06-04T22:00:15.333
```

```
(time/with-time (time/zoned-date-time) 22 00 15 333)
=> 2026-06-04T22:00:15.333+02:00[Europe/Zurich]
```

SEE ALSO

[time/local-date](#)

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

[time/local-date-time](#)

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

[time/zoned-date-time](#)

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

[section](#)

[top](#)

time/within?

```
(time/within? date start end)
```

Returns true if the date is after or equal to the start and is before or equal to the end. All three dates must be of the same type. The start and end date may each be nil meaning start is -infinity and end is +infinity. (same semantics as `start <= date <= end`)

```
(time/within? (time/local-date 2018 8 15)
              (time/local-date 2018 8 10)
              (time/local-date 2018 8 20))
=> true
```

```
(time/within? (time/local-date 2018 8 25)
              (time/local-date 2018 8 10)
              (time/local-date 2018 8 20))
=> false
```

```
(time/within? (time/local-date 2018 8 20)
              (time/local-date 2018 8 10)
              nil)
=> true
```

```
(time/within? (time/local-date-time "2019-01-01T10:00:00.000")
              (time/local-date-time "2010-01-01T10:00:00.000")
              (time/local-date-time "2020-01-01T10:00:00.000"))
=> true
```

```
(time/within? (time/zoned-date-time "2010-01-01T10:00:00.000+01:00")
              (time/zoned-date-time "2019-01-01T10:00:00.000+01:00")
              (time/zoned-date-time "2020-01-01T10:00:00.000+01:00"))
=> false
```

time/year

```
(time/year date)
```

Returns the year of the date

```
(time/year (time/local-date))
=> 2026
```

```
(time/year (time/local-date-time))
=> 2026
```

```
(time/year (time/zoned-date-time))
=> 2026
```

SEE ALSO

[time/month](#)

Returns the month of the date 1..12

[time/day-of-year](#)

Returns the day of the year (1..366)

[time/day-of-month](#)

Returns the day of the month (1..31)

[time/first-day-of-month](#)

Returns the first day of a month as a local-date.

[time/last-day-of-month](#)

Returns the last day of a month as a local-date.

[time/day-of-week](#)

Returns the day of the week (:MONDAY ... :SUNDAY)

time/zone

```
(time/zone date)
```

Returns the zone of the date

```
(time/zone (time/zoned-date-time))
=> "Europe/Zurich"
```

time/zone-ids

```
(time/zone-ids)
```

Returns all available zone ids with time offset

```
(nfirst (seq (time/zone-ids)) 10)
=> ([ "Africa/Abidjan" "+00:00" ] [ "Africa/Accra" "+00:00" ] [ "Africa/Addis_Ababa" "+03:00" ] [ "Africa/Algiers"
"+01:00" ] [ "Africa/Asmara" "+03:00" ] [ "Africa/Asmera" "+03:00" ] [ "Africa/Bamako" "+00:00" ] [ "Africa/Bangui"
"+01:00" ] [ "Africa/Banjul" "+00:00" ] [ "Africa/Bissau" "+00:00" ] )
```

[section](#)

[top](#)

time/zone-offset

(time/zone-offset date)

Returns the zone-offset of the date in minutes

```
(time/zone-offset (time/zoned-date-time))
=> 120
```

SEE ALSO

[time/zoned-date-time](#)

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

[section](#)

[top](#)

time/zoned-date-time

```
(time/zoned-date-time)
(time/zoned-date-time year month day)
(time/zoned-date-time year month day hour minute second)
(time/zoned-date-time year month day hour minute second millis)
(time/zoned-date-time date)
(time/zoned-date-time zone-id)
(time/zoned-date-time zone-id year month day)
(time/zoned-date-time zone-id year month day hour minute second)
(time/zoned-date-time zone-id year month day hour minute second millis)
(time/zoned-date-time zone-id date)
```

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

```
(time/zoned-date-time)
=> 2026-06-04T17:21:08.996+02:00[Europe/Zurich]
```

```
(time/zoned-date-time 2018 8 1)
=> 2018-08-01T00:00+02:00[Europe/Zurich]
```

```
(time/zoned-date-time 2018 8 1 14 20 10)
=> 2018-08-01T14:20:10+02:00[Europe/Zurich]
```

```
(time/zoned-date-time 2018 8 1 14 20 10 200)
=> 2018-08-01T14:20:10.200+02:00[Europe/Zurich]
```

```
(time/zoned-date-time "2018-08-01T14:20:10.200+01:00")
=> 2018-08-01T14:20:10.200+01:00
```



```

(time/zoned-date-time (time/local-date 2018 8 1))
=> 2018-08-01T00:00+02:00[Europe/Zurich]

(time/zoned-date-time (time/local-date-time 2018 8 1 14 20 10))
=> 2018-08-01T14:20:10+02:00[Europe/Zurich]

(time/zoned-date-time 1375315200000)
=> 2013-08-01T02:00+02:00[Europe/Zurich]

(time/zoned-date-time (. :java.util.Date :new))
=> 2026-06-04T17:21:09.275+02:00[Europe/Zurich]

(time/zoned-date-time "UTC")
=> 2026-06-04T15:21:09.309Z[UTC]

(time/zoned-date-time "UTC" 2018 8 1)
=> 2018-08-01T00:00Z[UTC]

(time/zoned-date-time "UTC" 2018 8 1 14 20 10)
=> 2018-08-01T14:20:10Z[UTC]

(time/zoned-date-time "UTC" 2018 8 1 14 20 10 200)
=> 2018-08-01T14:20:10.200Z[UTC]

(time/zoned-date-time "UTC" "2018-08-01T14:20:10.200+01:00")
=> 2018-08-01T14:20:10.200Z[UTC]

(time/zoned-date-time "UTC" (time/local-date 2018 8 1))
=> 2018-08-01T00:00Z[UTC]

(time/zoned-date-time "UTC" (time/local-date-time 2018 8 1 14 20 10))
=> 2018-08-01T14:20:10Z[UTC]

(time/zoned-date-time "UTC" 1375315200000)
=> 2013-08-01T00:00Z[UTC]

(time/zoned-date-time "UTC" (. :java.util.Date :new))
=> 2026-06-04T15:21:09.602Z[UTC]

```

SEE ALSO

[time/local-date](#)

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

[time/local-date-time](#)

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

[section](#)

[top](#)

time/zoned-date-time-parse

```

(time/zoned-date-time-parse str format
(time/zoned-date-time-parse str format locale

```

Parses a zoned-date-time.

To parse a large number of dates a pre instantiated formatter delivers best performance:

```
(let [fmt (time/formatter "yyyy-MM-dd'T'HH:mm:ssz")]
  (dotimes [n 100] (time/zoned-date-time-parse "2018-12-01T14:20:01+01:00" fmt)))
```

```
(time/zoned-date-time-parse "2018-08-01T14:20:01+01:00" "yyyy-MM-dd'T'HH:mm:ssz")
=> 2018-08-01T14:20:01+01:00
```

```
(time/zoned-date-time-parse "2018-08-01T14:20:01.000+01:00" "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> 2018-08-01T14:20:01+01:00
```

```
(time/zoned-date-time-parse "2018-08-01T14:20:01.000+01:00" :iso)
=> 2018-08-01T14:20:01+01:00
```

```
(time/zoned-date-time-parse "2018-08-01 14:20:01.000 +01:00" "yyyy-MM-dd' 'HH:mm:ss.SSS' 'z")
=> 2018-08-01T14:20:01+01:00
```

[section](#)[top](#)

time/zoned-date-time?

```
(time/zoned-date-time? date)
```

Returns true if date is a zoned-date-time ('java.time.ZonedDateTime') else false

```
(time/zoned-date-time? (time/zoned-date-time))
=> true
```

[section](#)[top](#)

timeout-after

```
(timeout-after p time time-unit)
```

Returns a promise that timeouts after the specified time. The promise throws a TimeoutException.

```
(> (promise (fn [] (sleep 100) "The quick brown fox"))
  (accept-either (timeout-after 500 :milliseconds)
    (fn [v] (println (pr-str v)))))
  (deref))
"The quick brown fox"
=> nil
```

```
(> (promise (fn [] (sleep 1000) "The quick brown fox"))
  (accept-either (timeout-after 500 :milliseconds)
    (fn [v] (println (pr-str v)))))
  (deref))
=> TimeoutException: java.util.concurrent.TimeoutException
```

```
(> (promise (fn [] (sleep 1000) "The quick brown fox"))
  (accept-either (timeout-after 500 :milliseconds)
    (fn [v] (println (pr-str v)))))
  (deref 2000 :timeout))
=> :timeout
```

```
(-> (promise (fn [] (sleep 200) "The quick brown fox"))
    (apply-to-either (timeout-after 100 :milliseconds)
                     identity)
    (deref))
=> TimeoutException: java.util.concurrent.TimeoutException
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with `deref`, and set, once only, with `deliver`. Calls to `deref` prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function `f` with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function `f` with the two ...

[then-apply](#)

Applies a function `f` on the result of the previous stage of the promise `p`.

[then-combine](#)

Applies a function `f` to the result of the previous stage of promise `p` and the result of another promise `p-other`

[then-compose](#)

Composes the result of two promises. `f` receives the result of the first promise `p` and returns a new promise that composes that value ...

[when-complete](#)

Returns the promise `p` with the same result or exception at this stage, that executes the action `f`. Passes the current stage's result ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)[top](#)

timing/elapsed

```
(timing/elapsed f)
```

Runs a function `f` and returns the elapsed time in milliseconds.

```
(timing/elapsed #(sleep 500))
=> 505
```

SEE ALSO

[timing/run](#)

Runs a function `f` with printing the elapsed time. Returns the value that `f` has produced.

[section](#)[top](#)

timing/run

```
(timing/run f)
(timing/run f start-msg)
```

Runs a function `f` with printing the elapsed time. Returns the value that `f` has produced.

```
(timing/run #(sleep 500))
Elapsed: 505ms
=> nil

(timing/run #(sleep 500) "Sleeping...")
Sleeping...
Elapsed: 506ms
=> nil
```

SEE ALSO

[timing/elapsed](#)

Runs a function `f` and returns the elapsed time in milliseconds.

[section](#)

[top](#)

tomcat/create-servlet

```
(create-servlet handler-map)
```

Creates a servlet from a HTTP method handler map

```
;; minimal servlet
(tomcat/create-servlet
 { :doGet (fn [req res servlet] (tomcat/send-ok res "Hello World")) })

;; servlet with lifecycle and all HTTP methods
(tomcat/create-servlet
 { :init (fn [config] nil)
   :destroy (fn [servlet] nil)
   :doGet (fn [req res servlet] (tomcat/send-ok res "Hello World"))
   :doHead (fn [req res servlet] (tomcat/send-not-implemented res "HTTP Method HEAD"))
   :doPost (fn [req res servlet] (tomcat/send-not-implemented res "HTTP Method POST"))
   :doPut (fn [req res servlet] (tomcat/send-not-implemented res "HTTP Method PUT"))
   :delete (fn [req res servlet] (tomcat/send-not-implemented res "HTTP Method DELETE"))
   :getLastModified (fn [req] -1) })
```

[section](#)

[top](#)

tomcat/destroy

```
(destroy server)
```

Destroys a Tomcat server after having stopped it.

```
(do
 (load-module :tomcat ['tomcat :as 'tc])
 (let [server (tc/start (tc/hello-world-servlet)
                        {:await? false, :base-dir ".", :port 8080})])
```

```
(tc/state server)
(sleep 20_000)
(tc/stop server)
(tc/destroy server)))
```

SEE ALSO

[tomcat/start](#)

Start a Tomcat to serve given servlet with supplied options:

[tomcat/state](#)

Returns the state of a Tomcat server.

[tomcat/stop](#)

Stops a Tomcat server.

[tomcat/shutdown](#)

Shutdown a Tomcat server.

[section](#)

[top](#)

tomcat/hello-world-servlet

'Hello World' demo servlet

[section](#)

[top](#)

tomcat/shutdown

```
(shutdown server)
```

Shutdown a Tomcat server.

Shutdown effectively calls

- `(stop server)`
- `(destroy server)`

on the server

```
(do
  (load-module :tomcat ['tomcat :as 'tc])
  (let [server (tc/start (tc/hello-world-servlet)
                        {:await? false, :base-dir ".", :port 8080})]
    (tc/state server)
    (sleep 20_000)
    (tc/shutdown server)))
```

SEE ALSO

[tomcat/start](#)

Start a Tomcat to serve given servlet with supplied options:

[tomcat/state](#)

Returns the state of a Tomcat server.

[tomcat/stop](#)

Stops a Tomcat server.

[tomcat/destroy](#)

Destroys a Tomcat server after having stopped it.

[section](#)

[top](#)

tomcat/start

```
(start servlet options)
(start servlet context-path context-doc-base options)
```

Start a Tomcat to serve given servlet with supplied options:

Server options:

:base-dir	the server's base directory (default: ".")
:await?	block the thread until server get shutdown command (default: true)
:http?	create http connector (default: true)
:port	the port to listen on http connector (default: 8080)
:https?	create https connector (default: false)
:https-port	the port to listen on https connector (default: 8443)
:keystore	path to keystore file include server certificate
:key-pass	password of keystore file
:tls-hostname	hostname to listen for https connector (default: _default_)
:tls-protocol	list of SSL/TLS protocol to support for https connector (default: TLS)
:tls-ciphers	list of SSL/TLS ciphers to support for https connector
:executor?	use executor (default: true)
:executor-name	name of executor (default: tc-executor)
:max-threads	max number of threads in executor (default: 200)
:min-spare-threads	minimum number of spare threads in executor (default: 25)
:max-idle-time	max milliseconds before an idle thread shutdown (default: 60000)

| :max-post-size | max post size for file uploads. Tomcat defaults to 2MB. A value of -1 specifies a indefinite upload size

Servlet options:

:name	the servlet's name (default: "venice-servlet")
:mapping	the servlet's mapping path (default: "/*") single or multiple mappings are possible for a servlet: - single: "/employees" - multiple: ["/employees" "/employees/*"]
:async-support	if true add async support for servlet (default: false)
:load-on-startup	the load-on-startup order value, a negative value means load on first call. (default: -1)
:file-upload	if true configure as file-upload servlet (default: false)
:location	file-upload location (default: "")
:max-file-size	file-upload max file size in bytes (default: -1)
:max-request-size	file-upload max request size in bytes (default: -1)
:file-size-threshold	file-upload max file threshold in bytes (default: 0l)

```
;; Example 1:
;; start Tomcat with
;;   - a servlet
```

```

;; - server options
(tomcat/start (tomcat/hello-world-servlet)
  {:await? false, :base-dir ".", :port 8080})

;; Example 2:
;; start Tomcat with
;; - a servlet
;; - web app context-path
;; - web app context-doc-base
;; - server options
(tomcat/start (tomcat/hello-world-servlet)
  ""
  "."
  {:await? false, :base-dir ".", :port 8080})

;; Example 3:
;; start Tomcat with
;; - a single servlet with servlet options
;; - web app context-path
;; - web app context-doc-base
;; - server options
(tomcat/start [ [ (tomcat/hello-world-servlet)
  {:name "hello-servlet" :mapping "/*"} ] ]
  ""
  "."
  {:await? false, :base-dir ".", :port 8080})

;; Example 4:
;; start Tomcat with
;; - a single fileupload servlet with servlet options
;; - web app context-path
;; - web app context-doc-base
;; - server options
(tomcat/start [ [ (upload-servlet)
  {:name "upload-servlet"
   :mapping "/upload"
   :file-upload true
   :location "/tmp"
   :max-file-size 10485760
   :max-request-size 10485760
   :file-size-threshold -1} ] ]
  ""
  "."
  {:await? false, :base-dir ".", :port 8080})

```

SEE ALSO

[tomcat/state](#)

Returns the state of a Tomcat server.

[tomcat/stop](#)

Stops a Tomcat server.

[tomcat/destroy](#)

Destroys a Tomcat server after having stopped it.

[tomcat/shutdown](#)

Shutdown a Tomcat server.

[section](#)

[top](#)

tomcat/state

```
(state server)
```

Returns the state of a Tomcat server.

A Tomcat server state is of:

- `:NEW`
- `:INITIALIZING`
- `:INITIALIZED`
- `:STARTING_PREP`
- `:STARTING`
- `:STARTED`
- `:STOPPING_PREP`
- `:STOPPING`
- `:STOPPED`
- `:DESTROYING`
- `:DESTROYED`
- `:FAILED`

```
(do
  (load-module :tomcat ['tomcat :as 'tc])
  (let [server (tc/start (tc/hello-world-servlet)
                        {:await? false, :base-dir ".", :port 8080})]
    (tc/state server)
    (sleep 20_000)
    (tc/shutdown server)))
```

SEE ALSO

[tomcat/start](#)

Start a Tomcat to serve given servlet with supplied options:

[tomcat/stop](#)

Stops a Tomcat server.

[tomcat/destroy](#)

Destroys a Tomcat server after having stopped it.

[tomcat/shutdown](#)

Shutdown a Tomcat server.

[section](#)

[top](#)

tomcat/stop

```
(stop server)
```

Stops a Tomcat server.

Note: Do not forget to call `destroy` on the server after having stopped it.

```
(do
  (load-module :tomcat ['tomcat :as 'tc])
  (let [server (tc/start (tc/hello-world-servlet)
                        {:await? false, :base-dir ".", :port 8080})]
    (tc/state server)
    (sleep 20_000)
```



```
(tc/stop server)
(tc/destroy server)))
```

SEE ALSO

[tomcat/start](#)

Start a Tomcat to serve given servlet with supplied options:

[tomcat/state](#)

Returns the state of a Tomcat server.

[tomcat/destroy](#)

Destroys a Tomcat server after having stopped it.

[tomcat/shutdown](#)

Shutdown a Tomcat server.

[section](#)

[top](#)

total-memory

```
(total-memory)
```

Returns the total amount of memory available to the Java VM.

```
(total-memory)
=> "2532.5MB"
```

SEE ALSO

[used-memory](#)

Returns the currently used memory by the Java VM.

[section](#)

[top](#)

trace/tee

```
(tee x)
```

Allows to branch off values passed to `tee` to a printer.

The form is equivalent to:

```
(tee-> x #(println "trace:" %))
```

```
(tee->> x #(println "trace:" %))
```

when used with the threading macros `->` and `->>`

```
(do
  (load-module :trace ['trace :as 't])

  (-> 5
    (+ 3)
    t/tee
    (/ 2)
    t/tee
    (- 1)))
trace: 8
trace: 4
=> 3
```

SEE ALSO

[trace/tee->](#)

Allows to branch off values passed through the forms of a `->` macro

[trace/tee->>](#)

Allows to branch off values passed through the form of a `->>` macro

[section](#)

[top](#)

trace/tee->

```
(tee-> x f!)
```

Allows to branch off values passed through the forms of a `->` macro

```
(do
  (load-module :trace ['trace :as 't])

  (-> 5
    (+ 3)
    (t/tee-> #(println "trace:" %))
    (/ 2)
    (t/tee-> #(println "trace:" %))
    (- 1)))
trace: 8
trace: 4
=> 3
```

SEE ALSO

[trace/tee->>](#)

Allows to branch off values passed through the form of a `->>` macro

[trace/tee](#)

Allows to branch off values passed to tee to a printer.

[section](#)

[top](#)

trace/tee->>

```
(tee->> f! x)
```

Allows to branch off values passed through the form of a `->>` macro

```
(do
  (load-module :trace ['trace :as 't])

  (->> 5
    (+ 3)
    (t/tee->> #(println "trace:" %))
    (/ 32)
    (t/tee->> #(println "trace:" %))
    (- 1)))
```

```
trace: 8
trace: 4
=> -3
```

SEE ALSO

[trace/tee->](#)

Allows to branch off values passed through the forms of a -> macro

[trace/tee](#)

Allows to branch off values passed to tee to a printer.

[section](#)

[top](#)

trace/trace

```
(trace val)
(trace name val)
```

Sends name (optional) and value to the tracer function, then returns value. May be wrapped around any expression without affecting the result.

```
(trace/trace (+ 1 2))
TRACE: 3
=> 3
```

```
(trace/trace "add" (+ 1 2))
TRACE add: 3
=> 3
```

```
(* 4 (trace/trace (+ 1 2)))
TRACE: 3
=> 12
```

SEE ALSO

[trace/trace-var](#)

Traces the var

[trace/trace-str-limit](#)

Manages the trace string limit for the current thread. Without argument returns the current limit. With argument sets the trace string ...

[section](#)

[top](#)

trace/trace-str-limit

```
(trace-str-limit)
(trace-str-limit n)
```

Manages the trace string limit for the current thread. Without argument returns the current limit. With argument sets the trace string length limit to n. The limit defaults to 80.

```
(trace/trace-str-limit 120)
=> 120
```

SEE ALSO

trace/trace-var

Traces the var

trace/trace

Sends name (optional) and value to the tracer function, then returns value. May be wrapped around any expression without affecting the result.

[section](#)

[top](#)

trace/trace-var

```
(trace-var v)
```

Traces the var

```
(do
  (load-module :trace ['trace :as 't])
```

```
  (t/trace-var +)
```

```
  (+ 1 2))
```

```
TRACE t115257: (core/+ 1 2)
```

```
TRACE t115257: => 3
```

```
=> 3
```

```
(do
  (load-module :trace ['trace :as 't])
```

```
  (defn foo [x] (+ x 2))
```

```
  (defn zoo [x] (foo x))
```

```
  (defn bar [x] (zoo x))
```

```
  (t/trace-var +)
```

```
  (t/trace-var foo)
```

```
  (t/trace-var bar)
```

```
  (bar 5))
```

```
TRACE t115286: (user/bar 5)
```

```
TRACE t115287: | (user/foo 5)
```

```
TRACE t115288: | | (core/+ 5 2)
```

```
TRACE t115288: | | => 7
```

```
TRACE t115287: | => 7
```

```
TRACE t115286: => 7
```

```
=> 7
```

```
(do
  (load-module :trace ['trace :as 't])
```

```
  (defn foo [x] (/ x 0)) ;; division by zero!
```

```
  (defn bar [x] (foo x))
```

```
  (t/trace-var /)
```

```
  (t/trace-var foo)
```

```
  (t/trace-var bar)
```

```
  (bar 5))
```

```
TRACE t115317: (user/bar 5)
```

```
TRACE t115318: | (user/foo 5)
```

```
TRACE t115319: | | (core// 5 0)
```

```
TRACE t115319: | | => com.github.jlangch.venice.VncException: / by zero
```

```
TRACE t115318: | => com.github.jlangch.venice.VncException: / by zero
```

```
TRACE t115317: => com.github.jlangch.venice.VncException: / by zero
=> VncException: / by zero
```

SEE ALSO

[trace/untrace-var](#)

Untraces the var

[trace/traced?](#)

Returns true if the given var is currently traced, false otherwise

[trace/traceable?](#)

Returns true if the given var can be traced, false otherwise

[trace/trace](#)

Sends name (optional) and value to the tracer function, then returns value. May be wrapped around any expression without affecting the result.

[trace/trace-str-limit](#)

Manages the trace string limit for the current thread. Without argument returns the current limit. With argument sets the trace string ...

[section](#)

[top](#)

trace/traceable?

```
(traceable? v)
```

Returns true if the given var can be traced, false otherwise

```
(trace/traceable? +)
```

```
=> true
```

SEE ALSO

[trace/trace-var](#)

Traces the var

[trace/traced?](#)

Returns true if the given var is currently traced, false otherwise

[section](#)

[top](#)

trace/traced?

```
(traced? v)
```

Returns true if the given var is currently traced, false otherwise

```
(trace/traced? +)
```

```
=> false
```

SEE ALSO

[trace/trace-var](#)

Traces the var

[trace/untrace-var](#)

Untraces the var

[trace/untrace-all](#)

Untraces all

[trace/traceable?](#)

Returns true if the given var can be traced, false otherwise

[trace/trace](#)

Sends name (optional) and value to the tracer function, then returns value. May be wrapped around any expression without affecting the result.

[section](#)

[top](#)

trace/untrace-all

```
(untrace-all)
```

Untraces all

```
(trace/untrace-all)  
=> nil
```

SEE ALSO

[trace/untrace-var](#)

Untraces the var

[trace/traced?](#)

Returns true if the given var is currently traced, false otherwise

[section](#)

[top](#)

trace/untrace-var

```
(untrace-var v)
```

Untraces the var

```
(trace/untrace-var +)  
=> nil
```

SEE ALSO

[trace/untrace-all](#)

Untraces all

[trace/trace-var](#)

Traces the var

[trace/traced?](#)

Returns true if the given var is currently traced, false otherwise

[section](#)

[top](#)

trampoline

```
(trampoline f)  
(trampoline f & args)
```

trampoline can be used to convert algorithms requiring mutual recursion without stack consumption. Calls f with supplied args, if any. If f returns a fn, calls that fn with no arguments, and continues to repeat, until the return value is not a fn, then returns that non-fn value.

Note that if you want to return a fn as a final value, you must wrap it in some data structure and unpack it after trampoline returns.

```
(do
  (defn factorial
    ([n] #(factorial n 1N))
    ([n acc] (if (< n 2)
                acc
                #(factorial (dec n) (* acc n))))))

(trampoline (factorial 20)))
=> 2432902008176640000N
```

[section](#)

[top](#)

transduce

```
(transduce xform f coll)
(transduce xform f init coll)
```

Reduce with a transformation of a reduction function f (xf). If init is not supplied, (f) will be called to produce it. f should be a reducing step function that accepts both 1 and 2 arguments. Returns the result of applying (the transformed) xf to init and the first item in coll, then applying xf to that result and the 2nd item, etc. If coll contains no items, returns init and f is not called.

transduce can work with queues as collection, given that the end of the queue is marked by adding a nil element. Otherwise the transducer does not know when to stop reading elements from the queue.

Transformations		Reductions	Control
-----		-----	-----
map	map-indexed	rf-first	halt-when
filter	flatten	rf-last	
drop	drop-while	rf-any?	
drop-last	remove	rf-every?	
take	take-while		
take-last	keep	conj	
dedupe	distinct	+, *	
sorted	reverse	max, min	

```
(transduce identity + [1 2 3 4])
=> 10
```

```
(transduce (map #(+ % 3)) + [1 2 3 4])
=> 22
```

```
(transduce identity max [1 2 3])
=> 3
```

```
(transduce identity rf-last [1 2 3])
=> 3
```

```
(transduce identity (rf-every? pos?) [1 2 3])
=> true
```

```
(transduce (map inc) conj [1 2 3])
=> [2 3 4]
```

```
;; transduce all elements of a queue or a deque.
;; calls (take! queue) to get the elements of the queue.
;; note: use nil to mark the end of the queue otherwise
;;      transduce will block forever!
(let [q (conj! (queue) 1 2 3 nil)]
  (transduce (map inc) conj q))
=> [2 3 4]
```

```
;; reduce data supplied by a finit lazy seq
(do
  (def counter (atom 5))
  (defn generate []
    (swap! counter dec)
    (if (pos? @counter) @counter nil))
  (transduce (map inc) conj (lazy-seq generate)))
=> [5 4 3 2]
```

```
(do
  (def xform (comp (drop 2) (take 3)))
  (transduce xform conj [1 2 3 4 5 6]))
=> [3 4 5]
```

```
(do
  (def xform (comp
    (map #(* % 10))
    (map #(+ % 1))
    (sorted compare)
    (drop 3)
    (take 2)
    (reverse)))
  (transduce xform conj [1 2 3 4 5 6]))
=> [51 41]
```

SEE ALSO

[map](#)

Applies f to the set of first items of each coll, followed by applying f to the set of second items in each coll, until any one of the ...

[filter](#)

Returns a collection of the items in coll for which (predicate item) returns logical true.

[reduce](#)

f should be a function of 2 arguments. If val is not supplied, returns the result of applying f to the first 2 items in coll, then ...

[map-indexed](#)

Returns a collection of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item of coll, etc. until ...

[tee](#)

Applies f to every item of each the coll and returns the unaltered coll.

[section](#)

[top](#)

true?

```
(true? x)
```

Returns true if x is true, false otherwise


```
(true? true)
=> true

(true? false)
=> false

(true? nil)
=> false

(true? 0)
=> false

(true? (== 1 1))
=> true
```

SEE ALSO

[false?](#)

Returns true if x is false, false otherwise

[not](#)

Returns true if x is logical false, false otherwise.

[section](#)[top](#)

try

```
(try expr*)
(try expr* (catch selector ex-sym expr*)*)
(try expr* (catch selector ex-sym expr*)* (finally expr*))
```

Exception handling: try - catch - finally

`(try)` without any expression returns `nil`.

The exception types

- `:java.lang.Exception`
- `:java.lang.RuntimeException`
- `:com.github.jlangch.venice.VncException`
- `:com.github.jlangch.venice.ValueException`

are imported implicitly so its alias `:Exception`, `:RuntimeException`, `:VncException`, and `:ValueException` can be used as selector without an import of the class.

Selectors

- a class: (e.g., `:RuntimeException`, `:java.text.ParseException`), matches any instance of that class
- a key-values vector: (e.g., `[key val & kvs]`), matches any instance of `:ValueException` where the exception's value meets the expression `(and (= (get ex-value key) val) ...)`
- a predicate: (a function of one argument like `map?`, `set?`), matches any instance of `:ValueException` where the predicate applied to the exception's value returns true

Notes:

The finally block is just for side effects, like closing resources. It never returns a value!

All exceptions in Venice are *unchecked*. If *checked* exceptions are thrown in Venice they are immediately wrapped in a `:RuntimeException` before being thrown! If Venice catches a *checked* exception from a Java interop call it wraps it in a `:RuntimeException` before handling it by the catch block selectors.

Venice follows the Java rules when propagating exceptions:

1. exception from finally block
2. exception from catch block
3. exception from body block

```
(try
  (throw "test")
  (catch :ValueException e
    "caught ~(ex-value e)"))
=> "caught test"
```

```
(try
  (throw 100)
  (catch :Exception e -100))
=> -100
```

```
(try
  (throw 100)
  (catch :ValueException e (ex-value e))
  (finally (println "...finally")))
...finally
=> 100
```

```
(try
  (throw (ex :RuntimeException "message")))
  (catch :RuntimeException e (ex-message e)))
=> "message"
```

;; exception type selector:

```
(try
  (throw [1 2 3])
  (catch :ValueException e (ex-value e))
  (catch :RuntimeException e "runtime ex")
  (finally (println "...finally")))
...finally
=> [1 2 3]
```

;; key-value selector:

```
(try
  (throw {:a 100, :b 200})
  (catch [:a 100] e
    (println "ValueException, value: ~(ex-value e)"))
  (catch [:a 100, :b 200] e
    (println "ValueException, value: ~(ex-value e)")))
ValueException, value: {:a 100 :b 200}
=> nil
```

;; key-value selector (exception cause):

```
(try
  (throw (ex :java.io.IOException "failure")))
  (catch [:cause-type :java.io.IOException] e
    (println "IOException, msg: ~(ex-message (ex-cause e))"))
  (catch :RuntimeException e
    (println "RuntimeException, msg: ~(ex-message e)")))
IOException, msg: failure
=> nil
```

;; predicate selector:

```
(try
  (throw {:a 100, :b 200})
  (catch long? e
```

```

    (println "ValueException, value: ~(ex-value e)")
  (catch map? e
    (println "ValueException, value: ~(ex-value e)")
    (catch #(and (map? %) (= 100 (:a %))) e
      (println "ValueException, value: ~(ex-value e))))
ValueException, value: {:a 100 :b 200}
=> nil

;; predicate selector with custom types:
(do
  (deftype :my-exception1 [message :string, position :long])
  (deftype :my-exception2 [message :string])

  (try
    (throw (my-exception1. "error" 100))
    (catch my-exception1? e
      (println (:value e)))
    (catch my-exception2? e
      (println (:value e)))))
{:custom-type* :user/my-exception1 :message error :position 100}
=> nil

```

SEE ALSO

[try-with](#)

try-with-resources allows the declaration of resources to be used in a try block with the assurance that the resources will be closed ...

[throw](#)

Throws an exception.

[ex](#)

Creates an exception of type class with optional args. The class must be a subclass of :java.lang.Exception

[section](#)

[top](#)

try-acquire

```

(try-acquire lock)
(try-acquired lock timeout time-unit)

```

Acquires a lock within the given timeout time. Without a timeout returns immediately if the lock is not available.

Returns `true` if the lock could be acquired within the given time else `false` .

```

(let [l (lock)]
  (when (try-acquire l)
    ;; do something
    (release l)))
=> nil

(let [l (lock)]
  (when (try-acquire l 3 :seconds)
    ;; do something
    (release l)))
=> nil

```

SEE ALSO

[lock](#)

Creates a new lock object.

acquire

Acquires a lock, blocking until the lock is available.

release

Releases a lock.

locked?

Returns true if the lock is in use else false.

[section](#)

[top](#)

try-with

```
(try-with [bindings*] expr*)
(try-with [bindings*] expr* (catch selector ex-sym expr*))
(try-with [bindings*] expr* (catch selector ex-sym expr*)* (finally expr))
```

try-with-resources allows the declaration of resources to be used in a try block with the assurance that the resources will be closed after execution of that block. The resources declared must implement the `Closeable` or `AutoCloseable` interface.

Venice follows the Java rules when propagating exceptions:

1. exception from finally block
2. exception from catch block
3. exception from body block
4. exception from resource auto-close

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (io/spit file "123456789" :append true)
    (try-with [is (io/file-in-stream file)]
      (io/slurp-stream is :binary false))))
=> "123456789"
```

SEE ALSO

try

Exception handling: try - catch - finally

throw

Throws an exception.

ex

Creates an exception of type class with optional args. The class must be a subclass of `java.lang.Exception`

[section](#)

[top](#)

type

```
(type x)
```

Returns the type of x.

```
(type 5)
=> :core/long
```

```
(type [1 2])  
=> :core/vector
```

```
(type (. :java.math.BigInteger :valueOf 100))  
=> :java.math.BigInteger
```

SEE ALSO

[supertype](#)

Returns the super type of x.

[supertypes](#)

Returns the super types of x.

[instance-of?](#)

Returns true if x is an instance of the given type

[section](#)[top](#)

union

```
(union s1)  
(union s1 s2)  
(union s1 s2 & sets)
```

Return a set that is the union of the input sets

```
(union (set 1 2 3))  
=> #{1 2 3}
```

```
(union (set 1 2) (set 2 3))  
=> #{1 2 3}
```

```
(union (set 1 2 3) (set 1 2) (set 1 4) (set 3))  
=> #{1 2 3 4}
```

SEE ALSO

[difference](#)

Return a set that is the first set without elements of the remaining sets

[intersection](#)

Return a set that is the intersection of the input sets

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[conj](#)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item) and (conj item) returns item.

[disj](#)

Returns a new set with the x, xs removed.

[section](#)[top](#)

update

```
(update m k f)
(update m k f & fargs)
```

Updates a value in an associative structure, where `k` is a key and `f` is a function that will take the old value and any supplied `fargs` and return the new value. Returns a new structure.

If the key does not exist, `nil` is passed as the old value. The optional `fargs` are passed to the function `f` as `(f old-value (f old-value arg1 arg2 ...) ...)`.

```
(update [] 0 (fn [x] 5))
=> [5]

(update [0 1 2] 0 (fn [x] 5))
=> [5 1 2]

(update [0 1 2] 1 (fn [x] (+ x 3)))
=> [0 4 2]

(update {} :a (fn [x] 5))
=> {:a 5}

(update {:a 0} :b (fn [x] 5))
=> {:a 0 :b 5}

(update {:a 0 :b 1} :a (fn [x] (+ x 5)))
=> {:a 5 :b 1}

(update [0 1 2] 1 + 3)
=> [0 4 2]

(update {:a 0 :b 1} :b * 4)
=> {:a 0 :b 4}
```

SEE ALSO

[assoc](#)

When applied to a map, returns a new map of the same type, that contains the mapping of key(s) to val(s). When applied to a vector, ...

[dissoc](#)

Returns a new coll of the same type, that does not contain a mapping for key(s)

[section](#)

[top](#)

update!

```
(update! m k f & fargs)
```

Updates a value in a mutable associative structure, where `k` is a key and `f` is a function that will take the old value and any supplied `fargs` and return the new value. Returns a new structure.

If the key does not exist, `nil` is passed as the old value. The optional `fargs` are passed to the function `f` as `(f old-value arg1 arg2 ...)`.

```
(update! (mutable-vector) 0 (fn [x] 5))
=> [5]

(update! (mutable-vector 0 1 2) 0 (fn [x] 5))
=> [5 1 2]
```

```
(update! (mutable-vector 0 1 2) 0 (fn [x] (+ x 1)))
=> [1 1 2]

(update! (mutable-map) :a (fn [x] 5))
=> {:a 5}

(update! (mutable-map :a 0) :b (fn [x] 5))
=> {:a 0 :b 5}

(update! (mutable-map :a 0 :b 1) :a (fn [x] 5))
=> {:a 5 :b 1}

(update! (mutable-vector 0 1 2) 0 + 4)
=> [4 1 2]

(update! (mutable-map :a 0 :b 1) :b * 4)
=> {:a 0 :b 4}
```

SEE ALSO

[assoc!](#)

Associates key/vals with a mutable map, returns the map

[dissoc!](#)

Dissociates keys from a mutable map, returns the map

[section](#)[top](#)

update-in

```
(update-in [m ks f & fargs])
```

Updates' a value in a nested associative structure, where ks is a sequence of keys and f is a function that will take the old value and any supplied fargs and return the new value, and returns a new nested structure.

If any levels do not exist, hash-maps will be reated.

```
(do
  (def users [ {:name "James" :age 26}
                {:name "John" :age 43} ])
  (update-in users [1 :age] inc))
=> [{:name "James" :age 26} {:name "John" :age 44}]

(update-in {:a 12} [:a] * 4)
=> {:a 48}

(update-in {:a 12} [:a] + 3 4)
=> {:a 19}
```

[section](#)[top](#)

used-memory

```
(used-memory)
```

Returns the currently used memory by the Java VM.

([used-memory](#))

```
=> "360.8MB"
```

SEE ALSO

[total-memory](#)

Returns the total amount of memory available to the Java VM.

[section](#)

[top](#)

user-name

([user-name](#))

Returns the logged-in's user name.

([user-name](#))

```
=> "juerg"
```

SEE ALSO

[io/user-home-dir](#)

Returns the user's home dir as a java.io.File.

[section](#)

[top](#)

uuid

([uuid](#))

Generates a UUID.

([uuid](#))

```
=> "8a35331f-e7f7-448d-aab3-109044dc5bd1"
```

[section](#)

[top](#)

val

([val](#) e)

Returns the val of the map entry.

([val](#) ([find](#) {[:a](#) 1 [:b](#) 2} [:b](#)))

```
=> 2
```

([val](#) ([first](#) ([entries](#) {[:a](#) 1 [:b](#) 2 [:c](#) 3})))

```
=> 1
```

SEE ALSO

map

Applies `f` to the set of first items of each coll, followed by applying `f` to the set of second items in each coll, until any one of the ...

entries

Returns a collection of the map's entries.

key

Returns the key of the map entry.

vals

Returns a collection of the map's values.

[section](#)[top](#)

vals

```
(vals map)
```

Returns a collection of the map's values.

Please note that the functions 'keys' and 'vals' applied to the same map are not guaranteed not return the keys and vals in the same order!

To achieve this, keys and vals can be calculated based on the map's entry list:

```
(let [e (entries {:a 1 :b 2 :c 3})]  
  (println (map key e))  
  (println (map val e)))
```

```
(vals {:a 1 :b 2 :c 3})  
=> (1 2 3)
```

SEE ALSO

keys

Returns a collection of the map's keys.

entries

Returns a collection of the map's entries.

map

Applies `f` to the set of first items of each coll, followed by applying `f` to the set of second items in each coll, until any one of the ...

[section](#)[top](#)

var-get

```
(var-get v)
```

Returns a var's value.

The var must exist (bound with a value) otherwise nil is returned.

```
(var-get +)  
=> +
```

```
(var-get '+)  
=> +
```

```
(var-get (symbol "+"))  
=> +
```

```
((var-get +) 1 2)
=> 3
```

```
(do
  (def x 10)
  (var-get 'x))
=> 10
```

SEE ALSO

[var-sym](#)

Returns the var's symbol.

[var-name](#)

Returns the unqualified name of the var's symbol.

[var-ns](#)

Returns the namespace of the var's symbol.

[var-val-meta](#)

Returns the var's value meta data.

[var-local?](#)

Returns true if the var is local else false

[var-global?](#)

Returns true if the var is global else false

[var-thread-local?](#)

Returns true if the var is thread-local else false

[section](#)[top](#)

var-global?

```
(var-global? v)
```

Returns true if the var is global else false

```
(var-global? +)
=> true
```

```
(var-global? '+)
=> true
```

```
(var-global? (symbol "+"))
=> true
```

```
(do
  (def x 10)
  (var-global? x))
=> true
```

```
(let [x 10]
  (var-global? x))
=> false
```

SEE ALSO

[var-get](#)

Returns a var's value.

[var-name](#)

Returns the unqualified name of the var's symbol.

[var-ns](#)

Returns the namespace of the var's symbol.

[var-local?](#)

Returns true if the var is local else false

[var-thread-local?](#)

Returns true if the var is thread-local else false

[bound?](#)

Returns true if the symbol is bound to a value else false

[section](#)

[top](#)

var-local?

```
(var-local? v)
```

Returns true if the var is local else false

```
(var-local? +)
```

```
=> true
```

```
(var-local? '+)
```

```
=> true
```

```
(var-local? (symbol "+"))
```

```
=> true
```

```
(let [x 10]
```

```
  (var-local? x))
```

```
=> true
```

```
(do
```

```
  (def x 10)
```

```
  (var-local? x))
```

```
=> false
```

SEE ALSO

[var-get](#)

Returns a var's value.

[var-name](#)

Returns the unqualified name of the var's symbol.

[var-ns](#)

Returns the namespace of the var's symbol.

[var-global?](#)

Returns true if the var is global else false

[var-thread-local?](#)

Returns true if the var is thread-local else false

[bound?](#)

Returns true if the symbol is bound to a value else false

[section](#)

[top](#)

var-name

```
(var-name v)
```

Returns the unqualified name of the var's symbol.

The var must exist (bound with a value) otherwise nil is returned.

```
(var-name +)
```

```
=> "+"
```

```
(var-name '+)
```

```
=> "+"
```

```
(var-name (symbol "+"))
```

```
=> "+"
```

```
;; aliased function
```

```
(do
```

```
  (ns foo)
```

```
  (def add +)
```

```
  (var-name add))
```

```
=> "add"
```

```
(do
```

```
  (def x 10)
```

```
  (var-name x))
```

```
=> "x"
```

```
(let [x 10]
```

```
  (var-name x))
```

```
=> "x"
```

```
;; compare with name
```

```
(do
```

```
  (ns foo)
```

```
  (def add +)
```

```
  (name add))
```

```
=> "+"
```

```
;; compare aliased function with name
```

```
(do
```

```
  (ns foo)
```

```
  (def add +)
```

```
  (name add))
```

```
=> "+"
```

SEE ALSO

[name](#)

Returns the name string of a string, symbol, keyword, or function. If applied to a string it returns the string itself.

[var-get](#)

Returns a var's value.

var-sym

Returns the var's symbol.

var-ns

Returns the namespace of the var's symbol.

var-sym-meta

Returns the var's symbol meta data.

var-local?

Returns true if the var is local else false

var-global?

Returns true if the var is global else false

var-thread-local?

Returns true if the var is thread-local else false

[section](#)

[top](#)

var-ns

```
(var-ns v)
```

Returns the namespace of the var's symbol.

The var must exist (bound with a value) otherwise nil is returned.

```
(var-ns +)
```

```
=> "core"
```

```
(var-ns '+)
```

```
=> "core"
```

```
(var-ns (symbol "+"))
```

```
=> "core"
```

```
;; aliased function
```

```
(do
```

```
  (ns foo)
```

```
  (def add +)
```

```
  (var-ns add))
```

```
=> "foo"
```

```
(do
```

```
  (def x 10)
```

```
  (var-ns x))
```

```
=> "user"
```

```
(let [x 10]
```

```
  (var-ns x))
```

```
=> nil
```

```
;; compare with namespace
```

```
(do
```

```
  (ns foo)
```

```
  (def add +)
```

```
  (namespace add))
```

```
=> nil
```

```
;; compare aliased function with namespace
(do
  (ns foo)
  (def add +)
  (namespace add))
=> nil
```

SEE ALSO

[namespace](#)

Returns the namespace string of a symbol, keyword, or function. If x is a registered namespace returns x.

[var-get](#)

Returns a var's value.

[var-name](#)

Returns the unqualified name of the var's symbol.

[var-local?](#)

Returns true if the var is local else false

[var-global?](#)

Returns true if the var is global else false

[var-thread-local?](#)

Returns true if the var is thread-local else false

[section](#)[top](#)

var-sym

```
(var-sym v)
```

Returns the var's symbol.

The var must exist (bound with a value) otherwise nil is returned.

```
(var-sym +)
```

```
=> core/+
```

```
(var-sym '+)
```

```
=> core/+
```

```
(var-sym (symbol "+"))
```

```
=> core/+
```

```
(do
```

```
  (ns test)
```

```
  (defn x [] nil)
```

```
  (var-sym x))
```

```
=> test/x
```

```
(let [x 100] (var-sym x))
```

```
=> x
```

```
(binding [x 100] (var-sym x))
```

```
=> x
```

```
(do
  (defn foo [x] (var-sym x))
  (foo nil))
=> x
```

SEE ALSO

[var-get](#)

Returns a var's value.

[var-name](#)

Returns the unqualified name of the var's symbol.

[var-ns](#)

Returns the namespace of the var's symbol.

[var-sym-meta](#)

Returns the var's symbol meta data.

[var-local?](#)

Returns true if the var is local else false

[var-global?](#)

Returns true if the var is global else false

[var-thread-local?](#)

Returns true if the var is thread-local else false

[section](#)

[top](#)

var-sym-meta

```
(var-sym-meta v)
```

Returns the var's symbol meta data.

The var must exist (bound with a value) otherwise nil is returned.

```
(do
  (def ^{:foo 3} x 100)
  (:foo (var-sym-meta 'x)))
=> 3
```

```
(do
  (let [^{:foo 3} x 100]
    (:foo (var-sym-meta 'x))))
=> 3
```

```
(do
  (defn bar [^{:foo 3} x]
    (:foo (var-sym-meta 'x)))
  (bar 100))
=> 3
```

SEE ALSO

[var-val-meta](#)

Returns the var's value meta data.

[var-get](#)

Returns a var's value.

[var-sym](#)

Returns the var's symbol.

[var-name](#)

Returns the unqualified name of the var's symbol.

[bound?](#)

Returns true if the symbol is bound to a value else false

[section](#)[top](#)

var-thread-local?

```
(var-thread-local? v)
```

Returns true if the var is thread-local else false

```
(binding [x 100]
  (var-thread-local? x))
=> true
```

SEE ALSO

[var-get](#)

Returns a var's value.

[var-name](#)

Returns the unqualified name of the var's symbol.

[var-ns](#)

Returns the namespace of the var's symbol.

[var-local?](#)

Returns true if the var is local else false

[var-global?](#)

Returns true if the var is global else false

[bound?](#)

Returns true if the symbol is bound to a value else false

[section](#)[top](#)

var-val-meta

```
(var-val-meta v)
```

Returns the var's value meta data.

The var must exist (bound with a value) otherwise nil is returned.

```
(do
  (def x ^{:foo 4} 100)
  (:foo (var-val-meta 'x)))
=> 4
```



```
(do
  (def x (vary-meta 100 assoc :foo 4))
  (:foo (var-val-meta 'x)))
=> 4
```

```
(do
  (let [x ^{:foo 4} 100]
    (:foo (var-val-meta 'x))))
=> 4
```

```
(do
  (defn bar [x]
    (:foo (var-val-meta 'x)))
  (bar (vary-meta 100 assoc :foo 4)))
=> 4
```

SEE ALSO

[var-sym-meta](#)

Returns the var's symbol meta data.

[var-get](#)

Returns a var's value.

[var-sym](#)

Returns the var's symbol.

[var-name](#)

Returns the unqualified name of the var's symbol.

[bound?](#)

Returns true if the symbol is bound to a value else false

[section](#)

[top](#)

vary-meta

```
(vary-meta obj f & args)
```

Returns a copy of the object obj, with (apply f (meta obj) args) as its metadata.

```
(meta (vary-meta [1 2] assoc :foo 3))
=> {:foo 3 :line 70 :column 28 :file "example"}
```

SEE ALSO

[meta](#)

Returns the metadata of obj, returns nil if there is no metadata.

[with-meta](#)

Returns a copy of the object obj, with a map m as its metadata.

[var-val-meta](#)

Returns the var's value meta data.

[var-sym-meta](#)

Returns the var's symbol meta data.

[section](#)

[top](#)

vector

```
(vector & items)
```

Creates a new vector containing the items.

```
(vector)  
=> []
```

```
(vector 1 2 3)  
=> [1 2 3]
```

```
(vector 1 2 3 [:a :b])  
=> [1 2 3 [:a :b]]
```

```
(vector "abc")  
=> ["abc"]
```

[section](#)[top](#)

vector*

```
(vector* args)  
(vector* a args)  
(vector* a b args)  
(vector* a b c args)  
(vector* a b c d & more)
```

Creates a new vector containing the items prepended to the rest, the last of which will be treated as a collection.

```
(vector* 1 [2 3])  
=> [1 2 3]
```

```
(vector* 1 2 3 [4])  
=> [1 2 3 4]
```

```
(vector* 1 2 3 '(4 5))  
=> [1 2 3 4 5]
```

```
(vector* '[1 2] 3 [4])  
=> [[1 2] 3 4]
```

```
(vector* nil)  
=> nil
```

```
(vector* nil [2 3])  
=> [nil 2 3]
```

```
(vector* 1 2 nil)  
=> (1 2)
```

SEE ALSO

[cons](#)

Returns a new collection where x is the first element and coll is the rest.

[conj](#)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item) and (conj item) returns item.

[concat](#)

Returns a list of the concatenation of the elements in the supplied collections.

[list*](#)

Creates a new list containing the items prepended to the rest, the last of which will be treated as a collection.

[section](#)[top](#)

vector?

```
(vector? obj)
```

Returns true if obj is a vector

```
(vector? (vector 1 2))  
=> true
```

```
(vector? [1 2])  
=> true
```

[section](#)[top](#)

venice->java

```
(venice->java val)
```

Converts a Venice value to a Java object.

Converts scalar values and collections.

[section](#)[top](#)

version

```
(version)
```

Returns the Venice version.

```
(version)  
=> "0.0.0"
```

SEE ALSO

[latest](#)

Returns the latest available Venice version or nil if the internet is not available.

[parse-version](#)

Parses a version to its elements: major, minor, patch and suffix.

[newer-version?](#)

Returns true if v1 is newer than v2 else false.

[section](#)

[top](#)

volatile

```
(volatile x)
```

Creates a volatile with the initial value x

```
(do
  (def counter (volatile 0))
  (swap! counter inc)
  (deref counter))
=> 1
```

```
(do
  (def counter (volatile 0))
  (reset! counter 9)
  @counter)
=> 9
```

SEE ALSO

[deref](#)

Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When applied to a future, will ...

[reset!](#)

Sets the value of an atom or a volatile to newval without regard for the current value. Returns newval.

[swap!](#)

Atomically swaps the value of an atom or a volatile to be: (apply f current-value-of-box args). Note that f may be called multiple ...

[section](#)

[top](#)

volatile?

```
(volatile? x)
```

Returns true if x is a volatile, otherwise false

```
(do
  (def counter (volatile 0))
  (volatile? counter))
=> true
```

[section](#)

[top](#)

when

```
(when test & body)
```

Evaluates test. If logical true, evaluates body in an implicit do.

```
(when (== 1 1) true)
=> true
```

SEE ALSO

[when-not](#)

Evaluates test. If logical false, evaluates body in an implicit do.

[when-let](#)

bindings is a vector with 2 elements: binding-form test.

[if](#)

Evaluates test. If logical true, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

[if-not](#)

Evaluates test. If logical false, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

[if-let](#)

bindings is a vector with 2 elements: binding-form test.

[section](#)[top](#)

when-complete

```
(when-complete p f)
```

Returns the promise p with the same result or exception at this stage, that executes the action f. Passes the current stage's result value as first and a possible exception as second argument to the function. The asynchronous function f is called presumably for handling side effects.

```
(-> (promise (fn [] "The Quick Brown Fox"))
    (then-apply str/upper-case)
    (when-complete (fn [v,e] (println (pr-str {:value v :ex e}))))
    (then-apply str/lower-case)
    (deref))
{:value "THE QUICK BROWN FOX" :ex nil}
=> "the quick brown fox"
```

SEE ALSO

[promise](#)

Returns a promise object that can be read with deref, and set, once only, with deliver. Calls to deref prior to delivery will block, ...

[then-accept](#)

Returns a new promise that, when this promise completes normally, is executing the function f with this stage's result as the argument.

[then-accept-both](#)

Returns a new promise that, when either this or the other given promise completes normally, is executing the function f with the two ...

[then-apply](#)

Applies a function f on the result of the previous stage of the promise p.

[then-combine](#)

Applies a function f to the result of the previous stage of promise p and the result of another promise p-other

[then-compose](#)

Composes the result of two promises. f receives the result of the first promise p and returns a new promise that composes that value ...

[accept-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[apply-to-either](#)

Returns a new promise that, when either this or the other given promise completes normally, is executed with the corresponding result ...

[or-timeout](#)

Exceptionally completes the promise with a `TimeoutException` if not otherwise completed before the given timeout.

[complete-on-timeout](#)

Completes the promise with the given value if not otherwise completed before the given timeout.

[section](#)

[top](#)

when-let

```
(when-let bindings & body)
```

`bindings` is a vector with 2 elements: binding-form test.

If test is true, evaluates the body expressions with binding-form bound to the value of test, if not, yields nil

```
(when-let [value (* 100 2)]  
  (str "The expression is true. value=" value))  
=> "The expression is true. value=200"
```

SEE ALSO

[if-let](#)

`bindings` is a vector with 2 elements: binding-form test.

[let](#)

Evaluates the expressions and binds the values to symbols in the new local context.

[section](#)

[top](#)

when-not

```
(when-not test & body)
```

Evaluates test. If logical false, evaluates body in an implicit do.

```
(when-not (== 1 2) true)  
=> true
```

SEE ALSO

[when](#)

Evaluates test. If logical true, evaluates body in an implicit do.

[when-let](#)

`bindings` is a vector with 2 elements: binding-form test.

[if](#)

Evaluates test. If logical true, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

[if-not](#)

Evaluates test. If logical false, evaluates and returns then expression, otherwise else expression, if supplied, else nil.

[if-let](#)

`bindings` is a vector with 2 elements: binding-form test.

[section](#)

[top](#)

while

```
(while test & body)
```

Repeatedly executes body while test expression is true. Presumes some side-effect will cause test to become false/nil. Returns nil.

```
(do
  (def a (atom 5))
  (while (pos? @a)
    (println @a)
    (swap! a dec)))
5
4
3
2
1
=> nil
```

[section](#)[top](#)

with-err-str

```
(with-err-str & forms)
```

Evaluates exprs in a context in which `*err*` is bound to a capturing output stream. Returns the string created by any nested printing calls. `with-err-str` can be nested.

```
(with-err-str (println *err* "a string"))
=> "a string\n"
```

SEE ALSO

[with-out-str](#)

Evaluates exprs in a context in which `*out*` is bound to a capturing output stream. Returns the string created by any nested printing ...

[section](#)[top](#)

with-jmx-connection

```
(with-jmx-connection url ssl env & forms)
```

Establishes a JMX connection to access MBeans of a local or remote Java VM.

Without an explicit JMX connection local MBeans will be accessed. This is the default setup.

It is strongly recommended to configure SSL and authentication for the JMX connector!

Authentication

To enable authentication, the server JVM must be started with these system properties:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.authenticate=true
```

```
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.password.file=/path/to/jmxremote.password
-Dcom.sun.management.jmxremote.access.file=/path/to/jmxremote.access
```

Example password file (jmxremote.password):

```
admin secret
```

Set file permissions strictly:

```
chmod 600 /path/to/jmxremote.password
```

Example access file (jmxremote.access):

```
admin readwrite
```

SSL

Step 1: Generate Keystore and Truststore

For both client and server, the same keystore can be used for testing:

```
keytool -genkeypair -alias jmxssl -keyalg RSA -keystore jmx.keystore -storepass changeit -validity 365
```

Export the certificate:

```
keytool -export -alias jmxssl -keystore jmx.keystore -file jmx.cer -storepass changeit
```

Create a truststore (to trust the server's certificate):

```
keytool -import -alias jmxssl -file jmx.cer -keystore jmx.truststore -storepass changeit -noprompt
```

Step 2: JMX Server Configuration with SSL

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.rmi.port=9999
-Dcom.sun.management.jmxremote.ssl=true
-Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.access.file=/path/to/jmxremote.access
-Dcom.sun.management.jmxremote.password.file=/path/to/jmxremote.password
```

```
# SSL configuration
-Djavax.net.ssl.keyStore=/path/to/jmx.keystore
-Djavax.net.ssl.keyStorePassword=changeit
-Djavax.net.ssl.trustStore=/path/to/jmx.truststore
-Djavax.net.ssl.trustStorePassword=changeit
```

```
(with-jmx-connection "service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi"
  nil
  nil
  (let [name (mbean/object-name "java.lang:type=OperatingSystem")]
    (mbean/info name)))

;; example with ssl and username/password authentication
(with-jmx-connection "service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi"
  { "javax.net.ssl" true
    "javax.net.ssl.trustStore" "/path/to/jmx.truststore"
    "javax.net.ssl.trustStorePassword" "changeit" }
  { "jmx.remote.credentials" ["username" "password"] }
  (let [name (mbean/object-name "java.lang:type=OperatingSystem")]
    (mbean/info name)))

;; local access (without explicit JMX connection)
(let [name (mbean/object-name "java.lang:type=OperatingSystem")]
  (mbean/info name))
```


with-meta

```
(with-meta obj m)
```

Returns a copy of the object `obj`, with a map `m` as its metadata.

```
(meta (with-meta [1 2] {:foo 3}))  
=> {:foo 3}
```

SEE ALSO

[meta](#)

Returns the metadata of `obj`, returns `nil` if there is no metadata.

[vary-meta](#)

Returns a copy of the object `obj`, with `(apply f (meta obj) args)` as its metadata.

[var-val-meta](#)

Returns the var's value meta data.

[var-sym-meta](#)

Returns the var's symbol meta data.

[section](#)

[top](#)

with-out-str

```
(with-out-str & forms)
```

Evaluates `exprs` in a context in which `*out*` is bound to a capturing output stream. Returns the string created by any nested printing calls. `with-out-str` can be nested.

```
(with-out-str (println "a string"))  
=> "a string\n"
```

SEE ALSO

[with-err-str](#)

Evaluates `exprs` in a context in which `*err*` is bound to a capturing output stream. Returns the string created by any nested printing ...

[section](#)

[top](#)

with-sh-dir

```
(with-sh-dir dir & forms)
```

Sets the directory for use with `sh`, see `sh` for details.

```
(with-sh-dir "/tmp" (sh "ls" "-l"))
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[with-sh-env](#)

Sets the environment for use with `sh`.

[with-sh-throw](#)

Shell commands executed within a `with-sh-throw` context throw an exception if the spawned shell process returns an exit code other than 0.

[section](#)

[top](#)

with-sh-env

```
(with-sh-env env & forms)
```

Sets the environment for use with `sh`.

```
(with-sh-env {"NAME" "foo"} (sh "ls" "-l"))
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[with-sh-dir](#)

Sets the directory for use with `sh`, see `sh` for details.

[with-sh-throw](#)

Shell commands executed within a `with-sh-throw` context throw an exception if the spawned shell process returns an exit code other than 0.

[section](#)

[top](#)

with-sh-throw

```
(with-sh-throw forms)
```

Shell commands executed within a `with-sh-throw` context throw an exception if the spawned shell process returns an exit code other than 0.

For use with `sh`, see `sh` for details. `with-sh-throw` can be nested.

```
(with-sh-throw (sh "ls" "-l"))
```

SEE ALSO

[sh](#)

Launches a new sub-process.

[with-sh-env](#)

Sets the environment for use with `sh`.

[with-sh-dir](#)

Sets the directory for use with `sh`, see `sh` for details.

[section](#)

[top](#)

with-tmp-dir

```
(with-tmp-dir subdir & forms)
```

Creates the sub directory 'subdir' in the tmp dir and runs the forms. After running the forms removes the created sub directory recursively.

Binds the created directory to `*tmp-dir*`

```
(with-tmp-dir "test"
  (io/spit (io/file *tmp-dir* "hello-1.txt") "Hello Wold")
  (io/spit (io/file *tmp-dir* "hello-2.txt") "Hello Wold"))
=> nil
```

SEE ALSO

[io/temp-dir](#)

Creates a new temp directory with prefix. Returns a `java.io.File`.

[section](#)

[top](#)

xml/children

(xml/children nodes)

Returns the children of the XML nodes collection

```
(do
  (load-module :xml)
  (xml/children
    (list (xml/parse-str "<a><b>B</b></a>"))))
=> ({:content ["B"] :tag "b"})
```

[section](#)

[top](#)

xml/parse

(xml/parse s)
(xml/parse s handler)

Parses and loads the XML from the source `s` with the parser `XMLHandler` handler. The source may be an `InputStream` or an `InputSource`.

Returns a tree of XML element maps with the keys `:tag`, `:attrs`, and `:content`.

[section](#)

[top](#)

xml/parse-str

(xml/parse-str s)
(xml/parse-str s handler)

Parses an XML from the string `s`. Returns a tree of XML element maps with the keys `:tag`, `:attrs`, and `:content`.

```
(do
  (load-module :xml)
  (xml/parse-str "<a><b>B</b></a>"))
=> {:content [{:content ["B"] :tag "b"}] :tag "a"}
```

xml/path->

```
(xml/path-> path nodes)
```

Applies the path to a node or a collection of nodes

```
(do
  (load-module :xml)
  (let [nodes (xml/parse-str "<a><b><c>C</c></b></a>")
        path [(xml/tag= "b")
               (xml/tag= "c")
               xml/text
               first]]
    (xml/path-> path nodes)))
=> "C"
```

xml/text

```
(xml/text nodes)
```

Returns a list of text contents of the XML nodes collection

```
(do
  (load-module :xml)
  (let [nodes (xml/parse-str "<a><b>B</b></a>")
        path [(xml/tag= "b")
               xml/text]]
    (xml/path-> path nodes)))
=> ("B")
```

zero?

```
(zero? x)
```

Returns true if x zero else false

```
(zero? 0)
=> true
```

```
(zero? 2)
=> false
```

```
(zero? 0I)
=> true
```

```
(zero? 0.0F)
=> true
```

```
(zero? 0.0)
=> true
```

```
(zero? 0.0M)
=> true
```

SEE ALSO

[neg?](#)

Returns true if x smaller than zero else false

[pos?](#)

Returns true if x greater than zero else false

[section](#)[top](#)

zipmap

```
(zipmap keys vals)
```

Returns a map with the keys mapped to the corresponding vals.

To create a list of tuples from two or more lists use

```
(map list '(1 2 3) '(4 5 6)) .
```

```
(zipmap [:a :b :c :d :e] [1 2 3 4 5])
=> {:a 1 :b 2 :c 3 :d 4 :e 5}
```

```
(zipmap [:a :b :c] [1 2 3 4 5])
=> {:a 1 :b 2 :c 3}
```

[section](#)[top](#)

zipvault/add-empty-folder

```
(zipvault/add-empty-folder zip passphrase name)
```

Adds an empty folder to the zip file.

```
(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;   └─ data/
  ;;   └─ doc/
  ;;   └─ setup/

  (let [zip (io/file "vault.zip")]
    (zipvault/add-empty-folder zip "pwd" "data")
    (zipvault/add-empty-folder zip "pwd" "doc/setup")))
```

SEE ALSO

zipvault/zip

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

zipvault/add-file

Adds a file to the zip.

zipvault/add-files

Adds a list of files to the zip.

zipvault/add-folder

Adds a folder to the zip file.

zipvault/add-stream

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

zipvault/remove-files

Removes all files from the zip file that match the names in the input list.

section

top

zipvault/add-file

```
(zipvault/add-file zip passphrase file)
(zipvault/add-file zip passphrase filename-in-zip file)
```

Adds a file to the zip.

`filename-in-zip` Set the filename that will be used to include a file into the ZIP file to a different name that given by the source filename added to the ZIP file. The `filename-in-zip` must adhere to the ZIP filename specification, including the use of forward slash '/' as the directory separator, and it must also be a relative file.

```
(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;      └─ a.txt
  ;;      └─ b.txt

  (let [zip (io/file "vault.zip")
        tmp-1 (io/file (io/tmp-dir) "a.txt")
        tmp-2 (io/file (io/tmp-dir) "b.txt")]
    (io/spit tmp-1 "1234")
    (io/spit tmp-2 "2345")
    (io/delete-file-on-exit tmp-1)
    (io/delete-file-on-exit tmp-2)

    (zipvault/add-file zip "pwd" tmp-1)
    (zipvault/add-file zip "pwd" tmp-2)))

(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;      └─ test/
  ;;          └─ aa1.txt
  ;;          └─ bb2.txt

  (let [zip (io/file "vault.zip")
        tmp-1 (io/file (io/tmp-dir) "a1.txt")
        tmp-2 (io/file (io/tmp-dir) "b2.txt")]
    (io/spit tmp-1 "1234")
    (io/spit tmp-2 "2345")
```

```
(io/delete-file-on-exit tmp-1)
(io/delete-file-on-exit tmp-2)

(zipvault/add-file zip "pwd" "test/aa1.txt" tmp-1)
(zipvault/add-file zip "pwd" "test/bb2.txt" tmp-2)))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/add-files](#)

Adds a list of files to the zip.

[zipvault/add-folder](#)

Adds a folder to the zip file.

[zipvault/add-empty-folder](#)

Adds an empty folder to the zip file.

[zipvault/add-stream](#)

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

[zipvault/remove-files](#)

Removes all files from the zip file that match the names in the input list.

[section](#)

[top](#)

zipvault/add-files

```
(zipvault/add-files zip passphrase root-folder-name-in-zip & files)
```

Adds a list of files to the zip.

'root-folder-name-in-zip' set the folder name that will be prepended to the filenames in the ZIP. Must be either `nil` or a non blank string!

```
(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;      ┌── a1.txt
  ;;      ├── b2.txt
  ;;      └── copies/
  ;;          ├── a1.txt
  ;;          └── b2.txt

  (let [zip (io/file "vault.zip")
        tmp-1 (io/file (io/tmp-dir) "a1.txt")
        tmp-2 (io/file (io/tmp-dir) "b2.txt")]
    (io/spit tmp-1 "1234")
    (io/spit tmp-2 "2345")
    (io/delete-file-on-exit tmp-1)
    (io/delete-file-on-exit tmp-2)

    (zipvault/zip zip "pwd" "a.txt" "A")
    (zipvault/add-files zip "pwd" nil tmp-1 tmp-2)
    (zipvault/add-files zip "pwd" "copies/" tmp-1 tmp-2)))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/add-file](#)

Adds a file to the zip.

[zipvault/add-folder](#)

Adds a folder to the zip file.

[zipvault/add-empty-folder](#)

Adds an empty folder to the zip file.

[zipvault/add-stream](#)

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

[zipvault/remove-files](#)

Removes all files from the zip file that match the names in the input list.

[section](#)

[top](#)

zipvault/add-folder

```
(zipvault/add-folder zip passphrase folder)
(zipvault/add-folder zip passphrase folder include-root-folder?)
(zipvault/add-folder zip passphrase folder include-root-folder? exclude-fn)
(zipvault/add-folder zip passphrase folder root-folder-name-in-zip include-root-folder? exclude-fn)
```

Adds a folder to the zip file.

'root-folder-name-in-zip' set the folder name that will be prepended to the filenames in the ZIP. Must be either `nil` or a non blank string!

If 'include-root-folder?' (default true) is true the root folder name will be added to the entry name as folder.

The 'exclude-fn' filters the files in the folder that are to be excluded from the zip. 'exclude-fn' is a single argument function that receives a file and returns true if the files is to be excluded otherwise it returns false. May be `nil`.

Note: To add a folder but without any files of the folder

```
(let [zip (io/file "data.zip")]
      (zipvault/add-folder zip "123" (io/file "/tmp/test/data") true (fn [x] true)))
```

This creates the zip "data.zip" with the empty folder "data".

```
(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;      └─ a1.txt
  ;;      └─ b2.txt

  (let [zip      (io/file "vault.zip")
        tmp-folder (io/file (io/tmp-dir) "data")
        tmp-1      (io/file tmp-folder "a1.txt")
        tmp-2      (io/file tmp-folder "b2.txt")]
    (io/mkdir tmp-folder)
    (io/spit tmp-1 "1234")
    (io/spit tmp-2 "2345")
    (io/delete-file-on-exit tmp-folder)

    (zipvault/add-folder zip "pwd" tmp-folder false)))

(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;      └─ data/
  ;;          └─ a1.txt
```



```
;;
      └─ b2.txt

(defn exclude-fn [file] (io/file-ext? file "log"))

(let [zip      (io/file "vault.zip")
      tmp-folder (io/file (io/tmp-dir) "data")
      tmp-1      (io/file tmp-folder "a1.txt")
      tmp-2      (io/file tmp-folder "b2.txt")
      tmp-3      (io/file tmp-folder "c3.log")]
  (io/mkdir tmp-folder)
  (io/spit tmp-1 "12")
  (io/spit tmp-2 "23")
  (io/spit tmp-3 "34")
  (io/delete-file-on-exit tmp-folder))

(zipvault/zip zip "pwd")
(zipvault/add-folder zip "pwd" tmp-folder true exclude-fn)))

(do
  (load-module :zipvault)

  ;; create "vault.zip"
  ;;
  ;;      └─ backup/
  ;;          └─ a1.txt
  ;;          └─ b2.txt
  ;;

  (let [zip      (io/file "vault.zip")
        tmp-folder (io/file (io/tmp-dir) "data")
        tmp-1      (io/file tmp-folder "a1.txt")
        tmp-2      (io/file tmp-folder "b2.txt")]
    (io/mkdir tmp-folder)
    (io/spit tmp-1 "12")
    (io/spit tmp-2 "23")
    (io/delete-file-on-exit tmp-folder))

    (zipvault/add-folder zip "pwd" tmp-folder "backup" true nil)))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/add-file](#)

Adds a file to the zip.

[zipvault/add-files](#)

Adds a list of files to the zip.

[zipvault/add-empty-folder](#)

Adds an empty folder to the zip file.

[zipvault/add-stream](#)

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

[zipvault/remove-files](#)

Removes all files from the zip file that match the names in the input list.

[section](#)

[top](#)

zipvault/add-stream

```
(zipvault/add-stream zip passphrase name is)
```

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

```
(do
  (load-module :zipvault)

  (let [zip (io/file "vault.zip")
        is (io/string-in-stream "100")]
    (zipvault/add-stream zip "pwd" "100.txt" is)))

;; create "vault.zip"
;;      └─ a.txt
;;      └─ data/
;;          └─ b.txt
;;          └─ c.txt

(let [zip (io/file "vault.zip")
      is1 (io/string-in-stream "100")
      is2 (io/string-in-stream "200")
      is3 (io/string-in-stream "300")]
  (zipvault/add-stream zip "pwd" "a.txt" is1)
  (zipvault/add-stream zip "pwd" "data/b.txt" is2)
  (zipvault/add-stream zip "pwd" "data/c.txt" is3)))

;; create "vault.zip"
;;      └─ data/
;;          └─ a.txt
;;          └─ old/
;;              └─ b.txt
;;              └─ c.txt

(let [zip (io/file "vault.zip")
      is1 (io/string-in-stream "100")
      is2 (io/string-in-stream "200")
      is3 (io/string-in-stream "300")]
  (zipvault/add-stream zip "pwd" "data/a.txt" is1)
  (zipvault/add-stream zip "pwd" "data/old/b.txt" is2)
  (zipvault/add-stream zip "pwd" "data/old/c.txt" is3)))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/add-file](#)

Adds a file to the zip.

[zipvault/add-files](#)

Adds a list of files to the zip.

[zipvault/add-folder](#)

Adds a folder to the zip file.

[zipvault/add-empty-folder](#)

Adds an empty folder to the zip file.

[zipvault/remove-files](#)

Removes all files from the zip file that match the names in the input list.

zipvault/encrypted?

```
(zipvault/encrypted? zip)
```

Extracts a specific file from the zip file to the destination path.

```
(do
  (load-module :zipvault)

  (zipvault/zip (io/file "vault.zip") "pwd" "a.txt" "abc")
  (zipvault/encrypted? (io/file "vault.zip")))
```

zipvault/entropy

```
(zipvault/entropy passphrase)
```

Returns the passphrase's entropy in bits.

The password entropy using the formula: $E = \log_2(RL)$

- **E** stands for password entropy, measured in bits
- **Log₂** is a mathematical formula that converts the total number of possible character combinations to bits
- **R** stands for the range of characters
- **L** stands for the number of characters in a password

The entropy is calculated based on 26 lower and upper case letters, 10 digits, and 24 symbols like `°+*%&/()=?'^^_.,-$£!#~;`

Note: The function just calculates the entropy. A strong passphrase does not rely on the entropy solely. Avoid passphrases containing words from the dictionary ("admin_passw0rd"), dates (birthdate, ...), repetitions ("aaaaa"), or sequences ("123456")!

```
(do
  (load-module :zipvault)
  (zipvault/entropy "uibsd6b38hs7b_La'sdgk898wbver"))
=> 186.36167788636087
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

zipvault/extract-all

```
(zipvault/extract-all zip destpath)
(zipvault/extract-all zip passphrase destpath)
```

Extracts all files from the zip file to the destination path.

```
(do
  (load-module :zipvault)

  (zipvault/zip (io/file "vault.zip")
    "pwd"
    "a.txt" "abc"
    "b.txt" "def")

  (zipvault/extract-all (io/file "vault.zip")
    "pwd"
    "."))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/extract-file](#)

Extracts a specific file or folder from the zip file to the destination path.

[zipvault/extract-file-data](#)

Extracts a specific file from the zip file and returns it as binary data. in may be a file or an input stream.

[section](#)[top](#)

zipvault/extract-file

```
(zipvault/extract-file zip password filename destpath)
```

Extracts a specific file or folder from the zip file to the destination path.

```
(do
  (load-module :zipvault)

  (zipvault/zip (io/file "vault.zip")
    "pwd"
    "a.txt" "abc"
    "b.txt" "def")

  ;; extract a file
  (zipvault/extract-file (io/file "vault.zip")
    "pwd"
    "a.txt"
    "."))

(do
  (load-module :zipvault)

  (zipvault/zip (io/file "vault.zip")
    "pwd"
    "words/one.txt" "one"
    "words/two.txt" "two"
    "logs/001.log" "xxx")

  ;; extract a folder
  (zipvault/extract-file (io/file "vault.zip")
    "pwd"
    "words/"
    "."))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/extract-all](#)

Extracts all files from the zip file to the destination path.

[zipvault/extract-file-data](#)

Extracts a specific file from the zip file and returns it as binary data. in may be a file or an input stream.

[section](#)[top](#)

zipvault/extract-file-data

```
(zipvault/extract-file-data in passphrase filename)
```

Extracts a specific file from the zip file and returns it as binary data. in may be a file or an input stream.

Returns `nil` if the file does not exist.

```
(do
  (load-module :zipvault)

  (zipvault/zip (io/file "vault.zip")
    "pwd"
    "a.txt" "abc"
    "b.txt" "def")

  (zipvault/extract-file-data (io/file "vault.zip")
    "pwd"
    "a.txt"))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/extract-file](#)

Extracts a specific file or folder from the zip file to the destination path.

[zipvault/extract-all](#)

Extracts all files from the zip file to the destination path.

[section](#)[top](#)

zipvault/remove-files

```
(zipvault/remove-files zip passphrase & files)
```

Removes all files from the zip file that match the names in the input list.

If any of the files is a directory, all the files and directories under this directory will be removed as well.

```
(do
  (load-module :zipvault)
```

```
(let [zip (io/file "vault.zip")]
  (zipvault/zip zip "pwd" "a.txt" "A" "b.txt" "B")
  (zipvault/remove-files zip "pwd" "a.txt"))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/add-file](#)

Adds a file to the zip.

[zipvault/add-files](#)

Adds a list of files to the zip.

[zipvault/add-folder](#)

Adds a folder to the zip file.

[zipvault/add-stream](#)

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

[section](#)[top](#)

zipvault/valid-zip-file?

```
(zipvault/valid-zip-file? zip)
```

Returns true if the zip is a valid zip file else false.

```
(do
  (load-module :zipvault)

  (zipvault/zip (io/file "vault.zip") "pwd" "a.txt" "abc")
  (zipvault/valid-zip-file? (io/file "vault.zip")))
```

[section](#)[top](#)

zipvault/zip

```
(zipvault/zip out passphrase & entries)
```

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

An entry is given by a name and data. The entry data may be nil, a bytebuf, a string, a file, an input stream, or a producer function. An entry name with a trailing '/' creates a directory.

Entry value types:

nil	an empty file is written to the zip entry
bytebuf	the bytes are written to the zip entry
string	the string is written to the zip entry
file	the content of the file is written to the zip entry
input stream	the slurped input stream data is written to the zip entry
function	a producer function with a single output stream argument. All data written to the stream is written to the zip entry. The stream can be flushed but must not be closed!

Passphrases:

The AES-256 algorithm requires a 256-bit key as input. One should use a passphrase with at least 128 bits of entropy (that's roughly a 20-character passphrase of random upper/lower/digits/symbols). Less is dropping below general limits of safety, and more than 256 bits won't accomplish anything.

See function: `zipvault/entropy`

```
(do
  (load-module :zipvault)
  (zipvault/zip (io/file "vault.zip") "pwd")) ; empty zip

(do
  (load-module :zipvault)
  (zipvault/zip (io/file "vault.zip") "pwd" "a.txt" "abc"))

(do
  (load-module :zipvault)
  (zipvault/zip (io/file-out-stream "vault.zip")
    "pwd"
    "a.txt"      "abc"
    "b.txt"      (bytebuf [100 101 102])))

(do
  (load-module :zipvault)

  (let [file (io/file (io/tmp-dir) "c.txt")]
    (io/spit file "1234")
    (io/delete-file-on-exit c-tmp))

  ;; create "vault.zip"
  ;;      └─ a.txt
  ;;      └─ b.txt
  ;;      └─ c.txt
  ;;      └─ d.txt
  ;;      └─ e.txt
  ;;      └─ empty.txt
  ;;      └─ xx/
  ;;           └─ g.txt
  (zipvault/zip
    (io/file "vault.zip")
    "pwd"
    "a.txt"      "abc"
    "b.txt"      (bytebuf "def")
    file         file ; equivalent: (io/file-basename file) file
    "d.txt"      (io/string-in-stream "ghi")
    "e.txt"      (fn [os]
      (let [wr (io/wrap-os-with-buffered-writer os)]
        (println wr "200")
        (flush wr)))
    "empty.txt"  nil
    "xx/g.txt"   "jkl")))
```

SEE ALSO

[zipvault/zip-folder](#)

Creates an AES-256 encrypted and password protected zip from the folder.

[zipvault/entries](#)

Returns a list of the entry names in the zip.

[zipvault/add-file](#)

Adds a file to the zip.

[zipvault/add-files](#)

Adds a list of files to the zip.

[zipvault/add-folder](#)

Adds a folder to the zip file.

[zipvault/add-empty-folder](#)

Adds an empty folder to the zip file.

[zipvault/add-stream](#)

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

[zipvault/remove-files](#)

Removes all files from the zip file that match the names in the input list.

[zipvault/extract-file](#)

Extracts a specific file or folder from the zip file to the destination path.

[zipvault/extract-all](#)

Extracts all files from the zip file to the destination path.

[zipvault/extract-file-data](#)

Extracts a specific file from the zip file and returns it as binary data. in may be a file or an input stream.

[zipvault/entropy](#)

Returns the passphrase's entropy in bits.

[section](#)[top](#)

zipvault/zip-folder

```
(zipvault/zip-folder out passphrase folder)
(zipvault/zip-folder out passphrase folder include-root-folder?)
(zipvault/zip-folder out passphrase folder include-root-folder? exclude-fn)
```

Creates an AES-256 encrypted and password protected zip from the folder.

If 'include-root-folder?' (default true) is true the root folder name will be added to the entry name as folder.

The 'exclude-fn' filters the files in the folder that are to be excluded from the zip. 'exclude-fn' is a single argument function that receives a file and returns true if the files is to be excluded otherwise it returns false.

```
(do
  (load-module :zipvault)

  (let [zip      (io/file "vault.zip")
        tmp-folder (io/file (io/tmp-dir) "ziptest")
        tmp-1     (io/file tmp-folder "a1.txt")
        tmp-2     (io/file tmp-folder "a2.txt")]
    (io/mkdir tmp-folder)
    (io/spit tmp-1 "1234")
    (io/spit tmp-2 "2345")
    (io/delete-file-on-exit tmp-folder)

    (zipvault/zip-folder zip "pwd" tmp-folder)))

(do
  (load-module :zipvault)

  (defn exclude-fn [file] (io/file-ext? file "log"))

  (let [zip      (io/file "vault.zip")
        tmp-folder (io/file (io/tmp-dir) "ziptest")
        tmp-1     (io/file tmp-folder "a.txt")
        tmp-2     (io/file tmp-folder "b.txt")
        tmp-3     (io/file tmp-folder "c.log")]
```



```
(io/mkdir tmp-folder)
(io/spit tmp-1 "12")
(io/spit tmp-2 "23")
(io/spit tmp-3 "34")
(io/delete-file-on-exit tmp-folder)

(zipvault/zip-folder zip "pwd" tmp-folder true exclude-fn))
```

SEE ALSO

[zipvault/zip](#)

Creates an AES-256 encrypted and password protected zip form the entries and writes it to out. out may be a file or an output stream.

[zipvault/add-file](#)

Adds a file to the zip.

[zipvault/add-files](#)

Adds a list of files to the zip.

[zipvault/add-folder](#)

Adds a folder to the zip file.

[zipvault/add-empty-folder](#)

Adds an empty folder to the zip file.

[zipvault/add-stream](#)

Creates a new entry in the zip file and adds the content of the input stream to the zip file.

[section](#)

[top](#)



Creates a hash map.

```
{:a 10 :b 20}
=> {:a 10 :b 20}
```