

Overview

Primitives	Literals	Numbers	Strings	Chars	Other					
Collections	List	Vector	Set	Map	LazySeq	Stack	Queue	DAG	Array	ByteBuf
Custom Types	Types	Protocols								
Core Functions	Functions	Macros	Special Forms	Transducers	Namespaces	Exceptions				
Concurrency	Atoms	Futures	Promises	Delay	Agents	Scheduler	Locking	Volatiles	Parallel	
Threads	ThreadLocal	Threads								
System & Java	System	System Vars	Java Interop	REPL						
Util	Math	Time	Regex	CIDR						
I/O	I/O	File	Zip/GZip							
Documents	JSON	PDF	PDF Tools	CSV	XML	Excel				
Modules	Kira Templates Shell	Parsifal Geo IP Ansi	Configuration Gradle Maven	Component Tracing	XML Benchmark	Cryptography App	Java	Semver	Hexdump	
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
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

Collections

Collections

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

Lists

Create	()	list	list*	mutable-list
Access	first last nlast	second peek sublist	third rest some	fourth butlast nth nfirst
Modify	cons concat partition-by	conj distinct interpose	rest dedupe interleave	pop partition interleave

Arithmetic	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>
Convert	<code>int</code>	<code>long</code>	<code>double</code>	<code>decimal</code> <code>bigint</code>
Compare	<code>==</code>	<code>=</code>	<code>&lt;</code>	<code>&gt;</code> <code>&lt;=</code> <code>&gt;=</code> <code>compare</code>
Test	<code>zero?</code>	<code>pos?</code>	<code>neg?</code>	<code>even?</code> <code>odd?</code> <code>number?</code> <code>int?</code> <code>long?</code> <code>double?</code> <code>decimal?</code>
NaN/Infinite	<code>nan?</code>	<code>infinite?</code>		
BigDecimal	<code>dec/add</code> <code>dec/div</code>	<code>dec/sub</code> <code>dec/scale</code>	<code>dec/mul</code>	

### Strings

Create	<code>str</code>	<code>str/format</code>	<code>str/quote</code> <code>str/double-quote</code> <code>str/double-unquote</code>
Use	<code>count</code>	<code>compare</code>	<code>empty-to-nil</code> <code>first</code> <code>last</code> <code>nth</code> <code>nfirst</code> <code>nlast</code> <code>seq</code> <code>rest</code> <code>butlast</code> <code>reverse</code> <code>shuffle</code> <code>str/index-of</code> <code>str/last-index-of</code> <code>str/subs</code> <code>str/nfirst</code> <code>str/nlast</code> <code>str/rest</code> <code>str/butlast</code> <code>str/chars</code> <code>str/pos</code> <code>str/repeat</code> <code>str/reverse</code> <code>str/truncate</code> <code>str/expand</code> <code>str/lorem-ipsum</code>
Split/Join	<code>str/split</code>	<code>str/split-lines</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>	
Conversion	<code>str/lower-case</code> <code>str/cr-lf</code>	<code>str/upper-case</code>	
Regex	<code>match?</code>	<code>not-match?</code>	
Trim	<code>str/trim</code>	<code>str/trim-to-nil</code> <code>str/trim-left</code> <code>str/trim-right</code>	
Hex	<code>str/hex-to-bytebuf</code> <code>str/bytebuf-to-hex</code> <code>str/format-bytebuf</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>		
Validation	<code>str/valid-email-addr?</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&gt;equals-ignore-case?</code> <code>str/quoted?</code> <code>str/double-quoted?</code>	

	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?	every?
	not-every?	any?	not-any?

### Vectors

Create	<code>[]</code>	<code>vector</code>	<code>vector*</code>	<code>mutable-vector</code> <code>mapv</code>
Access	<code>first</code> <code>peek</code> <code>subvec</code>	<code>second</code> <code>butlast</code> <code>some</code>	<code>third</code> <code>rest</code>	<code>nth</code> <code>last</code> <code>nfirst</code> <code>nlast</code>
Modify	<code>cons</code> <code>concat</code> <code>partition-by</code> <code>cartesian-product</code> <code>mapcat</code> <code>take</code> <code>drop-while</code> <code>update!</code>	<code>conj</code> <code>distinct</code> <code>interpose</code> <code>combinations</code> <code>flatten</code> <code>take-while</code> <code>drop-last</code> <code>assoc</code>	<code>rest</code> <code>dedupe</code> <code>interleave</code> <code>update</code> <code>assoc!</code>	<code>pop</code> <code>into</code> <code>partition</code> <code>interleave</code> <code>split-with</code>
Nested	<code>get-in</code> <code>dissoc-in</code>	<code>assoc-in</code>	<code>update-in</code>	
Test	<code>vector?</code> <code>not-contains?</code> <code>any?</code>	<code>mutable-vector?</code> <code>every?</code> <code>not-every?</code>	<code>contains?</code> <code>not-every?</code>	

### Sets

Create	<code>#{} </code>	<code>set</code>	<code>sorted-set</code>	<code>mutable-set</code>
Modify	<code>cons</code>	<code>cons!</code>	<code>conj</code> <code>conj!</code>	<code>disj</code>
Algebra	<code>difference</code> <code>subset?</code>	<code>union</code> <code>superset?</code>	<code>intersection</code>	
Test	<code>set?</code> <code>contains?</code> <code>not-every?</code>	<code>sorted-set?</code> <code>not-contains?</code> <code>any?</code>	<code>mutable-set?</code> <code>every?</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>update!</code> <code>concat</code> <code>reduce-kv</code> <code>merge-deep</code> <code>map-vals</code>	<code>conj</code> <code>dissoc</code> <code>flatten</code> <code>merge</code> <code>map-invert</code> <code>select-keys</code>	<code>assoc</code> <code>assoc!</code> <code>dissoc!</code> <code>into</code> <code>filter-k</code> <code>filter-kv</code> <code>merge-with</code> <code>map-keys</code>	<code>update</code> <code>map-keys</code>
Entries	<code>map-entry</code> <code>map-entry?</code>	<code>key</code>	<code>val</code>	<code>entries</code>
Nested	<code>get-in</code> <code>dissoc-in</code>	<code>assoc-in</code>	<code>update-in</code>	

Test char	str/char?	str/digit?	str/hexdigit?	str/letter?	str/whitespace?	str/linefeed?	str/lower-case?	str/upper-case?
Other	str/levenshtein							
Chars								
Use	char	char?	char-literals					
Conversion	str	str/lower-case	str/upper-case					
Test char	str/char?	str/digit?	str/letter?	str/whitespace?	str/linefeed?	str/lower-case?	str/upper-case?	
Other								
Nil	nil?	some?						
Keywords	:a :blue	keyword?	keyword					
Symbols	'a 'blue	symbol?	symbol					
Just	just	just?						
Boolean	boolean	not	boolean?	true?	false?			

## Byte Buffer

Create	bytebuf	bytebuf-allocate	bytebuf-from-string	
Test	empty?	not-empty?	bytebuf?	
Use	count	bytebuf-capacity	bytebuf-limit	bytebuf-to-string bytebuf-to-list bytebuf-pos bytebuf-pos!
Read	bytebuf-get-byte	bytebuf-get-int	bytebuf-get-long	bytebuf-get-float bytebuf-get-double
Write	bytebuf-put-byte!	bytebuf-put-int!	bytebuf-put-long!	bytebuf-put-float! bytebuf-put-double! bytebuf-put-buf!
Base64	str/encode-base64	str/decode-base64		
Hex	str/hex-to-bytebuf	str/bytebuf-to-hex	str/format-bytebuf	

## Regex

General	regex/pattern	regex/matcher	regex/reset	regex/matches?	regex/matches	regex/group
---------	---------------	---------------	-------------	----------------	---------------	-------------

Test	map?	sequential?	hash-map?	
	ordered-map?	sorted-map?		
	mutable-map?	contains?		
	not-contains?			
Stack				
Create	stack			
Access	peek	pop!	push!	count
Test	empty?	stack?		
Queue				
Create	queue			
Access	peek	poll!	offer!	count
Test	empty?	queue?		
DAG (directed acyclic graph)				
Create	dag/dag	dag/add-edges	dag/add-nodes	
Access	dag/nodes	dag/edges	dag/roots	count
Children	dag/children	dag/direct-children		
Parents	dag/parents	dag/direct-parents		
Sort	dag/topological-sort	dag/compare-fn		
Test	dag/dag?	dag/node?	dag/parent-of?	dag/child-of? empty?

## Lazy Sequences

Create	lazy-seq			
Realize	doall			
Test	lazy-seq?			

## Arrays

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

## Concurrency

--	--	--	--	--	--	--	--

```
regex/count  regex/find?  regex/find
regex/find-all  regex/find+
regex/find-all+
```

## Math

Arithmetic	mod  inc  dec  min  max  abs sgn  negate  floor  ceil  sqrt square  pow  exp  log  log10
Util	digits
Random	rand-long  rand-double 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?

## Functions

Create	fn  defn  defn-  identity comp  partial  memoize  juxt fn!  trampoline  complement constantly  every-pred any-pred
Call	apply  ->  -->
Test	fn?
Misc	nil?  some?  eval  name namespace  fn-name  callstack coalesce  load-resource

## Atoms

	atom  atom?  deref  deref?
	reset!  swap!  swap-vals!
	compare-and-set!  add-watch
	remove-watch
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
Scheduler	schedule-delay schedule-at-fixed-rate
Locking	locking
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-id  thread-name thread-daemon?  thread-interrupted? thread-interrupted
Parallel	pmap  pcalls

## System

Venice	version  sandboxed?  sandbox-type
System	system-prop  system-env system-exit-code charset-default-encoding
Java	java-version  java-version-info java-major-version java-source-location

Environment	set! resolve bound?
	var-get var-name var-ns
	var-thread-local? var-local?
	var-global? name namespace
Tree Walker	prewalk postwalk prewalk-replace postwalk-replace
Meta	meta with-meta vary-meta
Documentation	doc modules
Definiton	fn-body fn-pre-conditions
Syntax	highlight

## Macros

Create	def- defn defn- defmacro macroexpand macroexpand-all macro?
Test	macro?
Quoting	quote quasiquote
Branch	and or when when-not if-not if-let when-let letfn
Conditions	cond condp case
Loop	while dotimes list-comp doseq
Call	doto -> ->> -<> as-> cond-> cond->> some-> some->>
Loading	load-module load-file load-classpath-file load-string
Assert	assert
Util	comment gensym time with-out-str with-err-str
Profiling	time perf

## Special Forms

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

## Exceptions

Java VM	pid gc total-memory used-memory
OS	os-type os-type? os-arch os-name os-version
Time	current-time-millis nano-time format-nano-time format-micro-time format-milli-time
Host	host-name host-address ip-private? cpus
User	user-name io/user-home-dir
Util	uuid sleep shutdown-hook
Shell	sh with-sh-dir with-sh-env with-sh-throw
Shell Tools	sh/open sh/pwd

## System Vars

System Vars	*version* *newline* *loaded-modules* *loaded-files* *ns* *run-mode* *ansi-term*
-------------	---

## Time

Date	time/date time/date?
Local Date	time/local-date time/local-date? time/local-date-parse
Local Date Time	time/local-date-time time/local-date-time? time/local-date-time-parse
Zoned Date Time	time/zoned-date-time time/zoned-date-time? time/zoned-date-time-parse
Fields	time/year time/month time/day-of-week time/day-of-month time/day-of-year time/hour time/minute time/second
Fields etc	time/length-of-year time/length-of-month time/first-day-of-month time/last-day-of-month
Zone	time/zone time/zone-offset
Format	time/formatter time/format
Test	time/after? time/not-after? time/before? time/not-before? time/within? time/leap-year?
Miscellaneous	

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

## Types

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

## Protocols

Core	Object
------	--------

## Namespace

Open	ns		
Current	*ns*		
Remove	ns-unmap	ns-remove	
Util	ns-list	namespace	
Alias	ns-alias	ns-aliases	ns-unalias

## Java Interoperability

Java	. import	java-iterator-to-list		
		java-enumeration-to-list		
		java-unwrap-optional	cast	class
Proxify	proxify	as-runnable	as-callable	
	as-predicate	as-function		
	as-consumer	as-supplier		
	as-bipredicate	as-bifunction		
	as-biconsumer	as-binaryoperator		
Test	java-obj?	exists-class?		
Support	imports	supers	bases	formal-type
	stacktrace			
Classes	class	class-of	class-name	
	class-version	classloader		
	classloader-of			
JARs				

	time/with-time	time/plus
	time/minus	time/period
	time/earliest	time/latest
Util	time/zone-ids	time/to-millis

## I/O

to	print	println	printf	flush
	newline	pr	prn	
to-str	pr-str	with-out-str		
from	read-line	read-string		
classpath	io/load-classpath-resource	io/classpath-resource?		
stream	io/slurp	io/slurp-lines		
	io/copy-stream	io/slurp-stream		
	io/spit-stream	io/uri-stream		
	io/file-in-stream			
	io/string-in-stream			
	io/bytebuf-in-stream			
	io/wrap-os-with-buffered-writer			
	io/wrap-os-with-print-writer			
	io/wrap-is-with-buffered-reader			
reader/writer	io/buffered-reader	io/buffered-writer		
http	io/download	io/internet-avail?		
other	with-out-str	io/mime-type		
	io/default-charset			

## File I/O

file	io/file	io/file-parent		
	io/file-name	io/file-path		
	io/file-absolute-path			
	io/file-canonical-path	io/file-ext		
	io/file-ext?	io/file-size		
	io/file-last-modified			
file dir	io/mkdir	io/mkdirs		
file i/o	io/slurp	io/slurp-lines	io/spit	
	io/copy-file	io/move-file		
	io/delete-file			
	io/delete-file-on-exit			
	io/delete-file-tree			
file list	io/list-files	io/list-files-glob		
	io/list-file-tree			
file test	io/file?	io/exists-file?		
	io/exists-dir?	io/file-can-read?		
	io/file-can-write?			
	io/file-can-execute?	io/file-hidden?		
	io/file-symbolic-link?			
URL/URI	io/->url	io/->uri		
file watch				

	jar-maven-manifest-version
	java-package-version
Modules	module-name

REPL		
Info	repl/info	
Terminal	repl/term-rows	repl/term-cols

PDF			
PDF	pdf/render	pdf/text-to-pdf	
	pdf/available?		
	pdf/check-required-libs		
PDF Tools	pdf/merge	pdf/copy	pdf/pages
	pdf/watermark		
Required 3rd party libraries:			
<ul style="list-style-type: none"> <li>org.xhtmlrenderer:flyiing-saucer-core:9.1.22</li> <li>org.xhtmlrenderer:flyiing-saucer-pdf-openpdf:9.1.22</li> <li>com.github.librepdf:openpdf:1.3.26</li> <li>com.github.librepdf:pdf-toolbox:1.3.26</li> </ul>			

Zip/GZip			
zip	io/zip	io/zip-file	io/zip-list
	io/zip-list-entry-names	io/zip-append	
	io/zip-remove	io/zip?	io/unzip
	io/unzip-first	io/unzip-nth	
	io/unzip-all	io/unzip-to-dir	
gzip	io/gzip	io/gzip-to-stream	io/gzip?
	io/ungzip	io/ungzip-to-stream	

	io/await-for	io/watch-dir
	io/close-watcher	
file other	io/temp-file	io/tmp-dir
	io/user-dir	io/user-home-dir

JSON		
read	json/read-str	json/slurp
write	json/write-str	json/spit
prettify	json/pretty-print	

CIDR (classless inter-domain routing)			
CIDR	cidr/parse	cidr/in-range?	
	cidr/start-inet-addr		
	cidr/end-inet-addr	cidr/inet-addr	
	cidr/inet-addr-to-bytes		
	cidr/inet-addr-from-bytes		
CIDR Trie	cidr/trie	cidr/size	cidr/insert
	cidr/lookup	cidr/lookup-reverse	

Application		
Management	app/build	app/manifest

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

## Modules

### Kira

Templating system

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

Kira	<a href="#">kira/eval</a>	<a href="#">kira/fn</a>
Escape	<a href="#">kira/escape-xml</a>	<a href="#">kira/escape-html</a>

### Cryptography

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

Hashes	<a href="#">crypt/md5-hash</a>	<a href="#">crypt/sha1-hash</a>
	<a href="#">crypt/sha512-hash</a>	<a href="#">crypt/pbkdf2-hash</a>
Encrypt	<a href="#">crypt/encrypt</a>	<a href="#">crypt/decrypt</a>

### XML

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

XML	<a href="#">xml/parse-str</a>	<a href="#">xml/parse</a>	<a href="#">xml/path-&gt;</a>
	<a href="#">xml/children</a>	<a href="#">xml/text</a>	

### Java

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

Java	<a href="#">java/javadoc</a>
------	------------------------------

### Parsifal

A parser combinator

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

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

Run	<a href="#">parsifal/run</a>
Define	<a href="#">parsifal/defparser</a>
Parsers	<a href="#">parsifal/any</a> <a href="#">parsifal/many</a> <a href="#">parsifal/many1</a> <a href="#">parsifal/times</a> <a href="#">parsifal/either</a> <a href="#">parsifal/choice</a> <a href="#">parsifal/between</a> <a href="#">parsifal/&gt;&gt;</a>
Special Parsers	<a href="#">parsifal/eof</a> <a href="#">parsifal/never</a> <a href="#">parsifal/always</a> <a href="#">parsifal/lookahead</a> <a href="#">parsifal/attempt</a>
Binding	<a href="#">parsifal/let-&gt;&gt;</a>

### Hexdump

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

Hexdump	<a href="#">hexdump/dump</a>
---------	------------------------------

### Semver

Semantic versioning

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

Semver	<a href="#">semver/parse</a>	<a href="#">semver/version</a>
Validation	<a href="#">semver/valid?</a>	<a href="#">semver/valid-format?</a>
Test	<a href="#">semver/newer?</a>	<a href="#">semver/older?</a>
	<a href="#">semver/equal?</a>	<a href="#">semver/cmp</a>

### Geo IP

Geolocation mapping for IP addresses

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

Lookup	<a href="#">geoip/ip-to-country-resolver</a> <a href="#">geoip/ip-to-country-loc-resolver</a> <a href="#">geoip/ip-to-city-loc-resolver</a> <a href="#">geoip/ip-to-city-loc-resolver-mem-optimized</a>
Databases	<a href="#">geoip/download-google-country-db-to-csvfile</a> <a href="#">geoip/download-maxmind-db-to-zipfile</a> <a href="#">geoip/download-maxmind-db</a>
DB Parser	<a href="#">geoip/parse-maxmind-country-ip-db</a> <a href="#">geoip/parse-maxmind-city-ip-db</a> <a href="#">geoip/parse-maxmind-country-db</a> <a href="#">geoip/parse-maxmind-city-db</a>
Util	<a href="#">geoip/build-maxmind-country-db-url</a> <a href="#">geoip/build-maxmind-city-db-url</a> <a href="#">geoip/map-location-to-numeric</a> <a href="#">geoip/country-to-location-resolver</a>

### Excel

Read/Write Excel files

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

Writer	<a href="#">excel/writer</a>	<a href="#">excel/add-sheet</a> <a href="#">excel/add-font</a> <a href="#">excel/add-style</a> <a href="#">excel/add-column</a>
Writer Data	<a href="#">excel/write-data</a> <a href="#">excel/write-item</a>	<a href="#">excel/write-items</a> <a href="#">excel/write-value</a>



Char Parsers	<p> <a href="#">parsifal/char</a> <a href="#">parsifal/not-char</a> </p> <p> <a href="#">parsifal/any-char</a> <a href="#">parsifal/digit</a> </p> <p> <a href="#">parsifal/hexdigit</a> </p> <p> <a href="#">parsifal/letter</a> </p> <p> <a href="#">parsifal/letter-or-digit</a> </p> <p> <a href="#">parsifal/any-char-of</a> </p> <p> <a href="#">parsifal/none-char-of</a> </p> <p> <a href="#">parsifal/string</a> </p>
Token Parsers	<p> <a href="#">parsifal/token</a> </p>
Protocols	<p> <a href="#">parsifal/SourcePosition</a> </p>
Line Info	<p> <a href="#">parsifal/lineno</a> <a href="#">parsifal/pos</a> </p>

## Gradle

(load-module :gradle)

Gradle	<a href="#">gradle/with-home</a> <a href="#">gradle/version</a>
	<a href="#">gradle/task</a>

## Maven

(load-module :maven)

Maven	<a href="#">maven/download</a> <a href="#">maven/get</a> <a href="#">maven/uri</a>
	<a href="#">maven/parse-artefact</a>

## Tracing

Tracing functions

(load-module :trace)

Tracing	<a href="#">trace/trace</a> <a href="#">trace/trace-var</a>
	<a href="#">trace/untrace-var</a>
Test	<a href="#">trace/traced?</a> <a href="#">trace/traceable?</a>
Util	<a href="#">trace/trace-str-limit</a>
Tee	<a href="#">trace/tee-&gt;</a> <a href="#">trace/tee-&gt;&gt;</a> <a href="#">trace/tee</a>

## Shell

Functions to deal with the operating system

(load-module :shell)

Open	<a href="#">shell/open</a> <a href="#">shell/open-macos-app</a>
Process	<a href="#">shell/kill</a> <a href="#">shell/kill-forcibly</a>
	<a href="#">shell/wait-for-process-exit</a>
	<a href="#">shell/alive?</a> <a href="#">shell/pid</a>
	<a href="#">shell/process-handle</a>
	<a href="#">shell/process-handle?</a>
	<a href="#">shell/process-info</a> <a href="#">shell/processes</a>

Writer I/O	<a href="#">excel/write-&gt;file</a>
	<a href="#">excel/write-&gt;stream</a>
	<a href="#">excel/write-&gt;bytebuf</a>
Writer Util	<a href="#">excel/cell-formula</a>
	<a href="#">excel/sum-formula</a>
	<a href="#">excel/cell-address</a>
	<a href="#">excel/auto-size-columns</a>
	<a href="#">excel/auto-size-column</a>
	<a href="#">excel/row-height</a>
	<a href="#">excel/evaluate-formulas</a>
	<a href="#">excel/convert-&gt;reader</a>
Reader	<a href="#">excel/open</a> <a href="#">excel/sheet</a>
	<a href="#">excel/read-string-val</a>
	<a href="#">excel/read-boolean-val</a>
	<a href="#">excel/read-long-val</a>
	<a href="#">excel/read-double-val</a>
	<a href="#">excel/read-date-val</a>
Reader Util	<a href="#">excel/sheet-count</a> <a href="#">excel/sheet-name</a>
	<a href="#">excel/sheet-row-range</a>
	<a href="#">excel/sheet-col-range</a>
	<a href="#">excel/evaluate-formulas</a>
	<a href="#">excel/cell-empty?</a> <a href="#">excel/cell-type</a>

Required 3rd party libraries:

- [org.apache.poi:poi:4.1.2](#)
- [org.apache.poi:poi-ooxml:4.1.2](#)
- [org.apache.poi:poi-ooxml-schemas:4.1.2](#)
- [commons-codec:commons-codec:1.15](#)
- [org.apache.commons:commons-collections:4.4.4](#)
- [org.apache.commons:commons-compress:1.20](#)
- [org.apache.commons:commons-math3:3.6.1](#)
- [org.apache.xmlbeans:xmlbeans:3.1.0](#)

## Configuration

Manages configurations with system property & env var support

(load-module :config)

Build	<a href="#">config/build</a>
File	<a href="#">config/file</a> <a href="#">config/resource</a>
Env	<a href="#">config/env-var</a> <a href="#">config/env</a>
Properties	<a href="#">config/property-var</a>
	<a href="#">config/properties</a>

## Component

Managing lifecycle and dependencies of components

(load-module :component)

Build	<a href="#">component/system-map</a>
	<a href="#">component/system-using</a>
Protocol	<a href="#">component/Component</a>

	<code>shell/processes-info</code> <code>shell/descendant-processes</code> <code>shell/parent-process</code>
Util	<code>shell/diff</code>

Ansi

ANSI codes, styles, and colorization helper functions

(load-module :ansi)

Colors	<code>ansi/fg-color</code>	<code>ansi/bg-color</code>
Styles	<code>ansi/style</code>	<code>ansi/ansi</code> <code>ansi/with-ansi</code> <code>ansi/without-ansi</code>
Cursor	<code>ansi/without-cursor</code>	
Progress	<code>ansi/progress</code>	<code>ansi/progress-bar</code>

Util	<code>component/deps</code>	<code>component/dep</code> <code>component/id</code>
------	-----------------------------	---

App

Venice application archive

(load-module :app)

Build	<code>app/build</code>
Manifest	<code>app/manifest</code>

Benchmark

(load-module :benchmark)

Utils	<code>benchmark/benchmark</code>
-------	----------------------------------

## Embedding in Java

### Eval

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

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

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

### Passing parameters

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

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

        Long val = (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.Venice;
import com.github.jlangch.venice.Parameters;

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));
    }
}
```

### Precompiled

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

public class Example {
    public static void main(String[] args) {
```

```

    Venice venice = new Venice();

    PreCompiled 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.Venice;
import com.github.jlangch.venice.javainterop.*;

public class Example {
    public static void main(String[] args) {
        final IInterceptor interceptor =
            new SandboxInterceptor(
                new SandboxRules()
                    .rejectAllVeniceIoFunctions()
                    .allowAccessToStandardSystemProperties()
                    .withClasses(
                        "java.lang.Math:min",
                        "java.time.ZonedDateTime:*",
                        "java.util.ArrayList:new",
                        "java.util.ArrayList:add"));

        final Venice venice = new Venice(interceptor);

        // => OK (static method)
        venice.eval("( . :java.lang.Math :min 20 30)");

        // => OK (constructor & instance method)
        venice.eval("( . (. :java.time.ZonedDateTime :now) :plusDays 5)");

        // => OK (constructor & instance method)
        venice.eval(
            "(doto (. :java.util.ArrayList :new) \n" +
            "    (. :add 1) \n" +
            "    (. :add 2)) ");

        // => FAIL (invoking non whitelisted static method)
        venice.eval("( . :java.lang.System :exit 0)");

        // => FAIL (invoking rejected Venice I/O function)
        venice.eval("(io/slurp \"/tmp/file\)");
    }
}

```

```
// => FAIL (accessing non whitelisted system property)
venice.eval("(system-prop \"db.password\")");
}
}
```

## 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 '("+", "-", "*", "/") }

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 <a href="#">Venice markdown</a> 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

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

- item 1
- item 2  
next line  
next line
- 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

## Tables

A simple table

```
| JAN | 1 |
| FEB | 20 |
| MAR | 300 |
```

#### Column alignment

```
| :--- | :---: | ---: |
| 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  |
```

Line breaks in cells

```
| 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

[top](#)

### #{} ---

Creates a set.

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

[top](#)

### () ---

Creates a list.

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

[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

[top](#)

## \*ansi-term\*

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

\*ansi-term\*

```
=> false
```

[top](#)

## \*loaded-files\*

The loaded files

\*loaded-files\*

```
=> #{} 
```

[top](#)

## \*loaded-modules\*

The loaded modules

\*loaded-modules\*

```
=> #{:crypt :csv :xchart :trace :java :xml :semver :json :cidr :app :geoip :hexdump :io :maven :ansi :benchmark :str :gradle :excel :core :regex :component :pdf :parsifal :shell :math :time :config :kira}
```

[top](#)

## `*newline*`

The system newline

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

[top](#)

## `*ns*`

The current namespace

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

[top](#)

## `*run-mode*`

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

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

[top](#)

## `*version*`

The Venice version

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

[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

[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

[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. If ...

[as->](#)

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

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))  
=> 3
```

```
(do  
  (def person  
    {:name "Peter Meier"  
     :address {:street "Lindenstrasse 45"  
               :city "Bern"  
               :zip 3000}})  
  
  (-> person :address :street))  
=> "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. If ...

[-<>](#)

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 ...

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)
```

## 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 ...

[top](#)

•

```
(. 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)
=> 2022-06-01T09:21:25.408+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 a Java class. 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.

### [as-runnable](#)

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

### [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>)

[top](#)

::

`(.: 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.

[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

[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
```

[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
```

[top](#)

**=**

```
(= x y)
```

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
```

```
=> false
```

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

## SEE ALSO

[==](#)

Returns true if both operands have equivalent value.

[top](#)

[==](#)

```
(== x y)
```

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
```

## SEE ALSO

[=](#)

Returns true if both operands have equivalent type and value

[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
```

[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
```

[top](#)

## Object

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

Definition:

```
(defprotocol Object
  (toString [this] (. this :toString))
  (compareTo [this other] (. this :compareTo 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.

top

## [\[\]](#)

Creates a vector.

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

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

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) "The quick brown fox"))
    (accept-either (promise (fn [] (sleep 100) "jumps over the lazy dog"))
                  (fn [v] (println (pr-str v)))))
    (deref))
"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-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 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.

[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, 3, 4, 5, 0, 0, 0, 0, 0, 0]
```

[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.

[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.

[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

[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.

## 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.

## 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
```

## alength

(alength array)

Returns the length of an array

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

## 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, ...

[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]"
```

[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.

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"))
```

top

## ansi/bg-color

(bg-color code)

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

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

top

## ansi/fg-color

(fg-color code)

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

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

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 :progress)
  (sleep 1 :seconds)
  (pb 50 :progress)
  (sleep 1 :seconds)
  (pb 100 :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:"))
```

[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      25
        :show-percent true)]
(pb 0 :progress)
(sleep 1 :seconds)
(pb 50 :progress)
(sleep 1 :seconds)
(pb 100 :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))
```

[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))
```

[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)))
```

[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)))
```

[top](#)

## ansi/without-cursor

(without-cursor & forms)

Runs the given forms with the cursor turned off.

[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, ...

[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
```

[top](#)

## 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
```

[top](#)

## 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.

E.g.:

```
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-1.10.14.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
```

[top](#)

## app/manifest

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

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

[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
```

[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) "The quick brown fox"))  
    (apply-to-either (promise (fn [] (sleep 100) "jumps over the lazy dog"))  
                     (fn [v] (str/upper-case v))))  
    (deref))  
=> "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`

### [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 the current stage's result ...

### [accept-either](#)

Returns a new promise that, when either this or the other given promise completess 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.

[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. If ...

--<>

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.

top

## as-biconsumer

(as-biconsumer f)

Wraps the function f in a [java.util.function.BiConsumer](#)

### SEE ALSO

[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>)

[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>)

[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>)

top

## as-bifunction

(as-bifunction f)

Wraps the function f in a [java.util.function.BiFunction](#)

### SEE ALSO

[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>)

[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>)

[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>)

top

## as-binaryoperator

(as-binaryoperator f)

Wraps the function f in a [java.util.function.BinaryOperator](#)

### SEE ALSO

[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>)

### [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>)

### [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>)

[top](#)

## as-bipredicate

`(as-bipredicate f)`

Wraps the function `f` in a `java.util.function.BiPredicate`

### SEE ALSO

#### [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>)

#### [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>)

#### [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>)

[top](#)

## as-callable

`(as-callable f)`

Wraps the function `f` in a `java.util.concurrent.Callable`

### SEE ALSO

#### [as-runnable](#)

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

#### [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>)

#### [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>)

#### [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>)

#### [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>)

[top](#)

## as-consumer

`(as-consumer f)`

Wraps the function `f` in a [java.util.function.Consumer](#)

#### SEE ALSO

##### [as-runnable](#)

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

##### [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>)

##### [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>)

##### [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>)

##### [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>)

[top](#)

## as-function

`(as-function f)`

Wraps the function `f` in a [java.util.function.Function](#)

#### SEE ALSO

##### [as-runnable](#)

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

##### [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>)

##### [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>)

##### [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>)

##### [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>)

[top](#)

## as-predicate

`(as-predicate f)`

Wraps the function `f` in a [java.util.function.Predicate](#)

#### SEE ALSO

##### [as-runnable](#)

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

##### [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>)

### [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>)

### [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>)

### [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>)

[top](#)

## as-runnable

`(as-runnable f)`

Wraps the function `f` in a [java.lang.Runnable](#)

### SEE ALSO

#### [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>)

#### [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>)

#### [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>)

#### [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>)

#### [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>)

[top](#)

## as-supplier

`(as-supplier f)`

Wraps the function `f` in a [java.util.function.Supplier](#)

### SEE ALSO

#### [as-runnable](#)

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

#### [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>)

#### [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>)

#### [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>)

#### [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>)



## 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]
```

## assert

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

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

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

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

## 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 <= (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 return the new value.

[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 map, where k is a key and f is a function that will take the old value return the new value.

[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}]
```

[top](#)

## asub

```
(asub array start len)
```

Returns a sub array

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

[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.

[top](#)

## atom?

```
(atom? x)
```

Returns true if x is an atom, otherwise false

```
(do
  (def counter (atom 0))
  (atom? counter))
=> true
```

[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 ...

[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.

[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.

[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.

[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)
```

[top](#)

## benchmark/benchmark

```
(benchmark expr warmup-iterations iterations & options)
```

Benchmarks the given expression.

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  
:outliers b    show the outlier range in the chart {true/false}, defaults to false

```
(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))
```

[top](#)

## bigint

```
(bigint x)
```

Converts to big integer.

```
(bigint 2000)
=> 2000N
```

```
(bigint 34897.65)
=> 34897N
```

```
(bigint "5676000000000")
=> 5676000000000N
```

```
(bigint nil)
=> 0N
```

[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.

[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
```

[top](#)

## boolean?

```
(boolean? n)
```

Returns true if n is a boolean

```
(boolean? true)
```

```
=> true
```

```
(boolean? false)
```

```
=> true
```

```
(boolean? nil)
```

```
=> false
```

```
(boolean? 0)
```

```
=> false
```

[top](#)

## 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

[top](#)

## 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.

[top](#)

## bytebuf

```
(bytebuf x)
```

Converts x to bytebuf. x can be a bytebuf, a list/vector of longs, or a string

```
(bytebuf [0 1 2])  
=> [0 1 2]
```

```
(bytebuf '(0 1 2))  
=> [0 1 2]
```

```
(bytebuf "abc")  
=> [97 98 99]
```

[top](#)

## bytebuf-allocate

```
(bytebuf-allocate length)
```

Allocates a new bytebuf. The values will be all zero.

```
(bytebuf-allocate 20)  
=> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

[top](#)

## bytebuf-capacity

```
(bytebuf-capacity buf)
```

Returns the capacity of a bytebuf.

```
(bytebuf-capacity (bytebuf-allocate 100))  
=> 100
```

[top](#)

## bytebuf-from-string

```
(bytebuf-from-string s encoding)
```

Converts a string to a bytebuf using an optional encoding. The encoding defaults to :UTF-8

```
(bytebuf-from-string "abcdef" :UTF-8)  
=> [97 98 99 100 101 102]
```

[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
```

[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
```

[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
```

[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! 1I)
    (bytebuf-put-int! 2I)
    (bytebuf-get-int 0))
=> 1I
```

[top](#)

## 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
```

[top](#)

## bytebuf-limit

```
(bytebuf-limit buf)
```

Returns the limit of a bytebuf.

```
(bytebuf-limit (bytebuf-allocate 100))
=> 100
```

[top](#)

## bytebuf-pos

```
(bytebuf-pos buf)
```

Returns the buffer's current position.

```
(bytebuf-pos (bytebuf-allocate 10))
=> 0
```

[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]
```

[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]
```

[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]
```

[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]
```

[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]
```

[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]
```

[top](#)

## bytebuf-put-long!

```
(bytebuf-put-long! buf l)
```

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 4 0 0 0 0 0 0 0 8]
```

[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]
```

[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)
```

[top](#)

## bytebuf-to-string

```
(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"
```

[top](#)

## bytebuf?

```
(bytebuf? x)
```

Returns true if x is a bytebuf

```
(bytebuf? (bytebuf [1 2]))  
=> true
```

```
(bytebuf? [1 2])  
=> false
```

```
(bytebuf? nil)  
=> false
```

[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 25 :col 18} {:fn-name "user/f4" :file "example" :line 24 :col 18}
{:fn-name "user/f3" :file "example" :line 23 :col 18} {:fn-name "user/f2" :file "example" :line 22 :col 18}
{:fn-name "user/f1" :file "example" :line 26 :col 5}]
```

[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

[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

[top](#)

## cartesian-product

```
(cartesian-product coll1 coll2 coll*)
```

Returns the cartesian product of two or more collections.

Removes all duplicates 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

[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.

[top](#)

## cast

```
(cast class object)
```

Casts a Java object

```
(do
  (import :java.awt.image.BufferedImage)
  (import :java.awt.Graphics)

  ;; cast the graphics context to 'java.awt.Graphics' instead of the
  ;; implicit cast to 'java.awt.Graphics2D' as Venice is doing
  (let [img (. :BufferedImage :new 40 40 1)
        gd (cast :Graphics (. img :createGraphics))]
    (. gd :fillOval 10 20 5 5)
    img))
=> BufferedImage@60bd273d: type = 1 DirectColorModel: rmask=ff0000 gmask=ff00 bmask=ff amask=0
IntegerInterleavedRaster: width = 40 height = 40 #Bands = 3 xOff = 0 yOff = 0 dataOffset[0] 0
```

[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

[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.

[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  | #\          |
| #\rparen             | \u0029  | #\)         |
| #\quote              | \u0022  | #\"         |
| #\backslash          | \u005C  | #\backslash |
| #\pildown            | \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.

[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.

[top](#)

## charset-default-encoding

```
(charset-default-encoding)
```

Returns the default charset of this Java virtual machine.

```
(charset-default-encoding)  
=> :UTF-8
```

[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
```

## cidr/in-range?

```
(cidr/in-range? ip cidr)
```

Returns true if the ip adress 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 CIDR Java object obtained from 'cidr/parse'.

```
(cidr/in-range? "222.220.0.0" "222.220.0.0/11")  
=> true
```

```
(cidr/in-range? (cidr/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
```

## cidr/inet-addr

```
(cidr/inet-addr addr)
```

Converts a stringified IPv4 or IPv6 to a Java InetAddress.

```
(cidr/inet-addr "222.192.0.0")  
=> /222.192.0.0
```

```
(cidr/inet-addr "2001:0db8:85a3:08d3:1319:8a2e:0370:7347")  
=> /2001:db8:85a3:8d3:1319:8a2e:370:7347
```

## cidr/inet-addr-from-bytes

```
(cidr/inet-addr-bytes addr)
```

Converts a IPv4 or IPv6 byte address (a vector of unsigned integers) to a Java InetAddress.

```
(cidr/inet-addr-from-bytes [222I 192I 12I 0I])  
=> /222.192.12.0
```

```
(cidr/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
```

## cidr/inet-addr-to-bytes

```
(cidr/inet-addr-to-bytes addr)
```

Converts a stringified IPv4/IPv6 address or a Java InetAddress to an InetAddress byte vector.

```
(cidr/inet-addr-to-bytes "222.192.12.0")
=> [222I 192I 12I 0I]

(cidr/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]

(cidr/inet-addr-to-bytes (cidr/inet-addr "222.192.0.0"))
=> [222I 192I 0I 0I]
```

[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"
```

[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"
```

[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]
```

[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]
```

[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
```

[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
```

[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"
```

[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.

[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.

[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.

[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.

[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.

[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.

[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
```

[top](#)

## coll?

```
(coll? coll)
```

Returns true if coll is a collection

```
(coll? {:a 1})  
=> true  
  
(coll? [1 2])  
=> true
```

[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.

top

## comment

(comment & body)

Ignores body, yields nil

```
(comment
  (println 1)
  (println 5))
=> nil
```

top

## comp

(comp f\*)

Takes a set of functions and returns a fn that is the composition of those fns. The returned fn takes a variable number of args, applies the rightmost of fns to the args, the next fn (right-to-left) to the result, etc.

```
((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
```

## 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
```

## 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.

[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?)
=> function anonymous-64792091-cd3b-4d94-a3f9-376abddf5f38 {visibility :public, ns "", native false} defined at
core: line 1469, col 10

(filter (complement even?) '(1 2 3 4))
=> (1 3)
```

[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 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.

[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.

[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'.

[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'.

[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.

[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 ...

[top](#)

## 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 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 succesfully 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 succesfully 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 ...

[top](#)

## concat

```

(concat coll)
(concat coll & colls)

```

Returns a collection of the concatenation of the elements in the supplied colls.

```
(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)
```

[top](#)

## 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

[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))
```

```
(cond-> 1      ; we start with 1
  true inc    ; the condition is true so (inc 1) => 2
  false (* 42) ; the condition is false so the operation is skipped
  (= 2 2) (* 3)) ; (= 2 2) is true so (* 2 3) => 6
=> 6
```

## 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 ...

[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.

```
(cond->> 1      ; we start with 1
  true inc    ; the condition is true so (inc 1) => 2
  false (* 42) ; the condition is false so the operation is skipped
  (= 2 2) (* 3)) ; (= 2 2) is true so (* 3 2) => 6
=> 6
```

## 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 ...

[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

[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 an 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 passed as prefix to get env vars.

### [config/properties](#)

Reads configuration part from system properties, filtered by a prefix. nil may passed as prefix to get property vars.

[top](#)

## config/env

(env prefix)

Reads configuration part from environment variables, filtered by a prefix. `nil` may 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 passed as prefix to get property vars.

[config/build](#)

Merges given configuration parts and returns it as a map.

[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/temurin-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 an 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 passed as prefix to get env vars.

### [config/build](#)

Merges given configuration parts and returns it as a map.

[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.

[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.

[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 "Temurin"}}
```

```
(config/property-var "java.version" [:java :version])  
=> {:java {:version "1.8.0_322"}}
```

```
(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 passed as prefix to get property vars.

[config/build](#)

Merges given configuration parts and returns it as a map.

[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.

[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). For list, vectors and ordered maps the values are added at the end. For all other sets and maps 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.

### [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.

[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)`. For mutable list the values are added at the end. For all mutable sets and maps 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!)
=> ()

(conj! 4)
=> 4
```

## cons

```
(cons x coll)
```

Returns a new collection where x is the first element and coll is the rest

```
(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). For list, vectors and ordered maps the values are ...

#### 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.

## cons!

```
(cons! x coll)
```

Adds x to the mutable coll

```
(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}
```

[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.

[top](#)

## contains?

```
(contains? coll key)
```

Returns true if key is present in the given collection, otherwise returns false.

```
(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
```

[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
```

[top](#)

## cpus

```
(cpus)
```

Returns the number of available processors or number of hyperthreads if the CPU supports hyperthreads.

```
(cpus)
=> 8
```

[top](#)

## crypt/decrypt

```
(crypt/decrypt algorithm passphrase & options)
```

Returns a new thread safe function to decrypt a string or a bytebuf given the algorithm and passphrase. If a string is passed it is base64 decoded, decrypted, and returned as string. If a bytebuf is passed the decrypted bytebuf is returned.

Supported algorithms: "DES", "3DES", "AES256"

Options:

`:url-safe {true/false}`

The boolean option directs the base64 decoder to decode standard or URL safe base64 encoded strings. If enabled (true) the base64 decoder will convert '-' and '\_' characters back to '+' and '/' before decoding.

Defaults to false.

```
(do
  (load-module :crypt)
  (def decrypt (crypt/decrypt "3DES" "secret" :url-safe true))
  (decrypt "ndmW1NLsDHA") ; => "hello"
  (decrypt "KPYjndkZ8vM") ; => "world"
  (decrypt (bytebuf [128 216 205 163 62 43 52 82]))) ; => [1 2 3 4]
=> [1 2 3 4]
```

top

## crypt/encrypt

(crypt/encrypt algorithm passphrase & options)

Returns a new thread safe function to encrypt a string or a bytebuf given the algorithm and passphrase. If a string is passed it is encrypted and returned as a base64 encoded string. If a bytebuf is passed the encrypted bytebuf is returned.

Supported algorithms: "DES", "3DES", "AES256"

Options:

:url-safe {true/false}

The boolean option directs the base64 encoder to emit standard or URL safe base64 encoded strings. If `true` the base64 encoder will emit '-' and '\_' instead of the usual '+' and '/' characters.

Defaults to false.

Note: no padding is added when encoding using the URL-safe alphabet.

```
(do
  (load-module :crypt)
  (def encrypt (crypt/encrypt "3DES" "secret" :url-safe true))
  (encrypt "hello") ; => "ndmW1NLsDHA"
  (encrypt "world") ; => "KPYjndkZ8vM"
  (encrypt (bytebuf [1 2 3 4]))) ; => [128 216 205 163 62 43 52 82]
=> [128 216 205 163 62 43 52 82]
```

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 instead!

```
(> (crypt/md5-hash "hello world")
  (str/bytebuf-to-hex :upper))
=> "5EB63BBE01EEED093CB22BB8F5ACDC3"

(> (crypt/md5-hash "hello world" "-salt-")
  (str/bytebuf-to-hex :upper))
=> "C40C4EAC3C1B87B6877E21FEB087D0A"
```

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.

```
(-> (crypt/pbkdf2-hash "hello world" "-salt-")
    (str/bytebuf-to-hex :upper))
=> "54F2B4411E8817C2A0743B2A7DD7EAE5AA3F748D1DDDC00766380914AFFE995"

(-> (crypt/pbkdf2-hash "hello world" "-salt-" 1000 256)
    (str/bytebuf-to-hex :upper))
=> "54F2B4411E8817C2A0743B2A7DD7EAE5AA3F748D1DDDC00766380914AFFE995"
```

[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.

```
(-> (crypt/sha1-hash "hello world")
    (str/bytebuf-to-hex :upper))
=> "2AAE6C35C94FCFB415DBE95F408B9CE91EE846ED"

(-> (crypt/sha1-hash "hello world" "-salt-")
    (str/bytebuf-to-hex :upper))
=> "90AECEDB9423CC9BC5BB7CBAFB88380BE5745B3D"
```

[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.

```
(let [s (-> (crypt/sha512-hash "hello world")
            (str/bytebuf-to-hex :upper))]
      (str (str/nfirst s 32) "... " (str/nlast s 32)))
=> "309ECC489C12D6EB4CC40F50C902F2B4...D830E81F605DCF7DC5542E93AE9CD76F"

(let [s (-> (crypt/sha512-hash "hello world" "-salt-")
            (str/bytebuf-to-hex :upper))]
      (str (str/nfirst s 32) "... " (str/nlast s 32)))
=> "316EBB70239D9480E91089D5D5BC6428...03095F186B19FC33C93D60282F3314A2"
```

[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.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `java.net.URI`

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"))
```

[top](#)

## csv/write

(csv/write writer records & options)

Writes data to a writer in CSV format. The writer is a Java `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

```
(let [file (io/file "test.csv")
      fs (. :java.io.FileOutputStream :new file)]
  (try-with [writer (. :java.io.OutputStreamWriter :new fs "utf-8")]
    (csv/write writer [[1 "AC" false] [2 "WS" true]])))
```

[top](#)

## csv/write-str

(csv/write-str records & options)

Writes data to a string in CSV format.

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"
```

[top](#)

## current-time-millis

(current-time-millis)

Returns the current time in milliseconds.

```
(current-time-millis)  
=> 1654068120834
```

### SEE ALSO

[nano-time](#)

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

[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.

[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)

[dag/topological-sort](#)

Topological sort of a DAG using Kahn's algorithm ([https://en.wikipedia.org/wiki/Topological\\_sorting](https://en.wikipedia.org/wiki/Topological_sorting))

[top](#)

## 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)

[dag/topological-sort](#)

Topological sort of a DAG using Kahn's algorithm ([https://en.wikipedia.org/wiki/Topological\\_sorting](https://en.wikipedia.org/wiki/Topological_sorting))

[top](#)

## 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)

[dag/children](#)

Returns the transitive child nodes

[dag/parent-of?](#)

Returns true if `p` is a transitive parent of `v`

[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)

[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

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)

#### dag/topological-sort

Topological sort of a DAG using Kahn's algorithm ([https://en.wikipedia.org/wiki/Topological\\_sorting](https://en.wikipedia.org/wiki/Topological_sorting))

top

## dag/dag

(dag)  
(dag edges\*)

Creates a new DAG (directed acyclic graph)

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](https://en.wikipedia.org/wiki/Topological_sorting))

### [dag/edges](#)

Returns the edges of a DAG

### [dag/nodes](#)

Returns the nodes of a DAG

### [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

[top](#)

## [dag/dag?](#)

```
(dag? coll)
```

Returns true if coll is a DAG

```
(dag/dag? (dag/dag))  
=> true
```

[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)

#### [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

[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)

[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

[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)

[dag/add-edges](#)

Add edges to a DAG. Returns a new DAG with added edges.

[dag/nodes](#)

Returns the nodes of a DAG

[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)

[dag/nodes](#)

Returns the nodes of a DAG

[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)

[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

[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)

#### [dag/parents](#)

Returns the transitive parent nodes

#### [dag/child-of?](#)

Returns true if c is a transitive child of v

[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)

[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

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", "C"] ; C G  
      ["F", "G"] ; \ /  
      ["G", "D"] ; D  
      (dag/roots))  
=> ("A" "E")
```

### SEE ALSO

[dag/dag](#)

Creates a new DAG (directed acyclic graph)

[dag/parents](#)

Returns the transitive parent nodes

[dag/children](#)

Returns the transitive child nodes

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)

[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.

[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

[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

[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

[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

[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

[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

[top](#)

## 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 2.5787M 3 :HALF_UP)
=> 2.579M
```

```
(decimal "2.5787" 3 :HALF_UP)
=> 2.579M
```

```
(decimal nil)
=> 0M
```

[top](#)

## decimal?

```
(decimal? n)
```

Returns true if n is a decimal

```
(decimal? 4.0M)  
=> true
```

```
(decimal? 4.0)  
=> false
```

```
(decimal? 3)  
=> false
```

```
(decimal? 3I)  
=> false
```

[top](#)

## 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.

[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.

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.

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.

[top](#)

# defmacro

```
(defmacro name [params*] body)
```

Macro definition

```
(defmacro unless [pred a b]
  `(if (not ~pred) ~a ~b))
=> macro user/unless {visibility :public, ns "user", native false} defined at example: line 21, col 21
```

## SEE ALSO

### macroexpand

If form represents a macro form, returns its expansion, else returns form.

### macroexpand-all

Recursively expands all macros in the form.

[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.

[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.

[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.

[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.

[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.

[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.

[top](#)

## deftype

```
(deftype name fields)
(deftype name fields validator)
```

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

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.

```
(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)

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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
```

[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
```

[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). For list, vectors and ordered maps the values are ...

### [disj](#)

Returns a new set with the x, xs removed.

[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 11111111111111111111111111111111N)
=> 32
```

[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}
```

[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 return the new value.

[top](#)

## 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 map, where k is a key and f is a function that will take the old value return the new value.

[top](#)

## 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"}]
```

[top](#)

## 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.

[top](#)

## do

```
(do exprs)
```

Evaluates the expressions in order and returns the value of the last.

```
(do (println "Test...") (+ 1 1))
Test...
=> 2
```

[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.

```
(->> (lazy-seq #(rand-long 100))
      (take 4)
      (doall))
=> (20 80 56 47)

(->> (lazy-seq #(rand-long 100))
      (doall 4))
=> (92 15 77 25)
```

## SEE ALSO

[lazy-seq](#)

Creates a new lazy sequence.

[top](#)

## dobench

```
(dobench count expr)
```

Runs the expr count times in the most effective way and returns a list of elapsed nanoseconds for each invocation. It's main purpose is supporting benchmark test.

```
(dobench 10 (+ 1 1))
=> (2997 816 463 247 259 228 260 240 244 225)
```

[top](#)

## 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 similiar var with the **Levenshtein distance** 1.

E.g:

```
> (doc dac)
Symbol 'dac' not found!

Did you mean?
dag/dag
dec
```

```
(doc +)
```

```
(doc def)
```

```
(do
  (deftype :complex [real :long, imaginary :long])
  (doc :complex))
```

top

## docoll

```
(docoll f coll)
```

Applies f to the items of the collection presumably for side effects. Returns nil.

```
(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
```

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

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:*

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
```

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
```

## SEE ALSO

### [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.

[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 ...

[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}
```

[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 1.2M)
=> 1.2

(double "1.2")
=> 1.2
```

[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

```
(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]
```

## double?

```
(double? n)
```

Returns true if n is a double

```
(double? 4.0)  
=> true
```

```
(double? 3)  
=> false
```

```
(double? 3I)  
=> false
```

```
(double? 3.0M)  
=> false
```

```
(double? true)  
=> false
```

```
(double? nil)  
=> false
```

```
(double? {})  
=> false
```

## 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])  
=> []
```

## 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])
=> []
```

[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]
```

[top](#)

## empty

```
(empty coll)
```

Returns an empty collection of the same category as coll, or nil

```
(empty {:a 1})
=> {}

(empty [1 2])
=> []

(empty '(1 2))
=> ()
```

[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
```

[top](#)

## empty?

```
(empty? x)
```

Returns true if x is empty. Accepts strings, collections and bytebufs.

```
(empty? {})  
=> true
```

```
(empty? [])  
=> true
```

```
(empty? '())  
=> true
```

```
(empty? "")  
=> true
```

[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

[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
```

```
(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.

[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? (int 3))  
=> false
```

## SEE ALSO

### [odd?](#)

Returns true if n is odd, throws an exception if n is not an integer

## 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
```

## every?

```
(every? pred coll)
```

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

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

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

```
(every? number? [1 2 3 4])
=> true
```

```
(every? number? [1 2 3 :a])
=> false
```

```
(every? #(>= % 10) [10 11 12])
=> true
```

## 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`

[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

[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.

[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

[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

[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 21, 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 21 :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.

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

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

top

## excel/add-column

```
(add-column sheet-builder title)
(add-column sheet-builder title options)
```

Defines a column with optional attributes.

Options:

|                     |                                                                                           |
|---------------------|-------------------------------------------------------------------------------------------|
| :field f            | a field, e.g. :first-name                                                                 |
| :width n            | width in points, e.g. 100                                                                 |
| :hidden b           | hidden, 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/writer :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.

[top](#)

## excel/add-font

```
(add-font writer font-id)
(add-font writer 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  
: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/writer :xlsx)]
    (excel/add-font wbook :header { :height 12
                                     :bold true
                                     :italic 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-sheet](#)

Adds a sheet with optional attributes to an Excel.

[excel/add-style](#)

Add a style with optional attributes to an Excel.

top

## excel/add-sheet

```
(add-sheet writer title)
(add-sheet writer 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/writer :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/writer :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.

### [excel/add-font](#)

Add font with optional attributes to an Excel.

### [excel/add-style](#)

Add a style with optional attributes to an Excel.

[top](#)

## excel/add-style

```

(add-style writer style-id)
(add-style writer 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"<br>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/writer :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-sheet](#)

Adds a sheet with optional attributes to an Excel.

[excel/add-font](#)

Add font with optional attributes to an Excel.

[top](#)

## excel/auto-size-column

```
(auto-size-column builder col)
```

Auto size the width of column col (1..n).

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/writer :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.

[excel/write-items](#)

Writes the passed data items to the sheet

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value 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).

[top](#)

## excel/auto-size-columns

```
(auto-size-columns builder)
```

Auto size the width of all columns.

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/writer :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).

[excel/write-items](#)

Writes the passed data items to the sheet

[excel/write-item](#)

Render a single data item to the sheet

[excel/write-value](#)

Writes a value 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).

[top](#)

## excel/cell-address

(cell-address builder row col)

Returns the cell address for a cell at row/col in a sheet

```
(do
  (load-module :excel)
  (let [data [ { :a 100 :b 200 }
                { :a 101 :b 201 }
                { :a 102 :b 202 } ]
        wbook (excel/writer :xlsx)
        sheet (excel/add-sheet wbook "Sheet 1"
                                { :no-header-row true })
        addr #(excel/cell-address sheet %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/cell-formula](#)

Set a formula for a specific cell given by its row and col.

[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/writer :xlsx)]
      (excel/write-data wbook "Data" [[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-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.LocalDateTime).

[top](#)

## excel/cell-formula

(cell-formula builder 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/writer :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/writer :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/cell-address](#)

Returns the cell address for a cell at row/col in a sheet

### [excel/sum-formula](#)

Returns a sum formula

### [excel/write-items](#)

Writes the passed data items to the sheet

### [excel/write-item](#)

Render a single data item to the sheet

### [excel/write-value](#)

Writes a value to a specific cell given by its row and col.

### [excel/auto-size-columns](#)

Auto size the width of all columns.

### [excel/auto-size-column](#)

Auto size the width of column col (1..n).

### [excel/row-height](#)

Set the height of a row (1..n).

[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 }

```

(do
  (load-module :excel)

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)]
      (excel/write-data wbook "Data" [[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-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.LocalDateTime).

[top](#)

## excel/convert->reader

```
(convert->reader builder)
```

Converts an excel or sheet builder to the corresponding reader.

```
(do
  (load-module :excel)
  (let [data [ {:a 100 :b 200 }
                {:a 101 :b 201 }
                {:a 102 :b 202 } ]
        wbook (excel/writer :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)")
    (let [reader (excel/convert->reader sheet)]
      (excel/evaluate-formulas reader)
      (excel/read-long-val reader 1 3))))
```

[top](#)

## excel/evaluate-formulas



```
(evaluate-formulas it)
```

Evaluate all formulas in the Excel.

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

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)]
      (excel/write-data wbook "Data" [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))]
    (excel/evaluate-formulas wbook)))
```

## SEE ALSO

[excel/writer](#)

Creates a new Excel builder for the given type :xls or :xlsx.

[top](#)

## excel/open

```
(open source)
```

Opens an Excel from a source and returns an Excel reader.

Supported sources are string file path, bytebuf, `:java.io.File`, or `:java.io.InputStream`.

```
(do
  (load-module :excel)
  (excel/open "sample.xls"))
```

## SEE ALSO

[excel/sheet-count](#)

Returns the number of sheets in the Excel.

[excel/sheet](#)

Returns a sheet from the Excel reader referenced by its name or sheet index.

[excel/evaluate-formulas](#)

Evaluate all formulas in the Excel.

[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/writer :xlsx)]
    (excel/write-data wbook "Data" [[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.LocalDateTime).

[top](#)

## excel/read-date-val

```
(read-date-val sheet row col)
```

Returns the sheet cell value as a date (:java.time.LocalDateTime).

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

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)
          dt    (time/local-date 2021 1 1)
          ts    (time/local-date-time 2021 1 1 15 30 45)]
      (excel/write-data wbook "Data" [[100 dt ts 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.

[top](#)

## 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/writer :xlsx)]
      (excel/write-data wbook "Data" [[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.LocalDateTime).

[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/writer :xlsx)]
    (excel/write-data wbook "Data" [[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/writer :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.LocalDateTime).

[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/writer :xlsx)]
      (excel/write-data wbook "Data" [[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.LocalDateTime).

[top](#)

## excel/row-height

```
(row-height builder row height)
```

Set the height of a row (1..n).

```
(do  
  (load-module :excel)  
  (let [os (. :java.io.FileOutputStream :new "sample.xlsx")  
        data [ {:first "John" :last "Doe" :age 28 }  
                 {:first "Sue" :last "Ford" :age 26 } ]  
        wbook (excel/writer :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 2 100)  
    (excel/write->stream wbook os)))
```

## SEE ALSO

### [excel/auto-size-columns](#)

Auto size the width of all columns.

### [excel/write-items](#)

Writes the passed data items to the sheet

### [excel/write-item](#)

Render a single data item to the sheet

### [excel/write-value](#)

Writes a value 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).

[top](#)

## excel/sheet

(sheet wbook ref)

Returns a sheet from the Excel reader referenced by its name or sheet index.

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

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)]
      (excel/write-data wbook "Data1" [[100 101 102] [200 201 202]])
      (excel/write-data wbook "Data2" [[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 the Excel.

#### [excel/sheet-name](#)

Returns a sheet from the Excel reader referenced by its name or sheet index.

#### [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.LocalDateTime`).

[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)

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)]
      (excel/write-data wbook "Data" [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (excel/sheet-col-range sheet 1)))
```

### 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.

[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/writer :xlsx)]
      (excel/write-data wbook "Data" [[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 reader referenced by its name or sheet index.

[excel/evaluate-formulas](#)

Evaluate all formulas in the Excel.

[top](#)

## excel/sheet-name

(sheet-name sheet)

Returns a sheet from the Excel reader referenced by its name or sheet index.

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

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)]
      (excel/write-data wbook "Data" [[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)))
```

[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)

  (defn test-xls []
    (let [wbook (excel/writer :xlsx)]
      (excel/write-data wbook "Data" [[100 101 102] [200 201 202]])
      (excel/write->bytebuf wbook)))

  (let [wbook (excel/open (test-xls))
        sheet (excel/sheet wbook "Data")]
    (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 ...

[top](#)

## excel/sum-formula

(sum-formula builder row-from row-to col-from col-to)

Returns a sum formula

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



```
(let [data [ {:a 100 :b 200 }
              {:a 101 :b 201 }
              {:a 102 :b 202 } ]
      wbook (excel/writer :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/cell-address](#)

Returns the cell address for a cell at row/col in a sheet

top

## excel/write->bytebuf

```
(write->bytebuf builder os)
```

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/writer :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.

top

## excel/write->file

```
(write->file builder 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/writer :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.

[top](#)

## excel/write->stream

(write->stream builder os)

Writes the excel to a Java :OutputStream.

```
(do
  (load-module :excel)
  (let [os (. :java.io.FileOutputStream :new "sample.xlsx")
        data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/writer :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.

[top](#)

## excel/write-data

(write-data builder sheet-name data)

Writes the data of a 2D array to an excel sheet. Creates a new sheet with the name given by 'sheet-name'.

```
(do
  (load-module :excel)
  (let [wbook (excel/writer :xlsx)
        dt    (time/local-date 2021 1 1)
        ts    (time/local-date-time 2021 1 1 15 30 45)]
    (excel/write-data wbook "Data" [[100 101 102 103 104 105]
                                     [200 "ab" 1.23 dt ts false]])
    (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.

[top](#)

## excel/write-item

(write-item builder item)

Render a single data item to the sheet

```
(do
  (load-module :excel)
  (let [wbook (excel/writer :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 to the sheet

[excel/write-value](#)

Writes a value 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.

[excel/auto-size-column](#)

Auto size the width of column col (1..n).

[excel/row-height](#)

Set the height of a row (1..n).

[top](#)

## excel/write-items

(write-items builder items)

Writes the passed data items to the sheet

```
(do
  (load-module :excel)
  (let [data [ {:first "John" :last "Doe" :age 28 }
                {:first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/writer :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 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.

[excel/auto-size-column](#)

Auto size the width of column col (1..n).

[excel/row-height](#)

Set the height of a row (1..n).

[top](#)

## excel/write-value

(write-value builder row col val)

Writes a value to a specific cell given by its row and col.

```
(do
  (load-module :excel)
  (let [wbook (excel/writer :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-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/writer :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/add-column sheet "First Name" { :field :first })
    (excel/add-column sheet "Last Name" { :field :last })
    (excel/add-column sheet "Age" { :field :age })
    (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-items](#)

Writes the passed data items 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.

### [excel/auto-size-column](#)

Auto size the width of column col (1..n).

### [excel/row-height](#)

Set the height of a row (1..n).

[top](#)

## excel/writer

(writer type)

Creates a new Excel builder for the given type :xls or :xlsx.

```

(do
  (load-module :excel)
  (let [data [ { :first "John" :last "Doe" :age 28 }
               { :first "Sue" :last "Ford" :age 26 } ]
        wbook (excel/writer :xls)
        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.xls")))
```

## 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.

[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 the Excel.

[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
```

[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.

## 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.

## 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.

[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.

[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 ...

[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}
```

[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}
```

[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
```

## 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
```

## 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.

## 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

```
(float-array '(1.0 2.0 3.0))
=> [1.0, 2.0, 3.0]

(float-array '(1I 2 3.2 3.56M))
=> [1.0, 2.0, 3.2000000047683716, 3.559999942779541]

(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 42.0)
=> [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0]
```

[top](#)

## 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

[top](#)

## flush

```
(flush)
(flush os)
```

Without arg flushes the output stream that is the current value of `*out*`. With arg flushes the passed output stream.  
Returns `nil`.

```
(flush)
=> nil

(flush *out*)
=> nil

(flush *err*)
=> nil
```

[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

;; 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.

[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)))))
```

[top](#)

## fn-name

```
(name x)
```

Returns the qualified name of a function or macro

```
(fn-name str/digit?)
=> "str/digit?"
```

### SEE ALSO

#### [name](#)

Returns the name String of a string, symbol, keyword, or function

#### [namespace](#)

Returns the namespace string of a symbol, keyword, or function.

[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)]
```

[top](#)

## fn?

```
(fn? x)
```

Returns true if `x` is a function

```
(do
  (def sum (fn [x] (+ 1 x)))
  (fn? sum))
=> true
```

[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.

```
((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
```

[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.

[top](#)

## formal-type

```
(formal-type object)
```

Returns the formal type of a Java object

```
(do  
  (import :java.awt.image.BufferedImage)  
  (import :java.awt.Graphics)  
  
  ;; cast the graphics context to 'java.awt.Graphics' instead of the  
  ;; implicit cast to 'java.awt.Graphics2D' as Venice is doing  
  (let [img (. :BufferedImage :new 40 40 1)  
        gd (cast :Graphics (. img :createGraphics))]  
    (formal-type gd)))  
=> :java.awt.Graphics
```

[top](#)

## format-micro-time

```
(format-micro-time time)
(format-micro-time time & options)
```

Formats a time given in microseconds as long or double.

Options: \n | :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.

[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.

[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.

[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
```

[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)
```

[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

(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.

[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 ...

[top](#)

## future?

```
(future? f)
```

Returns true if f is a Future otherwise false

```

(future? (future (fn [] 100)))
=> true

```

[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.

[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 23 :completed-task-count 23}
```

## 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 ...

[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 ...

[top](#)

## gc

```
(gc)
```

Run the Java garbage collector. Runs the finalization methods of any objects pending finalization prior to the GC.

```
(gc)
=> nil
```

[top](#)

## gensym

```
(gensym)
(gensym prefix)
```

Generates a symbol.

```
(gensym)
=> G__31314

(gensym "prefix_")
=> prefix_31350
```

[top](#)

## geoip/build-maxmind-city-db-url

```
(geoip/build-maxmind-city-db-url lic-key)
```

Build the URL for downloading the MaxMind city GEO IP database.

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 :geoup)
  (geoup/build-maxmind-city-db-url "YOUR-MAXMIND-LIC-KEY"))
=> "https://download.maxmind.com/app/geoup_download?edition_id=GeoLite2-City-CSV&license_key=YOUR-MAXMIND-LIC-KEY&suffix=zip"
```

#### SEE ALSO

[geoup/download-maxmind-db](#)

Downloads the MaxMind country or city GEO IP database. Returns the DB as bytearray. 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.

[top](#)

## geoup/build-maxmind-country-db-url

```
(geoup/build-maxmind-country-db-url lic-key)
```

Build the URL for the MaxMind country GEO IP database. The download requires a license key that is sent as part of the URL.

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 :geoup)
  (geoup/build-maxmind-country-db-url "YOUR-MAXMIND-LIC-KEY"))
=> "https://download.maxmind.com/app/geoup_download?edition_id=GeoLite2-Country-CSV&license_key=YOUR-MAXMIND-LIC-KEY&suffix=zip"
```

#### SEE ALSO

[geoup/download-maxmind-db](#)

Downloads the MaxMind country or city GEO IP database. Returns the DB as bytearray. 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.

[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

### [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 IP an address to its associated city and latitude/longitude location. The resolve function ...

### [geoiP/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves IP an address to its associated city and latitude/longitude location. The resolve function ...

[top](#)

## geoiP/download-google-country-db-to-csvfile

```
(geoiP/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 'geoiP/google-country-url'.

```
(do
  (load-module :geoiP)
  (geoiP/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).

[top](#)

## geoiP/download-maxmind-db

```
(geoiP/download-maxmind-db type 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 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 :country "YOUR-MAXMIND-LIC-KEY"))
```

## SEE ALSO

### [geoiP/build-maxmind-country-db-url](#)

Build the URL for the MaxMind country GEO IP database. The download requires a license key that is sent as part of the URL.

### [geoiP/build-maxmind-city-db-url](#)



Build the URL for downloading the MaxMind city GEO IP database.

top

## geoip/download-maxmind-db-to-zipfile

```
(geoip/download-maxmind-db-to-zipfile zipfile type 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-LIC-KEY"))
```

### SEE ALSO

[geoip/build-maxmind-country-db-url](#)

Build the URL for the MaxMind country GEO IP database. The download requires a license key that is sent as part of the URL.

[geoip/build-maxmind-city-db-url](#)

Build the URL for downloading the MaxMind city GEO IP database.

top

## geoip/ip-to-city-loc-resolver

```
(geoip/ip-to-city-loc-resolver geoip-zip)
```

Returns a resolve function that resolves IP an 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

### [geopip/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.

### [geopip/ip-to-country-resolver](#)

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information ...

### [geopip/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 ...

### [geopip/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves IP an address to its associated city and latitude/longitude location. The resolve function ...

### [geopip/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 ...

[top](#)

## geopip/ip-to-city-loc-resolver-mem-optimized

(geopip/ip-to-city-loc-resolver-mem-optimized geopip-zip)

Returns a resolve function that resolves IP an 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 geopip-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 'geopip/ip-to-city-loc-resolver' instead!

```
(do
  (def rv (geopip/ip-to-city-loc-resolver-mem-optimized "./geopip-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

### [geopip/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.

### [geopip/ip-to-country-resolver](#)

Returns a resolve function that resolves an IP addresses to its associated country. The resolve function returns the country information ...

### [geopip/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 IP an 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 ...

[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

#### [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-city-loc-resolver](#)

Returns a resolve function that resolves IP an address to its associated city and latitude/longitude location. The resolve function ...

#### [geoip/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves IP an 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 ...

[top](#)

## geoip/ip-to-country-resolver

(geoip/ip-to-country-resolver geoip-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 geoip-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 (geoip/ip-to-country-resolver "./geoip-country.zip"))
  (rv "91.223.55.1")) ;; => { :country-name "Poland"
                             ;;      :country-iso "PL" }
```

## 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-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 IP an address to its associated city and latitude/longitude location. The resolve function ...

[geoip/ip-to-city-loc-resolver-mem-optimized](#)

Returns a resolve function that resolves IP an 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 ...

[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]
```

[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-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.

[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-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-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 ...

## 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-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.

## 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-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-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 ...

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

```

```

;; keywords act like functions on maps
(:b {:a 1 :b 2})
=> 2

```

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
```

[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"))
```

[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))
```

[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))
```

[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]}
```

[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
```

[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)
```

## 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 "..."}<br>@ |
| :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) ("if" :symbol-special-form) (" " :whitespaces) ("(" :parenthesis-begin) ("=" :
symbol-function-name) (" " :whitespaces) ("1" :number) (" " :whitespaces) ("2" :number) (")" :parenthesis-end)
(" " :whitespaces) ("true" :constant) (" " :whitespaces) ("false" :constant) (")" :parenthesis-end))
```

top

## host-address

(host-address)

Returns this host's ip address.

```
(host-address)
=> "127.0.0.1"
```

### SEE ALSO

[host-name](#)

Returns this host's name.

top

## host-name

(host-name)

Returns this host's name.

```
(host-name)
=> "saturn.local"
```

## SEE ALSO

[host-address](#)

Returns this host's ip address.

[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)
```

[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.

## 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.

## 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.

## import

```
(import class)
```

Imports a Java class. Imports are bound to the current namespace.

```
(do
  (import :java.lang.Math)
  (. :Math :max 2 10))
=> 10

(ns alpha)
(import :java.lang.Math)
(println "alpha:" (any? #(== % :java.lang.Math) (imports)))

(ns beta)
(println "beta:" (any? #(== % :java.lang.Math) (imports)))

(ns alpha)
(println "alpha:" (any? #(== % :java.lang.Math) (imports)))
)
alpha: true
beta: false
alpha: true
=> nil
```

### SEE ALSO

[imports](#)

List the registered imports for the current namespace.

## imports

```
(imports)
```

List the registered imports for the current namespace.

```
(do
  (import :java.lang.Math)
  (imports))
=> (:com.github.jlangch.venice.ValueException :com.github.jlangch.venice.VncException :java.lang.Exception :
java.lang.IllegalArgumentException :java.lang.Math :java.lang.NullPointerException :java.lang.RuntimeException :
java.lang.Throwable)
```

### SEE ALSO

[import](#)

Imports a Java class. Imports are bound to the current namespace.

## 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

[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

[top](#)

## 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.

[top](#)

## int

```
(int x)
```

Converts to int

```
(int 1)  
=> 1I
```

```
(int nil)  
=> 0I
```

```
(int false)  
=> 0I
```

```
(int true)  
=> 1I
```

```
(int 1.2)  
=> 1I
```

```
(int 1.2M)  
=> 1I
```

```
(int "1")  
=> 1I
```

```
(int (char "A"))  
=> 65I
```

[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

```
(int-array '(1I 2I 3I))
=> [1I, 2I, 3I]

(int-array '(1I 2 3.2 3.56M))
=> [1I, 2I, 3I, 3I]

(int-array 10)
=> [0I, 0I, 0I, 0I, 0I, 0I, 0I, 0I, 0I, 0I]

(int-array 10 42I)
=> [42I, 42I, 42I, 42I, 42I, 42I, 42I, 42I, 42I, 42I]
```

[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
```

[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)
```

[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"
```

[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). For list, vectors and ordered maps the values are ...

#### [disj](#)

Returns a new set with the x, xs removed.

[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 [] {1 2, 3 4})
=> [[1 2] [3 4]]

(into '() '(1 2 3))
=> (3 2 1)

(into [1 2 3] '(4 5 6))
=> [1 2 3 4 5 6]

(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]

(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)
```

[top](#)

## io/->uri

```
(io/->uri s)
(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 (an URI spec)
- a `java.io.File`
- a `java.nio.file.Path`
- a `java.net.URL`

```
(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.

[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 (an URL spec)
- a `java.io.File`
- a `java.nio.file.Path`
- a `java.net.URI`

```
(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.

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 ...

top

## io/buffered-reader

```
(io/buffered-reader is encoding?)
(io/buffered-reader rdr)
```

Creates a `java.io.BufferedReader` from a `java.io.InputStream` `is` with optional encoding (defaults to `:utf-8`), from a `Reader` or from a string.

```
(do
  (import :java.io.ByteArrayInputStream)
  (let [data (byte-array [108 105 110 101 32 49 10 108 105 110 101 32 50])]
    is (. :ByteArrayInputStream :new data)
    rd (io/buffered-reader is :utf-8)]
  (println (. rd :readLine))
  (println (. rd :readLine))))
line 1
line 2
=> nil
```

```
(do
  (let [rd (io/buffered-reader "1\n2\n3\n4")]
    (println (. rd :readLine))
    (println (. rd :readLine))))
1
2
=> nil
```

#### SEE ALSO

[io/buffered-writer](#)

Creates a `java.io.BufferedWriter` from a `java.io.OutputStream` `os` with optional encoding (defaults to `:utf-8`) or from a `Writer`.

top

## io/buffered-writer

```
(io/buffered-writer os encoding?)
(io/buffered-writer wr)
```

Creates a `java.io.BufferedWriter` from a `java.io.OutputStream` `os` with optional encoding (defaults to `:utf-8`) or from a `Writer`.

#### SEE ALSO

[io/buffered-reader](#)

Creates a `java.io.BufferedReader` from a `java.io.InputStream` `is` with optional encoding (defaults to `:utf-8`), from a `Reader` or from a string.

top

## io/bytebuf-in-stream

```
(io/bytebuf-in-stream buf)
```

Returns a `java.io.InputStream` from a `bytebuf`.

```
(io/bytebuf-in-stream (bytebuf [97 98 99]))
```

#### 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`.

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

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 ...

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

#### 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/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/copy-stream](#)

Copies the input stream to the output stream. Returns nil or throws a VncException. Input and output must be a `java.io.InputStream` ...

top

## io/copy-stream

```
(io/copy-stream in-stream out-stream)
```

Copies the input stream to the output stream. Returns nil or throws a VncException. 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 ...

top

## io/default-charset

(io/default-charset)

Returns the default charset.

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-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](#)

Deletes a file f on JVM exit. f 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/move-file](#)

Moves source to target. Returns nil or throws a VncException. Source and target must be a file or a string (file path).

top

## io/delete-file-on-exit

(io/delete-file-on-exit f)

Deletes a file f on JVM exit. f must be a file or a string (file path).

#### 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 ...

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-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](#)

Deletes a file f on JVM exit. f must be a file or a string (file path).

top

## io/download

```
(io/download uri & options)
```

Downloads the content from the uri and reads it as text (string) or binary (bytebuf).

Options:

|                    |                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------|
| :binary true/false | 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                                                                 |
| :conn-timeout val  | e.g.: <code>:conn-timeout 10000</code> , connection timeout in milliseconds.<br>0 is interpreted as an infinite timeout. |
| :read-timeout val  | e.g.: <code>:read-timeout 10000</code> , read timeout in milliseconds.<br>0 is interpreted as an infinite timeout.       |
| :progress-fn fn    | a progress function that takes 2 args<br>[1] progress (0..100%)<br>[2] status {:start :progress :end :failed}            |

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.

```
(-<> "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" <>)))
```

top

## 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/file-symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[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/file-symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[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
```

## 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-path](#)

Returns the absolute path of the file *f*. *f* must be a file or a string (file path).

### [io/file-canonical-path](#)

Returns the canonical path of the file *f*. *f* must be a file or a string (file path).

[top](#)

## io/file-absolute-path

```
(io/file-absolute-path f)
```

Returns the absolute path of the file *f*. *f* must be a file or a string (file path).

```
(io/file-absolute-path (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-path](#)

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 ...

[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-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/file-symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[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-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/file-symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[top](#)

## 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-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/file-symbolic-link?](#)

Returns true if the file `f` exists and is a symbolic link. `f` must be a file or a string (file path).

[top](#)

## [io/file-canonical-path](#)

```
(io/file-canonical-path f)
```

Returns the canonical path of the file `f`. `f` must be a file or a string (file path).

```
(io/file-canonical-path (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-path](#)

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 ...

[top](#)

## [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).

[top](#)

## [io/file-ext?](#)

```
(io/file-ext? f ext)
```

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
```

## SEE ALSO

[io/file-ext](#)

Returns the file extension of a file. f must be a file or a string (file path).

[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/file-symbolic-link?](#)

Returns true if the file f exists and is a symbolic link. f must be a file or a string (file path).

[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")`

## 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`.

[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).

[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-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 ...

[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 ...

[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-path](#)

Returns the absolute path of the file *f*. *f* must be a file or a string (file path).

[io/file-canonical-path](#)

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 ...

[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 ...

[top](#)

## io/file-symbolic-link?

```
(io/file-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/file-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).

[top](#)

## io/file?

```
(io/file? x)
```

Returns true if `x` is a `java.io.File`.

```
(io/file? (io/file "/tmp/test.txt"))  
=> true
```

[top](#)

## io/gzip

```
(io/gzip f)
```

`gzipt 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](#)

`ungzipt 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`.

[top](#)

## io/gzip-to-stream

```
(io/gzip f os)
```

gzipt 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](#)

gzipt f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

[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](#)

gzipt f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

[top](#)

## 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? "http://www.google.com")
```

[top](#)

## 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-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`

[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-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-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`

[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:

- `*.` Matches a path that represents a file name ending in `.txt`
- `txt`
- `*.` Matches file names containing a dot
- `*` Matches file names ending with `.txt` or `.xml`

```
*.  
{txt,  
xml}  
foo.?    Matches file names starting with foo. and a single character extension  
/home/*/*Matches /home/gus/data on UNIX platforms  
/home/** Matches /home/gus and /home/gus/data on UNIX platforms  
C:       Matches C:\\foo and C:\\bar on the Windows platform  
\\*
```

```
(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 ...

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.

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"
```

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).

[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).

[top](#)

## io/move-file

```
(io/move-file source target)
```

Moves source to target. Returns nil or throws a VncException. Source and target must be a file or a string (file path).

### 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 ...

[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"

- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `java.net.URI`

Options:

:binary true/false     e.g.: `:binary true`, defaults to false

:encoding enc     e.g.: `:encoding :utf-8`, defaults to `:utf-8`

## 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/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.

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"`
- `java.io.File`, e.g: `(io/file "/temp/foo.json")`
- `java.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `java.net.URI`

Options:

:encoding enc     e.g.: `:encoding :utf-8`, defaults to `:utf-8`

```
(->> "1\n2\n3"
      io/string-in-stream
      io/slurp-lines)
=> ("1" "2" "3")
```

## 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/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`.

[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.

Options:

`:binary` true/false     e.g.: `:binary true`, defaults to false

`:encoding` enc     e.g.: `:encoding :utf-8`, defaults to `:utf-8`

```
(do
  (import :java.io.FileInputStream)
  (let [file (io/temp-file "test-", ".txt")]
    (io/delete-file-on-exit file)
    (io/spit file "123456789" :append true)
    (try-with [is (. :FileInputStream :new file)]
      (io/slurp-stream is :binary false)))
  )
=> "123456789"
```

### SEE ALSO

#### [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.

[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:

e.g.: `:append true` , defaults to false  
:append true/false  
e.g.: `:encoding :utf-8` , defaults to :utf-8  
:encoding enc

## 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/slurp](#)

Reads the content of file `f` as text (string) or binary (bytebuf).

[io/slurp-lines](#)

Read all lines from `f`.

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:

e.g.: `:flush true`, defaults to false  
:flush true/false  
e.g.: `:encoding :utf-8`, defaults to :utf-8  
:encoding enc

```
(do
  (import :java.io.FileOutputStream)
  (let [file (io/temp-file "test-", ".txt")]
    (io/delete-file-on-exit file)
    (try-with [os (. :FileOutputStream :new file)]
      (io/spit-stream os "123456789" :flush true)))
  )
=> nil
```

## SEE ALSO

[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.

top

## io/string-in-stream

`(io/string-in-stream s & options)`

Returns a `java.io.InputStream` for the string `s`.

Options:

e.g.: `:encoding :utf-8` , defaults to :utf-8  
:encoding enc

```
(io/string-in-stream "The quick brown fox jumped over the lazy dog")
```



## 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`.

[top](#)

## io/temp-file

`(io/temp-file prefix suffix)`

Creates an empty temp file with the given prefix and suffix.

```
(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 temp directory with prefix.

[top](#)

## io/tmp-dir

`(io/tmp-dir)`

Returns the tmp dir as a `java.io.File`.

```
(io/tmp-dir)
=> /var/folders/rm/pjqr5p1n3db4mxh5qq1j5yh80000gn/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 temp directory with prefix.

[top](#)

## 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.

[top](#)

## 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.

[top](#)

## 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

[top](#)

## io/unzip-all

```
(io/unzip-all 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.

```
(-> (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]}
```

### SEE ALSO

#### [io/unzip-to-dir](#)

Unzips *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

[top](#)

## 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 *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

[top](#)

## [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 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

[top](#)

## [io/unzip-to-dir](#)

```
(io/unzip-to-dir f dir)
```

Unzips 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`

[top](#)

## io/uri-stream

`(io/uri-stream uri)`

Returns a `java.io.InputStream` from the uri.

```
(-> (io/uri-stream "https://www.w3schools.com/xml/books.xml")
    (io/slurp-stream :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 ...

[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`.

[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`.

[top](#)

## io/watch-dir

```
(io/watch-dir dir event-fn)
(io/watch-dir dir event-fn failure-fn)
(io/watch-dir dir event-fn failure-fn termination-fn)
```

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.

`event-fn` is a two argument function that receives the path and mode `{:created, :deleted, :modified}` of the changed file.

`failure-fn` is a two argument function that receives the watch dir and the failure exception.

`termination-fn` is a one argument function that receives the watch dir.

Returns a *watcher* that is actively watching a directory. The *watcher* is a resource which should be closed with `(io/close-watcher w)`.

```
(do
  (defn log [msg] (locking log (println msg)))

  (let [w (io/watch-dir "/tmp"
                        #(log (str %1 " " %2))
                        (sleep 30 :seconds)
                        (io/close-watcher w)))]

    (do
      (defn log [msg] (locking log (println msg)))

      (let [w (io/watch-dir "/tmp"
                            #(log (str %1 " " %2))
                            #(log (str "failure " (:message %2)))
                            #(log (str "terminated watching " %1)))]

        (sleep 30 :seconds)
        (io/close-watcher w))))
```

### SEE ALSO

[io/await-for](#)

Blocks the current thread until the file has been created, deleted, or modified according to the passed modes `{:created, :deleted, ...`

[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`).

```
(do
  (import :java.io.ByteArrayInputStream)
  (let [data (byte-array [108 105 110 101 32 49 10 108 105 110 101 32 50])]
    is (. :ByteArrayInputStream :new data)
    rd (io/wrap-is-with-buffered-reader is :utf-8)]
  (println (. rd :readLine))
  (println (. rd :readLine))))
line 1
line 2
=> nil
```

## SEE ALSO

[io/buffered-reader](#)

Creates a `java.io.BufferedReader` from a `java.io.InputStream` `is` with optional encoding (defaults to `:utf-8`), from a `Reader` or from a string.

[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`).

```
(do
  (import :java.io.ByteArrayOutputStream)
  (let [os (. :ByteArrayOutputStream :new)
        wr (io/wrap-os-with-buffered-writer os :utf-8)]
    (. wr :write "line 1")
    (. wr :newLine)
    (. wr :write "line 2")
    (. wr :flush)
    (. os :toByteArray)))
=> [108 105 110 101 32 49 10 108 105 110 101 32 50]
```

## 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`).

[top](#)

## 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`).

```
(do
  (import :java.io.ByteArrayOutputStream)
  (let [os (. :ByteArrayOutputStream :new)
        wr (io/wrap-os-with-print-writer os :utf-8)]
    (. wr :println "line 1")
    (. wr :println "line 2"))
```

```
(. wr :flush)
(. os :toByteArray)))
=> [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`).

[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.

[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.

[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 <i>out</i> , 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 ...

[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 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 ...

[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.

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.

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 ...

[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 ...

[top](#)

## 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
```

[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)  
=> "1.3.28"
```

#### SEE ALSO

[java-package-version](#)

Returns version information for a Java package or nil if the package does not exist or is not visible.

[top](#)

## java-enumeration-to-list

```
(java-enumeration-to-list e)
```

Converts a Java enumeration to a list

[top](#)

## java-iterator-to-list

```
(java-iterator-to-list e)
```

Converts a Java iterator to a list

[top](#)

## java-major-version

```
(java-major-version)
```

Returns the Java major version (8, 9, 11, ...).

```
(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.

[top](#)

## java-obj?

```
(java-obj? obj)
```

Returns true if obj is a Java object

```
(java-obj? (. :java.math.BigInteger :new "0"))  
=> true
```

[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 "Temurin" :implementation-version  
"1.8.0_322" :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 "Temurin" :implementation-version  
"1.8.0_322" :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.

[top](#)

## java-source-location

```
(java-source-location class)
```

Returns the path of the source location of a class (fully qualified class name).

```
(java-source-location :com.github.jlangch.venice.Venice)
```

[top](#)

## java-unwrap-optional

```
(java-unwrap-optional val)
```

Unwraps a Java `java.util.Optional` to its contained value or nil

top

## java-version

(java-version)

Returns the Java VM version (1.8.0\_252, 11.0.7, ...)

```
(java-version)
=> "1.8.0_322"
```

### SEE ALSO

[java-major-version](#)

Returns the Java major version (8, 9, 11, ...).

[java-version-info](#)

Returns the Java VM version info.

top

## java-version-info

(java-version-info)

Returns the Java VM version info.

```
(java-version-info)
=> {:version "1.8.0_322" :vendor "Temurin" :vm-version "25.322-b06" :vm-name "OpenJDK 64-Bit Server VM" :vm-
vendor "Temurin"}
```

### 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, 9, 11, ...).

top

## java/javadoc

(javadoc class-or-object)

Opens a browser window displaying the javadoc for argument.

```
(java/javadoc :java.lang.String)
```

top

## json/pretty-print

```
(json/pretty-print s)
```

Pretty prints a JSON string

```
(json/pretty-print (json/write-str {:a 100 :b 100}))  
=> "{\n  \"a\": 100,\n  \"b\": 100\n}"
```

### 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. out maybe a file, a Java OutputStream, or a Java Writer.

[json/slurp](#)

Slurps a JSON data from a source and returns it as a Venice data.

[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. out maybe a file, a Java OutputStream, or a Java Writer.

[json/slurp](#)



Slurps a JSON data from a source and returns it as a Venice data.

[json/pretty-print](#)

Pretty prints a JSON string

[top](#)

## 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.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `java.net.URI`

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})
      data (bytebuf-from-string json :utf-8)
      in (. :java.io.ByteArrayInputStream :new data)]
  (str (json/slurp in)))
=> "{a 100 b 100}"
```

### 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. out maybe a file, a Java OutputStream, or a Java Writer.

[json/pretty-print](#)

Pretty prints a JSON string

[top](#)

## json/spit

(json/spit out val & options)

Spits the JSON converted val to the output. out maybe a file, a Java OutputStream, or a Java 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

```
(let [out (. :java.io.ByteArrayOutputStream :new)]
  (json/spit out {:a 100 :b 100 :c [10 20 30]})
  (. out :flush)
  (. :java.lang.String :new (. out :toByteArray) "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

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. out maybe a file, a Java OutputStream, or a Java Writer.

[json/slurp](#)

Slurps a JSON data from a source and returns it as a Venice data.

[json/pretty-print](#)

Pretty prints a JSON string

top

## just

```
(just x)
```

Creates a wrapped x, that is dereferenceable

```
(just 10)
=> (just 10)

(just "10")
=> (just "10")

(deref (just 10))
=> 10
```

top

## just?

```
(just? x)
```

Returns true if x is of type just

```
(just? (just 1))
=> true
```

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}}
```

## 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)
```

## 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.

## 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 ...

top

## keyword

```
(keyword name)
```

Returns a keyword from the given name

```
(keyword "a")  
=> :a
```

```
(keyword :a)  
=> :a
```

top

## keyword?

```
(keyword? x)
```

Returns true if x is a keyword

```
(keyword? (keyword "a"))  
=> true
```

```
(keyword? :a)  
=> true
```

```
(keyword? nil)  
=> false
```

```
(keyword? 'a)  
=> false
```

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.

[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.

## 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.

## 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.

[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
```

[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 sequence ends if the supplier returns nil.
```

```
(lazy-seq seed f)
(theoretically) infinitely lazy sequence with a seed value and a function to calculate the next value based on the previous.
```

```
(lazy-seq head tail-lazy-seq)
Constructs 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))
=> (3461245156057646284 349287172907692845 8676503237576948681 637434901075546825)

; 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)

; 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)

; 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)
```

## SEE ALSO

[doall](#)

When lazy sequences are produced doall can be used to force any effects and realize the lazy sequence.

[lazy-seq?](#)

Returns true if obj is a lazyseq

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.

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

[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.

[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])
```

[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). For list, vectors and ordered maps the values are ...

### [vector\\*](#)

Creates a new vector containing the items prepended to the rest, the last of which will be treated as a collection.

[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-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 (seq "abc") y [0 1 2]] [x y])
=> ([#\a 0] [#\a 1] [#\a 2] [#\b 0] [#\b 1] [#\b 2] [#\c 0] [#\c 1] [#\c 2])
```

## SEE ALSO

### [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.

[top](#)

## list?

```
(list? obj)
```

Returns true if obj is a list

```
(list? (list 1 2))
=> true
```

```
(list? '(1 2))  
=> true
```

[top](#)

## load-classpath-file

```
(load-classpath-file name)  
(load-classpath-file name force)  
(load-classpath-file name alias)  
(load-classpath-file name force alias)
```

Sequentially read and evaluate the set of forms contained in the classpath file. The function is restricted to classpath files with the extension '.venice'.

```
(do  
  (load-classpath-file "com/github/jlangch/venice/test.venice")  
  (test/test-fn "hello"))  
=> "test: hello"  
  
(do  
  (load-classpath-file "com/github/jlangch/venice/test.venice")  
  (test/test-fn "hello")  
  ; reload the classpath file  
  (ns-remove 'test)  
  (load-classpath-file "com/github/jlangch/venice/test.venice" true)  
  (test/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.

[top](#)

## load-file

```
(load-file file)  
(load-file file force)  
(load-file file alias)  
(load-file file force alias)
```

Sequentially read and evaluate the set of forms contained in the file.

If the file can not be found on the global load paths and the sandbox permits the file is either loaded from the current working directory if it has a relative path or it is loaded from its absolute path.

With 'force' set to false (the default) the file is only loaded once and then served from a cache. With 'force' set to true it is always loaded physically.

The function is restricted to load files with the extension '.venice'. If the file extension is missing '.venice' will be implicitly added. Returns 'true' if the file has been successfully loaded and 'false' if the file has been already loaded. Throws an exception on loading error.

```
(load-file "coffee")

(load-file "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.

[top](#)

## load-module

```
(load-module m)
(load-module m force)
(load-module m alias)
(load-module m force alias)
```

Loads a Venice predefined extension module.

Returns `true` if the module has been successfully loaded and `false` if the module has been already loaded. Throws an exception on loading error.

```
(load-module :trace)
=> nil

;; loading the :trace modul and define a ns alias 't for namespace
;; 'trace used in the module
(load-module :trace ['trace :as 't])
=> nil

;; reloading a module
(do
  (load-module :trace)
  ; reload the module
  (ns-remove 'trace)
  (load-module :trace true))
=> :trace
```

[top](#)

## load-resource

```
(load-resource res & options)
```

Loads a resource from the defined load paths. Returns a string, a bytebuffer or nil if the file does not exist.

Options:

:binary b	e.g :binary true, defaults to true
:encoding e	e.g :encoding :utf-8, defaults to :utf-8

[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 ...

[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

(do
  (defn log [msg] (locking log (println msg)))
  (log "message"))
message
=> nil
```

[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

[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

[top](#)

# long

(long x)

Converts to long

```
(long 1)
=> 1

(long nil)
=> 0

(long false)
=> 0

(long true)
=> 1

(long 1.2)
=> 1

(long 1.2M)
=> 1

(long "1")
=> 1

(long (char "A"))
=> 65
```

[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

```
(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]
```

[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
```

[top](#)

## loop

```
(loop [bindings*] exprs*)
```

Evaluates the exprs and binds the bindings. Creates a recursion point with the bindings.

```
;; 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

[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 ...

[top](#)

## macro?

```
(macro? x)
```

Returns true if x is a macro

```
(macro? and)
=> true
```

[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.

[top](#)

## macroexpand-all

```
(macroexpand-all form)
```

Recursively expands all macros in the form.

```
(macroexpand-all '(and true true))
=> (let [cond__28954__auto true] (if cond__28954__auto true cond__28954__auto))

(macroexpand-all '(and true (or true false) true))
=> (let [cond__28989__auto true] (if cond__28989__auto (let [cond__28989__auto (let [cond__28990__auto true]
(if cond__28990__auto cond__28990__auto false))] (if cond__28989__auto true cond__28989__auto))
cond__28989__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

[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]
```

[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.

```
(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 (fn [e] [(key e) (inc (val e))]) {:a 1 :b 2})
=> ([:a 2] [:b 3])

(map inc #{1 2 3})
=> (2 3 4)
```

### 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 ...

[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.

[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 ...

[top](#)

## map-indexed

```
(map-indexed f coll)
```

Retruns 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])
```

```
(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})
```

[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}
```

[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}
```

[top](#)

## 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}
```

## 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 (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 :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 ...



## 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]]
```

[top](#)

## match?

```
(match? s regex)
```

Returns true if the string `s` matches the regular expression `regex`

```
(match? "1234" "[0-9]+")  
=> true
```

```
(match? "1234ss" "[0-9]+")  
=> false
```

[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`.

[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  $\pi$

[math/atan](#)

Returns the arc tangent of a value; the returned angle is in the range  $-\pi/2$  through  $\pi/2$ .

[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  $\pi$

[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

[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)
=> 4.6000000000000000M
```

### 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

[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

[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

[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

[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

[top](#)

## math/softmax

```
(math/softmax coll)
```

Softmax algorithm

```
(math/softmax [3.2 1.3 0.2 0.8])  
=> [0.7751495482986049 0.1159380476300716 0.03859242355646149 0.07031998051486205]
```

[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

[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

[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 ...

[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 ...

[top](#)

## maven/download

```
(maven/download artefact options*)
```

Downloads an artefact in the format 'group-id:artefact-id:version' from a Maven repository. Can download any combination of the jar, sources, or pom artefacts to a directory.

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

```
(maven/download "org.knowm.xchart:xchart:3.6.1")
```

```
(maven/download "org.knowm.xchart:xchart:3.6.1" :sources true :pom true)
```

```
(maven/download "org.knowm.xchart:xchart:3.6.1" :dir "." :jar false :sources true)
```

```
(maven/download "org.knowm.xchart:xchart:3.6.1" :dir "." :sources true)
```

```
(maven/download "org.knowm.xchart:xchart:3.6.1" :dir "." :sources true :repo "https://repo1.maven.org/maven2")
```

```
(maven/download "org.knowm.xchart:xchart:3.6.1" :dir "." :silent false)
```

## SEE ALSO

[maven/get](#)

Downloads artefact in the format 'group-id:artefact-id:version' from a Maven repository. The artefact type 'type' is one of {jar, ...

[maven/uri](#)

Returns an URI for an artefact in the format 'group-id:artefact-id:version' from a Maven repository.

[maven/parse-artefact](#)

Parses a Maven artefact like 'com/vaadin:vaadin-client:8.7.2'

[top](#)

## maven/get

```
(maven/get artefact type options*)
```

Downloads artefact in the format 'group-id:artefact-id:version' from a Maven repository. The artefact type 'type' is one of {jar, :sources, :pom}.

Returns the artefact 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

```
(maven/get "org.knowm.xchart:xchart:3.6.1" :jar)
```

```
(maven/get "org.knowm.xchart:xchart:3.6.1" :jar :silent false)
```

```
(maven/get "org.knowm.xchart:xchart:3.6.1" :sources)
```

```
(maven/get "org.knowm.xchart:xchart:3.6.1" :jar :repo "https://repo1.maven.org/maven2")
```

## SEE ALSO

[maven/download](#)

Downloads an artefact in the format 'group-id:artefact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/uri](#)

Returns an URI for an artefact in the format 'group-id:artefact-id:version' from a Maven repository.

[maven/parse-artefact](#)

Parses a Maven artefact like 'com/vaadin:vaadin-client:8.7.2'

[top](#)

## maven/parse-artefact

```
(maven/parse-artefact artefact)
```

```
(maven/parse-artefact artefact file-suffix)
```

```
(maven/parse-artefact artefact file-suffix repo)
```

Parses a Maven artefact like 'com/vaadin:vaadin-client:8.7.2'

1. (maven/parse-artefact artefact)  
returns a vector with group-id, artefact-id, and version



2. (maven/parse-artefact artefact file-suffix)  
returns a vector with group-id, artefact-id, version and file name
3. (maven/parse-artefact artefact file-suffix repo)  
returns a vector with the Maven download URI and the file name

```
(maven/parse-artefact "org.knowm.xchart:xchart:3.6.1")
```

```
(maven/parse-artefact "org.knowm.xchart:xchart:3.6.1"  
                      ".jar")
```

```
(maven/parse-artefact "org.knowm.xchart:xchart:3.6.1"  
                      ".jar"  
                      "https://repo1.maven.org/maven2")
```

## SEE ALSO

[maven/download](#)

Downloads an artefact in the format 'group-id:artefact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/get](#)

Downloads artefact in the format 'group-id:artefact-id:version' from a Maven repository. The artefact type 'type' is one of {jar, ...

[maven/uri](#)

Returns an URI for an artefact in the format 'group-id:artefact-id:version' from a Maven repository.

[top](#)

## maven/uri

```
(maven/uri artefact type options*)
```

Returns an URI for an artefact in the format 'group-id:artefact-id:version' from a Maven repository.

The artefact type 'type' is one of {jar, :sources, :pom}

Options:

:repo maven-repo     a maven repo, defaults to "https://repo1.maven.org/maven2"

```
(maven/uri "org.knowm.xchart:xchart:3.6.1" :jar)
```

```
(maven/uri "org.knowm.xchart:xchart:3.6.1" :jar :repo "https://repo1.maven.org/maven2")
```

## SEE ALSO

[maven/download](#)

Downloads an artefact in the format 'group-id:artefact-id:version' from a Maven repository. Can download any combination of the jar, ...

[maven/get](#)

Downloads artefact in the format 'group-id:artefact-id:version' from a Maven repository. The artefact type 'type' is one of {jar, ...

[maven/parse-artefact](#)

Parses a Maven artefact like 'com.vaadin:vaadin-client:8.7.2'

[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

[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: 1.95ms
=> 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 ...

[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.

[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 ...

[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.

[top](#)

## meta

```
(meta obj)
```

Returns the metadata of obj, returns nil if there is no metadata.

```
(meta (vary-meta [1 2] assoc :a 1))  
=> {:a 1 :line 21 :column 28 :file "example"}
```

[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

## mod

(mod n d)

Modulus of n and d.

```
(mod 10 4)  
=> 2
```

```
(mod -1 5)  
=> 4
```

```
(mod 10I 4I)  
=> 2I
```

## 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.

## modules

(modules)

Lists the available modules

## 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])
```

[top](#)

## mutable-list?

```
(mutable-list? obj)
```

Returns true if obj is a mutable list

```
(mutable-list? (mutable-list 1 2))
=> true
```

[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}
```

[top](#)

## mutable-map?

```
(mutable-map? obj)
```

Returns true if obj is a mutable map

```
(mutable-map? (mutable-map :a 1 :b 2))
=> true
```

[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]}
```

[top](#)

## mutable-set?

```
(mutable-set? obj)
```

Returns true if obj is a mutable-set

```
(mutable-set? (mutable-set 1))
=> true
```

[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]]
```

[top](#)

## mutable-vector?



```
(mutable-vector? obj)
```

Returns true if obj is a mutable vector

```
(mutable-vector? (mutable-vector 1 2))  
=> true
```

[top](#)

## name

```
(name x)
```

Returns the name String of a string, symbol, keyword, or function

```
(name :x)  
=> "x"
```

```
(name 'x)  
=> "x"
```

```
(name "x")  
=> "x"
```

```
(name str/digit?)  
=> "digit?"
```

### SEE ALSO

[namespace](#)

Returns the namespace string of a symbol, keyword, or function.

[fn-name](#)

Returns the qualified name of a function or macro

[top](#)

## namespace

```
(namespace x)
```

Returns the namespace string of a symbol, keyword, or function.

```
(namespace 'user/foo)  
=> "user"
```

```
(namespace :user/foo)  
=> "user"
```

```
(namespace str/digit?)  
=> "str"
```

### SEE ALSO

## [name](#)

Returns the name String of a string, symbol, keyword, or function

## [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

[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

[top](#)

# nano-time

```
(nano-time)
```

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

```
(nano-time)
```

```
=> 195451005531495
```

```
(let [t (nano-time)
      _ (sleep 100)
      e (nano-time)]
```

```
(format-nano-time (- e t) :precision 2))  
=> "105.21ms"
```

## SEE ALSO

[current-time-millis](#)

Returns the current time in milliseconds.

[format-nano-time](#)

Formats a time given in nanoseconds as long or double.

[top](#)

## neg?

```
(neg? x)
```

Returns true if x smaller than zero else false

```
(neg? -3)  
=> true
```

```
(neg? 3)  
=> false
```

```
(neg? (int -3))  
=> 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

[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.

[top](#)

## newline

```
(newline)
(newline os)
```

Without arg writes a platform-specific newline to the output stream that is the current value of `*out*`. With arg writes a newline to the passed output stream.  
Returns `nil`.

```
(newline)
=> nil
```

```
(newline *out*)
=> nil
```

```
(newline *err*)
=> nil
```

## SEE ALSO

[print](#)

Prints the values xs to the output stream that is the current value of `*out*` or to the passed output stream os if given.

[println](#)

Prints the values xs to the output 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 output stream that is the current value of `*out*`. With an output ...

[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.

[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

[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.

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

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
```

[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
```

[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
```

[top](#)

## not-every?

```
(not-every? pred coll)
```

Returns false if the predicate is true for all collection items, true otherwise

```
(not-every? number? nil)
=> true

(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
```

[top](#)

## not-match?

```
(not-match? s regex)
```

Returns true if the string s does not match the regular expression regex

```
(not-match? "1234" "[0-9]+")
=> false

(not-match? "1234ss" "[0-9]+")
=> true
```

[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-unmap](#)

Removes the mappings for the symbol from the namespace.

### [ns-remove](#)

Removes the mappings for all symbols from the namespace.

### [ns-list](#)

Lists all the symbols in the 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 ...

### [namespace](#)

Returns the namespace string of a symbol, keyword, or function.

### [var-ns](#)

Returns the namespace of the var's symbol

[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.

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.

top

## ns-list

(ns-list ns)

Lists all the symbols in the namespace ns.

```
(ns-list 'regex)
=> (regex/count regex/find regex/find+ regex/find-all regex/find-all+ regex/find? regex/group regex/matcher
  regex/matches regex/matches? regex/pattern regex/reset)
```

### 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.

## var-ns

Returns the namespace of the var's symbol

[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

Lists all the symbols in the namespace ns.

#### namespace

Returns the namespace string of a symbol, keyword, or function.

#### var-ns

Returns the namespace of the var's symbol

[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.

[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](#)

Lists all the symbols in the namespace ns.

#### [namespace](#)

Returns the namespace string of a symbol, keyword, or function.

#### [var-ns](#)

Returns the namespace of the var's symbol

[top](#)

## nth

```
(nth coll idx)
```

Returns the nth element of coll.

```
(nth nil 1)
=> nil
```

```
(nth [1 2 3] 1)
```

```
=> 2
```

```
(nth '(1 2 3) 1)
```

```
=> 2
```

```
(nth "abc" 2)
```

```
=> #\c
```

[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
```

[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, 3.45M, 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, 42]
```

## 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? (int 4))
```

```
=> false
```

### SEE ALSO

[even?](#)

Returns true if n is even, throws an exception if n is not an integer

## offer!

```
(offer! queue v)
```

```
(offer! queue timeout v)
```

Offers an item to 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 [s (queue)]
```

```
  (offer! s 4)
```

```
  (offer! s 3)
```

```
  (poll! s)
```

```
  s)
```

```
=> (3)
```

### SEE ALSO

[queue](#)

Creates a new mutable threadsafe bounded or unbounded queue.

[peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element

[poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

[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

## or

```
(or x)
(or x & next)
```

Ors the predicate forms

```
(or true false)
=> true

(or false false)
=> false

(or)
=> false
```

### SEE ALSO

[and](#)

Ands the predicate forms

[not](#)

Returns true if x is logical false, false otherwise.

[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 the current stage's result ...

#### [accept-either](#)

Returns a new promise that, when either this or the other given promise completess 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.

[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}
```

[top](#)

## ordered-map?

```
(ordered-map? obj)
```

Returns true if obj is an ordered map

```
(ordered-map? (ordered-map :a 1 :b 2))
=> true
```

[top](#)

## os-arch

```
(os-arch)
```



Returns the OS architecture

```
(os-arch)  
=> "x86_64"
```

## SEE ALSO

[os-type](#)

Returns the OS type

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, or :linux

[os-name](#)

Returns the OS name

[os-version](#)

Returns the OS version

[top](#)

## os-name

```
(os-name)
```

Returns the OS name

```
(os-name)  
=> "Mac OS X"
```

## SEE ALSO

[os-type](#)

Returns the OS type

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, or :linux

[os-arch](#)

Returns the OS architecture

[os-version](#)

Returns the OS version

[top](#)

## os-type

```
(os-type)
```

Returns the OS type

```
(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, or :linux

[os-arch](#)

Returns the OS architecture

[os-name](#)

Returns the OS name

[os-version](#)

Returns the OS version

[top](#)

## os-type?

(os-type? type)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, or :linux

```
(os-type? :mac-osx)  
=> true
```

```
(os-type? :windows)  
=> false
```

### SEE ALSO

[os-type](#)

Returns the OS type

[os-arch](#)

Returns the OS architecture

[os-name](#)

Returns the OS name

[os-version](#)

Returns the OS version

[top](#)

## os-version

(os-version)

Returns the OS version

```
(os-version)  
=> "10.16"
```

### SEE ALSO

[os-type](#)

Returns the OS type

[os-type?](#)

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, or :linux

[os-arch](#)

Returns the OS architecture

[os-name](#)

Returns the OS name

[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. To add backtracking parsers can be wrapped with `attempt` !

```
(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)"
)

(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
)
```

[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.

[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")
; => #\?

)
```

[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

)
```

[top](#)

## 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

)
```

[top](#)

## 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
)
```

[top](#)

## 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
)
```

[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]
)
```

[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
)
```

[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]
)
```

[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
)
```

[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
)
```

[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]
)
```

[top](#)

## parsifal/eof

(eof)

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.

```
(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
)
```

[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
)
```

## 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. To add backtracking parsers can be wrapped with `attempt` !

```
(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"
)

(do
  (load-module :parsifal ['parsifal :as 'p])

  ; Backtracking

  ; No implicit 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

  ; Explicit backtracking using `attempt`!
  (p/run (p/either (p/attempt (p/let->> [c (p/letter)
   d (p/digit)]
                                     (p/always (list c d))))
          (p/letter))
    "abc")
  ; => #\a
)
```

## 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
)
```

## 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
)
```

## 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]
)

```

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!"
)

```

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")

```

```
; => []  
)
```

[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  
)
```

[top](#)

## 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  
)
```

[top](#)

## 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
)
```

[top](#)

## 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
)
```

[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)]
)
```

[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
)
```

[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
)
```

[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]
)
```

[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
)
```

[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
```

[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"))
```

[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 4 3 5 6])
=> ((1) (2 4) (3 5) (6))

(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))
```

## 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 ...

#### [cpus](#)

Returns the number of available processors or number of hyperthreads if the CPU supports hyperthreads.

## pdf/available?

```
(pdf/available?)
```

Checks if the 3rd party libraries required for generating PDFs are available.

```
(pdf/available?)
```

## 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)
```

## 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.

[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.

[top](#)

## pdf/pages

```
(pdf/pages pdf)
```

Returns the number of pages of a PDF. The PDF is passed as bytebuf.

```
(->> (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.

[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

```
(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 ...

[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/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.

## peek

```
(peek coll)
```

For a list, same as first, for a vector, same as last, for a stack the top element

```
(peek '(1 2 3 4))
=> 1

(peek [1 2 3 4])
=> 4
```

```
(let [s (stack)]
  (push! s 4)
  (peek s))
=> 4
```

[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 ...

[top](#)

## pid

```
(pid)
```

Returns the PID of this process.

```
(pid)
=> "61962"
```

[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: 2.00s
=> (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, ...

### [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.

[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 [s (queue)]
  (offer! s 4)
  (offer! s 3)
  (poll! s)
  s)
=> (3)
```

## SEE ALSO

### [queue](#)

Creates a new mutable threadsafe bounded or unbounded queue.

### [peek](#)

For a list, same as `first`, for a vector, same as `last`, for a stack the top element

### [offer!](#)

Offers an item to a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait time if necessary ...

### [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

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]
```

top

## pop!

```
(pop! stack)
```

Pops an item from a stack.

```
(let [s (stack)]  
  (push! s 4)  
  (push! s 3)  
  (pop! s)  
  s)  
=> (4)
```

### 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

#### [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

top



## pos?

```
(pos? x)
```

Returns true if x greater than zero else false

```
(pos? 3)  
=> true
```

```
(pos? -3)  
=> false
```

```
(pos? (int 3))  
=> 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

[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.

[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.

[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)  
=> 334.72571990233183
```

[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.

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 output stream `os` if given followed by a (newline).

[newline](#)

Without `arg` writes a platform-specific newline to the output stream that is the current value of `*out*`. With `arg` writes a newline to ...

[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 ...

[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"
```

## 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 ...

[top](#)

## 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.

[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.

[top](#)

## print

```
(print & xs)
(print 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.

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 output 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 output stream that is the current value of `*out*`. With an output ...

### [newline](#)

Without `arg` writes a platform-specific newline to the output stream that is the current value of `*out*`. With `arg` writes a newline to ...

[top](#)

## printf

```
(printf fmt & args)
(printf os fmt & args)
```

Without output stream prints formatted output as per format to the output stream that is the current value of `*out*`. With an output stream prints to that output stream. 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

(sprintf *err* "%s: %d" "abc" 100)
abc: 100
=> nil
```

## SEE ALSO

### [print](#)

Prints the values `xs` to the output stream that is the current value of `*out*` or to the passed output stream `os` if given.

### [println](#)

Prints the values `xs` to the output 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 stream that is the current value of `*out*`. With `arg` writes a newline to ...

[top](#)

## println

```
(println & xs)
(println 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 followed by a (newline) .

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 output stream that is the current value of `*out*` or to the passed output stream `os` if given.

### [printf](#)

Without output stream prints formatted output as per format to the output stream that is the current value of `*out*`. With an output ...

### [newline](#)

Without `arg` writes a platform-specific newline to the output stream that is the current value of `*out*`. With `arg` writes a newline to ...

[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 output stream `os` if given followed by a `(newline)`.

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.

#### [newline](#)

Without `arg` writes a platform-specific newline to the output stream that is the current value of `*out*`. With `arg` writes a newline to ...

#### [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 ...

[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 test") ; 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.

[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 500) 10)
  (def p (promise task))
  @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 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.

#### [timeout-after](#)

Returns a promise that timeouts afer the specified time. The promise throws a TimeoutException.

[top](#)

## promise?

(promise? p)

Returns true if f is a Promise otherwise false

```
(promise? (promise))
```

```
=> true
```

## proxify

```
(proxify classname method-map)
```

Proxifies 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 Venice sandbox even if the Java method that invokes the callback methods is running in another thread.

```
(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})))
)
=> []
```

## push!

```
(push! stack v)
```

Pushes an item to a stack.

```
(let [s (stack)]
  (push! s 4)
  (push! s 3)
  (pop! s)
  s)
=> (4)
```

### 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

#### [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

## 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.

[top](#)

## queue

(queue)

(queue 100)

Creates a new mutable threadsafe bounded or unbounded queue.

The queue can be turned into a synchronous queue when using indefinite timeouts for offering and polling values. With a synchronous queue `offer!` waits until the value can be added to the queue and `poll!` 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)

; bounded queue
(let [q (queue 10)]
  (offer! q 1000 1)
  (offer! q 1000 2)
  (offer! q 1000 3)
  (poll! q 1000)
  q)
=> (2 3)

; synchronous unbounded queue
(let [q (queue)]
  (offer! q :indefinite 1)
  (offer! q :indefinite 2)
  (offer! q :indefinite 3)
  (poll! q :indefinite)
  q)
=> (2 3)

; synchronous bounded queue
(let [q (queue 10)]
  (offer! q :indefinite 1)
  (offer! q :indefinite 2)
  (offer! q :indefinite 3)
  (poll! q :indefinite)
  q)
=> (2 3)

```

## SEE ALSO

### [peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element

### [poll!](#)

Polls an item from a queue with an optional timeout in milliseconds. For an indefinite timeout pass the timeout value :indefinite.

### [offer!](#)

Offers an item to a queue with an optional timeout in milliseconds. If a timeout is given waits up to the specified wait time if necessary ...

### [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

[top](#)

## queue?

```
(queue? coll)
```

Returns true if coll is a queue

```
(queue? (queue))
=> true
```

[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.

[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)
```

```
=> 0.7430058190292672
```

```
(rand-double 100.0)
```

```
=> 68.47910820687079
```

### 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-gaussian](#)

Without argument returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0. With argument `mean` and `stddev` ...

[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)
=> 0.543896800624642

(rand-gaussian 0.0 5.0)
=> -3.771952596741772
```

## 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.

[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)
=> 3739199743837580886

(rand-long 100)
=> 85
```

## 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-gaussian](#)

Without argument returns a Gaussian distributed double value with mean 0.0 and standard deviation 1.0. With argument mean and stddev ...

[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)
```

[top](#)

## read-line

```
(read-line)
```

Reads the next line from the stream that is the current value of `*in*`.  
Returns `nil` if the end of the stream is reached.

### SEE ALSO

[read-char](#)

Reads the next char from the stream that is the current value of `*in*`.

[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.

[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, ...



## 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.

## 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 (fn [x y] (+ x y)) [1 2 3 4 5 6 7])
=> 28

(reduce (fn [x y] (+ x y)) 10 [1 2 3 4 5 6 7])
=> 38

((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)))))
=> (19 20 17 13 5 10 18 20)
```

## 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.

[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}
```

[top](#)

## reduced

```
(reduced x)
```

Wraps *x* in a way such that a reduce will terminate with the value *x*.

[top](#)

## reduced?

(reduced? x)

Returns true if x is the result of a call to reduced.

[top](#)

## regex/count

(regex/count matcher)

Returns the matcher's group count.

```
(let [p (regex/pattern "[0-9]+(.*)")
      m (regex/matcher p "100abc")]
  (regex/count m))
=> 2
```

### SEE ALSO

[regex/matcher](#)

Returns an instance of java.util.regex.Matcher.

[top](#)

## regex/find

(regex/find matcher)

Returns the next regex match or nil if there is no further match.

To get the positional data for the matched group use (regex/find+ matcher) .

```
(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

[regex/find-all](#)

Returns all regex matches.

[regex/find+](#)

Returns the next regex match and returns the group with its positional data.

[regex/matcher](#)

Returns an instance of java.util.regex.Matcher.

## regex/find+

(regex/find+ matcher)

Returns the next regex match and returns the group with its positional data.

```
(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

[regex/find-all+](#)

Returns the all regex matches and returns the groups with its positional data

[regex/find](#)

Returns the next regex match or nil if there is no further match.

[regex/matcher](#)

Returns an instance of java.util.regex.Matcher.

## regex/find-all

(regex/find-all matcher)

Returns all regex matches.

To get the positional data for the matched groups use 'regex/find-all+'.

```
(>> (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

[regex/find](#)

Returns the next regex match or nil if there is no further match.

[regex/find-all+](#)

Returns the all regex matches and returns the groups with its positional data

## [regex/matcher](#)

Returns an instance of java.util.regex.Matcher.

[top](#)

## regex/find-all+

(regex/find-all+ matcher)

Returns the all regex matches and returns the groups with its positional data

```
(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

#### [regex/find+](#)

Returns the next regex match and returns the group with its positional data.

#### [regex/find-all](#)

Returns all regex matches.

#### [regex/matcher](#)

Returns an instance of java.util.regex.Matcher.

[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

[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. If the match succeeds then more information can be obtained via the [regex/group](#) function

[top](#)

## regex/group

`(regex/group matcher group)`

Returns the input subsequence captured by the given group during the previous match operation.

```
(let [p (regex/pattern "([0-9]+)(.*)")
      m (regex/matcher p "100abc")]
  (if (regex/matches? m)
    [(regex/group m 1) (regex/group m 2)]
    []))
=> ["100" "abc"]
```

## SEE ALSO

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[regex/matches?](#)

Attempts to match the entire region against the pattern. If the match succeeds then more information can be obtained via the [regex/group](#) function

[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)`.

JavaDoc: [Pattern](#)

```
(regex/matcher "[0-9]+" "100")
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]

(let [p (regex/pattern "[0-9]+")]
  (regex/matcher p "100"))
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]
```

## SEE ALSO

[regex/pattern](#)

Returns an instance of `java.util.regex.Pattern`.

[regex/matches?](#)

Attempts to match the entire region against the pattern. If the match succeeds then more information can be obtained via the `regex/group` function

#### [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.

#### [regex/find-all](#)

Returns all regex matches.

[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

[regex/pattern](#)

Returns an instance of java.util.regex.Pattern.

top

## regex/matches?

```
(regex/matches? matcher)
```

Attempts to match the entire region against 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/matches? m))
=> true

(let [m (regex/matcher "[0-9]+" "value: 100")]
  (regex/matches? m))
=> false

```

## SEE ALSO

[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().

top

## regex/pattern



```
(regex/pattern s)
```

Returns an instance of `java.util.regex.Pattern`.

JavaDoc: [Pattern](#)

```
(regex/pattern "[0-9]+")  
=> [0-9]+
```

## SEE ALSO

[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.

[regex/find-all](#)

Returns all regex matches.

[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

[regex/matcher](#)

Returns an instance of `java.util.regex.Matcher`.

[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 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)
```

[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.

[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.

[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))  
=> (3 1 0 0 1)
```

```
;; compare with repeat, which only calls the 'rand-long'  
;; function once, repeating the value five times.  
(repeat 5 (rand-long 11))  
=> (7 7 7 7 7)
```

### 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.

[top](#)

## repl/info

(repl/info)

Returns information on the 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/term-rows](#)

Returns number of rows in the REPL terminal.

[repl/term-cols](#)

Returns number of columns in the REPL terminal.

[top](#)

## repl/term-cols

(repl/term-cols)

Returns number of columns in the REPL terminal.

### SEE ALSO

[repl/term-rows](#)

Returns number of rows in the REPL terminal.

[repl/info](#)

Returns information on the REPL.

[top](#)

## repl/term-rows

(repl/term-rows)

Returns number of rows in the REPL terminal.

### SEE ALSO

[repl/term-cols](#)

Returns number of columns in the REPL terminal.

[repl/info](#)

Returns information on the REPL.

[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}
```

[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

[top](#)

## resolve

```
(resolve symbol)
```

Resolves a symbol.

```
(resolve '+)
=> function + {visibility :public, ns "", native true}
```

```
(resolve 'y)
=> nil
```

```
(resolve (symbol "+"))
=> function + {visibility :public, ns "", native true}
```

```
((-> "first" symbol resolve) [1 2 3])
=> 1
```

[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.

[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.

[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

[top](#)

## 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.

[top](#)

## 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.

[top](#)

## 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.

[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.

[top](#)

## 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.

top

## sandboxed?

```
(sandboxed?)
```

Returns true if there is a sandbox other than `:AcceptAllInterceptor` otherwise false.

```
(sandboxed?)
```

```
=> false
```

## SEE ALSO

[sandbox-type](#)

Returns the sandbox type.

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)`, `(future-cancel f)`, and `(future-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)  
  (future-cancel s))
```

## SEE ALSO

[schedule-delay](#)

Creates and executes a one-shot action that becomes enabled after the given delay.

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)`, `(future-cancel f)`, and `(future-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.

[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
```

[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 ...

[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?

[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.

[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.

[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.

[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

[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.

## 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.

## 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.

## 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.

[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.

[top](#)

## seq

```
(seq coll)
```

Returns a seq 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 [1 2 3])
=> (1 2 3)

(seq '(1 2 3))
=> (1 2 3)

(seq {:a 1 :b 2})
=> ([:a 1] [:b 2])

(seq "abcd")
=> (#\a #\b #\c #\d)
```

## 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
```

## 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}
```

## 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.

[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 occured: "
      (: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.

top

## set?

(set? obj)

Returns true if obj is a set

```
(set? (set 1))  
=> true
```

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

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 `(with-sh-throw (sh "ls" "-l"))` instead.

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 &"))

;; 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`.

top

## sh/open

(`sh/open`)

Opens a *file* or an *URL* with the associated platform specific application.

```
(sh/open "sample.pdf")
```

```
(sh/open "https://github.com/jlangch/venice")
```

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.

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.

[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 ...

[top](#)

## shell/diff

```
(diff file1 file2)
```

Compare two files and print the differences.

```
(diff "/tmp/x.txt" "/tmp/y.txt")
```

## 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.

## 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.

## 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.

[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.

[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.

## 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.

## 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 ...



## shell/process-handle?

(process-handle? p)

Returns true if p is a process handle (:java.lang.ProcessHandle).

Requires Java 9+.

[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.

[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.

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.

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.

[top](#)

## shuffle

```
(shuffle coll)
```

Returns a collection of the items in coll in random order.

```
(shuffle '(1 2 3 4 5 6))  
=> (4 3 1 2 5 6)
```

```
(shuffle [1 2 3 4 5 6])  
=> [3 1 5 2 4 6]
```

```
(shuffle "abcdef")  
=> (#\d #\f #\b #\a #\e #\c)
```

[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.

[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.

[top](#)

## shutdown-hook

```
(shutdown-hook f)
```

Registers the function `f` as JVM shutdown hook.

```
(shutdown-hook (fn [] (println "shutdown")))
=> nil
```

[top](#)

## 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`.

```
(sleep 30)
=> nil

(sleep 30 :milliseconds)
=> nil

(sleep 5 :seconds)
=> nil
```

[top](#)

## 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
```

[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.

[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.

[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

[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])
```

[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.

```
(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]]

; 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 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}]
```

[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]
```

[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}
```

[top](#)

## sorted-map?

```
(sorted-map? obj)
```

Returns true if `obj` is a sorted map

```
(sorted-map? (sorted-map :a 1 :b 2))  
=> true
```

[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]}"
```

[top](#)

## sorted-set?

```
(sorted-set? obj)
```

Returns true if obj is a sorted-set

```
(sorted-set? (sorted-set 1))
=> true
```

[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) ()]
```

[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)]
```

[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

[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

[top](#)

## stack

(stack)

Creates a new mutable threadsafe stack.

```
(let [s (stack)]
  (push! s 4)
  (push! s 3)
  (pop! s)
  s)
=> (4)
```

## SEE ALSO

### [peek](#)

For a list, same as first, for a vector, same as last, for a stack the top element

### [pop!](#)

Pops an item from a stack.

### [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

[top](#)

## stack?

```
(stack? coll)
```

Returns true if coll is a stack

```
(stack? (stack))
=> true
```

[top](#)

## stacktrace

```
(stacktrace ex)
```

Returns the stacktrace of a java exception

```
(println (stacktrace (. :VncException :new (str "test"))))
```

[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 +)
=> "function + {visibility :public, ns \"\", native true}"

(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 ...

[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

## str/butlast

```
(str/butlast s)
```

Returns a possibly empty string of the characters without the last.

```
(str/butlast "abcdef")  
=> "abcde"
```

## 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/char?

```
(str/char? s)
```

Returns true if `s` is a char or a single char string.

```
(str/char? "x")  
=> true
```

```
(str/char? #\x)  
=> true
```

## 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"
```

[top](#)

## str/contains?

```
(str/contains? s substr)
```

True if s contains with substr.

```
(str/contains? "abc" "ab")  
=> true
```

```
(str/contains? "abc" #\b)  
=> true
```

[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)
```

[top](#)

## str/decode-base64

```
(str/decode-base64 s)
```

Base64 decode.

```
(str/decode-base64 (str/encode-base64 (bytebuf [0 1 2 3 4 5 6])))  
=> [0 1 2 3 4 5 6]
```

[top](#)

## str/decode-url

```
(str/decode-url s)
```

URL decode.

```
(str/decode-url "The+string+%C3%BC%40foo-bar")  
=> "The string ü@foo-bar"
```

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.

top

## str/double-quote

(str/double-quote str)

Double quotes a string.

```
(str/double-quote "abc")  
=> "\"abc\""
```

```
(str/double-quote "")  
=> "\"\""
```

top

## str/double-quoted?

(str/double-quoted? str)

Returns true if the string is double quoted.

```
(str/double-quoted? "\"abc\"")  
=> true
```

[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
```

[top](#)

## str/encode-base64

```
(str/encode-base64 data)
```

Base64 encode.

```
(str/encode-base64 (bytebuf [0 1 2 3 4 5 6]))  
=> "AAECAwQFBg=="
```

[top](#)

## str/encode-url

```
(str/encode-url s)
```

URL encode.

```
(str/encode-url "The string ü@foo-bar")  
=> "The+string+%C3%BC%40foo-bar"
```

[top](#)

## str/ends-with?

```
(str/ends-with? s substr)
```

True if s ends with substr.

```
(str/ends-with? "abc" "bc")  
=> true
```



[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
```

[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; "
```

[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; "
```

[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"
```

[top](#)

## 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" "DE" ] "value: %,d" 2345000)
=> "value: 2.345.000"
```

[top](#)

## 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"
```

[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]
```

[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
```

[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
```

[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-2.56M"
```

[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
```

[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
```

[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
```

```
(str/levenshtein "Tier" "tor")  
=> 3
```

[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
```

[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."
```

[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"
```

[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
```

[top](#)

## str/nfirst

```
(str/nfirst s n)
```

Returns a string of the n first characters of s.

```
(str/nfirst "abcdef" 2)
=> "ab"
```

[top](#)

## str/nlast

```
(str/nlast s n)
```

Returns a string of the n last characters of s.

```
(str/nlast "abcdef" 2)
=> "ef"
```

[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

[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}
```

[top](#)

## str/quote

```
(str/quote str q)
(str/quote str start end)
```

Quotes a string.

```
(str/quote "abc" "-")
=> "-abc-
```

```
(str/quote "abc" "<" ">")
=> "<abc>"
```

[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
```

[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"
```

[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"
```

[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"
```

[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"
```

[top](#)

## str/rest

```
(str/rest s)
```

Returns a possibly empty string of the characters after the first.

```
(str/rest "abcdef")
=> "bcdef"
```

[top](#)

## str/reverse

```
(str/reverse s)
```

Reverses a string

```
(str/reverse "abcdef")
=> "fedcba"
```

## str/split

```
(str/split s regex)
```

Splits string on a regular expression.

```
(str/split "abc,def,ghi" ",")  
=> ("abc" "def" "ghi")  
  
(str/split "abc , def , ghi" "[ *],[ *]")  
=> ("abc" "def" "ghi")  
  
(str/split "abc,def,ghi" "((?<=,)|(?=,))")  
=> ("abc" ", " "def" ", " "ghi")  
  
(str/split nil ",")  
=> ()
```

## str/split-lines

```
(str/split-lines s)
```

Splits s into lines.

```
(str/split-lines "line1  
line2  
line3")  
=> ("line1" "line2" "line3")
```

## str/starts-with?

```
(str/starts-with? s substr)
```

True if s starts with substr.

```
(str/starts-with? "abc" "ab")  
=> true
```

## 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"
```

[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"
```

[top](#)

## 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"
```

[top](#)

## 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"
```

[top](#)

## 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"
```

[top](#)

## str/trim

```
(str/trim s)
```

Trims leading and trailing whitespaces from s.

```
(str/trim " abc ")
=> "abc"
```

### 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.

[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

[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

[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](#)

Trims leading and trailing whitespaces from s.

[str/trim-left](#)

Trims leading whitespaces from s.

[str/trim-right](#)

Trims trailing whitespaces from s.

[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..."
```

[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"
```

## 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
```

## 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
```

## 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
```



## 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]
```

## string?

```
(string? x)
```

Returns true if x is a string

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

(string? 1)
=> false

(string? nil)
=> false
```

## 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)

[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

[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).

[top](#)

## supers

(supers class)

Returns the immediate and indirect superclasses and interfaces of class, if any.

```
(supers :java.util.ArrayList)
=> (:java.util.AbstractList :java.util.AbstractCollection :java.util.List :java.util.Collection :java.lang.
Iterable)
```

[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

[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

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)
```

```
(supertype (. :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

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

[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

[top](#)

## 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
```

[top](#)

## symbol?

```
(symbol? x)
```

Returns true if x is a symbol

```
(symbol? (symbol "a"))
=> true

(symbol? 'a)
=> true

(symbol? nil)
=> false

(symbol? :a)
=> false
```

[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/bash"

(system-env :FOO "test")
=> "test"

(system-env "SHELL")
=> "/bin/bash"
```

## 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.

[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)
=> nil
```

[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.

## 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
```

## 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-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]
```

[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]
```

[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
```

### 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 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.

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 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 ...

#### [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.

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 the current stage's result ...

### [accept-either](#)

Returns a new promise that, when either this or the other given promise completess 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.

[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 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.

[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 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.

[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
```

[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.

[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.

[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 ...

[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, ...

[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.

[top](#)

## thread-local-clear

```
(thread-local-clear)
```

Removes all thread local vars

```
(thread-local-clear)
```

```
=> function thread-local-clear {visibility :public, ns "", native true}
```

## SEE ALSO

### [thread-local](#)

Creates a new thread-local accessor

### [dissoc!](#)

Dissociates keys from a mutable map, returns the map

[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}
```

### 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

[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

[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 ...

[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 ...

[top](#)

## time

```
(time expr)
```

Evaluates expr and prints the time it took. Returns the value of expr.

```

(time (+ 100 200))
Elapsed time: 6.67µs
=> 300

```

## SEE ALSO

[perf](#)

Performance test with the given expression.

[prof](#)

Controls the code profiling. See the companion functions/macros 'dorun' and 'perf'. The perf macro is built on prof and dorun and provides ...

[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
```

[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
```

[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)
=> Wed Jun 01 09:22:06 CEST 2022
```

[top](#)

## time/date?

```
(time/date? date)
```

Returns true if date is a 'java.util.Date' else false

```
(time/date? (time/date))  
=> true
```

[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))  
=> 1
```

```
(time/day-of-month (time/local-date-time))  
=> 1
```

```
(time/day-of-month (time/zoned-date-time))  
=> 1
```

[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))  
=> :WEDNESDAY
```

```
(time/day-of-week (time/local-date-time))  
=> :WEDNESDAY
```

```
(time/day-of-week (time/zoned-date-time))  
=> :WEDNESDAY
```

[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))  
=> 152
```

```
(time/day-of-year (time/local-date-time))  
=> 152
```

```
(time/day-of-year (time/zoned-date-time))  
=> 152
```

## 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
```

## 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))  
=> 2022-06-01
```

```
(time/first-day-of-month (time/local-date-time))  
=> 2022-06-01
```

```
(time/first-day-of-month (time/zoned-date-time))  
=> 2022-06-01
```

## time/format

```
(time/format date format locale?)  
(time/format date formatter locale?)
```

Formats a date with a format

```
(time/format (time/local-date) "dd-MM-yyyy")  
=> "01-06-2022"
```

```
(time/format (time/zoned-date-time) "yyyy-MM-dd'T'HH:mm:ss.SSSz")  
=> "2022-06-01T09:22:08.171CEST"
```

```
(time/format (time/zoned-date-time) :ISO_OFFSET_DATE_TIME)  
=> "2022-06-01T09:22:08.188+02:00"
```

```
(time/format (time/zoned-date-time) (time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz"))  
=> "2022-06-01T09:22:08.205CEST"
```

```
(time/format (time/zoned-date-time) (time/formatter :ISO_OFFSET_DATE_TIME))  
=> "2022-06-01T09:22:08.223+02:00"
```

## time/formatter

(time/formatter format locale?)

Creates a formatter

```
(time/formatter "dd-MM-yyyy")
=> Value(DayOfMonth,2) '- 'Value(MonthOfYear,2) '- 'Value(YearOfEra,4,19,EXCEEDS_PAD)

(time/formatter "dd-MM-yyyy" :en_EN)
=> Value(DayOfMonth,2) '- 'Value(MonthOfYear,2) '- 'Value(YearOfEra,4,19,EXCEEDS_PAD)

(time/formatter "dd-MM-yyyy" "en_EN")
=> Value(DayOfMonth,2) '- 'Value(MonthOfYear,2) '- 'Value(YearOfEra,4,19,EXCEEDS_PAD)

(time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> Value(YearOfEra,4,19,EXCEEDS_PAD) '- 'Value(MonthOfYear,2) '- 'Value(DayOfMonth,2) 'T'Value(HourOfDay,2) ':'Value(MinuteOfHour,2) ':'Value(SecondOfMinute,2) '.'Fraction(NanoOfSecond,3,3)ZoneText(SHORT)

(time/formatter :ISO_OFFSET_DATE_TIME)
=> ParseCaseSensitive(false)(ParseCaseSensitive(false)(Value(Year,4,10,EXCEEDS_PAD) '- 'Value(MonthOfYear,2) '- 'Value(DayOfMonth,2))'T'(Value(HourOfDay,2) ':'Value(MinuteOfHour,2) '['Value(SecondOfMinute,2) [Fraction(NanoOfSecond,0,9,DecimalPoint)]])Offset(+HH:MM:ss,'Z')
```

## 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))
=> 9

(time/hour (time/zoned-date-time))
=> 9
```

## 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))
=> 2022-06-30
```

```
(time/last-day-of-month (time/local-date-time))  
=> 2022-06-30
```

```
(time/last-day-of-month (time/zoned-date-time))  
=> 2022-06-30
```

[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
```

[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
```

[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
```

[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
```

[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)
```

```
=> 2022-06-01
```

```
(time/local-date 2018 8 1)
```

```
=> 2018-08-01
```

```
(time/local-date "2018-08-01")
```

```
=> 2018-08-01
```

```
(time/local-date 1375315200000)
```

```
=> 2013-08-01
```

```
(time/local-date (. :java.util.Date :new))
```

```
=> 2022-06-01
```

[top](#)

## time/local-date-parse



```
(time/local-date-parse str format locale?)
```

Parses a local-date.

```
(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
```

[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)  
=> 2022-06-01T09:22:06.869  
  
(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 1375315200000)  
=> 2013-08-01T02:00  
  
(time/local-date-time (. :java.util.Date :new))  
=> 2022-06-01T09:22:06.979
```

[top](#)

## time/local-date-time-parse

```
(time/local-date-time-parse str format locale?)
```

Parses a local-date-time.

```
(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 14:20:01.000" "yyyy-MM-dd HH:mm:ss.SSS")
=> 2018-08-01T14:20:01
```

[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
```

[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
```

[top](#)

## time/minus

```
(time/minus date unit n)
```

Subtracts the n units from the date. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(time/minus (time/local-date) :days 2)
=> 2022-05-30
```

```
(time/minus (time/local-date-time) :days 2)
=> 2022-05-30T09:22:08.725
```

```
(time/minus (time/zoned-date-time) :days 2)
=> 2022-05-30T09:22:08.742+02:00[Europe/Zurich]
```

[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))
=> 22

(time/minute (time/zoned-date-time))
=> 22
```

[top](#)

## 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
```

[top](#)

## 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
```

[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
```

[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
```

[top](#)

## time/plus

(time/plus date unit n)

Adds the n units to the date. Units: {years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(time/plus (time/local-date) :days 2)
=> 2022-06-03

(time/plus (time/local-date-time) :days 2)
=> 2022-06-03T09:22:08.672

(time/plus (time/zoned-date-time) :days 2)
=> 2022-06-03T09:22:08.689+02:00[Europe/Zurich]
```

[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))
```

```
=> 7
```

```
(time/second (time/zoned-date-time))
```

```
=> 7
```

[top](#)

## time/to-millis

```
(time/to-millis date)
```

Converts the passed date to milliseconds since epoch

```
(time/to-millis (time/local-date))
```

```
=> 1654034400000
```

[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)
```

```
=> 2022-06-01T22:00:15.333
```

```
(time/with-time (time/local-date-time) 22 00 15 333)
```

```
=> 2022-06-01T22:00:15.333
```

```
(time/with-time (time/zoned-date-time) 22 00 15 333)
```

```
=> 2022-06-01T22:00:15.333+02:00[Europe/Zurich]
```

[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
```

[top](#)

## time/year

(time/year date)

Returns the year of the date

```
(time/year (time/local-date))
=> 2022

(time/year (time/local-date-time))
=> 2022

(time/year (time/zoned-date-time))
=> 2022
```

[top](#)

## time/zone

(time/zone date)

Returns the zone of the date

```
(time/zone (time/zoned-date-time))
=> "Europe/Zurich"
```

[top](#)

## 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"]])
```

[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
```

[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)
=> 2022-06-01T09:22:07.047+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 1375315200000)
=> 2013-08-01T02:00+02:00[Europe/Zurich]

(time/zoned-date-time (. :java.util.Date :new))
=> 2022-06-01T09:22:07.153+02:00[Europe/Zurich]

(time/zoned-date-time "UTC")
=> 2022-06-01T07:22:07.170Z[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" 1375315200000)
=> 2013-08-01T00:00Z[UTC]

(time/zoned-date-time "UTC" (. :java.util.Date :new))
=> 2022-06-01T07:22:07.273Z[UTC]

```

top

## time/zoned-date-time-parse

```
(time/zoned-date-time-parse str format locale?)
```

Parses a zoned-date-time.

```

(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_OFFSET_DATE_TIME)
=> 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

```

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

```



## 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 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.

[top](#)

## total-memory

```
(total-memory)
```

Returns the total amount of memory available to the Java VM.

```
(total-memory)
=> "981.0MB"
```

### SEE ALSO

#### [used-memory](#)

Returns the currently used memory by the Java VM.

[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
  (-> 5
    (+ 3)
    trace/tee
    (/ 2)
    trace/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

[top](#)

## trace/tee->

```
(tee-> x f!)
```

Allows to branch off values passed through the forms of a `->` macro

```
(do
  (-> 5
    (+ 3)
    (trace/tee-> #(println "trace:" %))
    (/ 2)
    (trace/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.

[top](#)

## trace/tee->>

```
(tee->> x f!)
```

Allows to branch off values passed through the form of a `->>` macro

```
(do
  (->> 5
    (+ 3)
    (trace/tee->> #(println "trace:" %))
    (/ 32)
    (trace/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.

[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 ...

[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.

[top](#)

## trace/trace-var

```
(trace-var v)
```

Traces the var

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

```

(trace/trace-var +)
(+ 1 2))
TRACE t67099: (core/+ 1 2)
TRACE t67099: | => 3
=> 3

(do
  (load-module :trace)

  (defn foo [x] (+ x 2))
  (defn bar [x] (foo x))

  (trace/trace-var +)
  (trace/trace-var foo)
  (trace/trace-var bar)

  (bar 5))
TRACE t67135: (user/bar 5)
TRACE t67136: | (user/foo 5)
TRACE t67137: | | (core/+ 5 2)
TRACE t67137: | | | => 7
TRACE t67136: | | => 7
TRACE t67135: | => 7
=> 7

```

## 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 ...

[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

## 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/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/untrace-var

```
(untrace-var v)
```

Untraces the var

```
(trace/untrace-var +)  
=> nil
```

### SEE ALSO

[trace/trace-var](#)

Traces the var

[trace/traced?](#)

Returns true if the given var is currently traced, false otherwise

## 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))))))

(transduce (factorial 20)))
=> 2432902008176640000N
```

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.

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]

(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]
```

[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.

[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.

```
(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

[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.

```

(do
  (import :java.io.FileInputStream)
  (let [file (io/temp-file "test-", ".txt")]
    (io/spit file "123456789" :append true)
    (try-with [is (. :FileInputStream :new 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

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

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). For list, vectors and ordered maps the values are ...

### [disj](#)

Returns a new set with the x, xs removed.

[top](#)

## update

```
(update m k f)
```

Updates a value in an associative structure, where k is a key and f is a function that will take the old value return the new value. Returns a new structure.

```
(update [] 0 (fn [x] 5))  
=> [5]
```

```
(update [0 1 2] 0 (fn [x] 5))  
=> [5 1 2]
```

```
(update [0 1 2] 0 (fn [x] (+ x 1)))  
=> [1 1 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] 5))  
=> {:a 5 :b 1}
```

### 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)

[top](#)

## update!

```
(update! m k f)
```

Updates a value in a mutable map, where k is a key and f is a function that will take the old value return the new value.

```
(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 1 2 3) 0 (fn [x] 10))
=> [10 2 3]
```

## SEE ALSO

### [assoc!](#)

Associates key/vals with a mutable map, returns the map

### [dissoc!](#)

Dissociates keys from a mutable map, returns the map

[top](#)

## update-in

```
(update-in [m ks f & args])
```

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 args and return the new value, and returns a new nested structure. If any levels do not exist, hash-maps will be created.

```
(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 3}
```

[top](#)

## used-memory

```
(used-memory)
```

Returns the currently used memory by the Java VM.

```
(used-memory)
=> "92.2MB"
```

## SEE ALSO

### [total-memory](#)

Returns the total amount of memory available to the Java VM.

[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.

[top](#)

## uuid

(uuid)

Generates a UUID.

```
(uuid)
=> "9902326a-3de5-4df5-bebc-11401f80eded"
```

[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.

[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 ...

[top](#)

## var-get

```
(var-get v)
```

Returns a var's value.

```
(var-get +)  
=> function + {visibility :public, ns "", native true}
```

```
(var-get '+)  
=> function + {visibility :public, ns "", native true}
```

```
(var-get (symbol "+"))  
=> function + {visibility :public, ns "", native true}
```

```
((var-get +) 1 2)  
=> 3
```

```
(do  
  (def x 10)  
  (var-get 'x))  
=> 10
```

### SEE ALSO

[var-ns](#)

Returns the namespace of the var's symbol

[var-name](#)

Returns the 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

[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-ns](#)

Returns the namespace of the var's symbol

#### [var-name](#)

Returns the name 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

[top](#)

## var-local?

```
(var-local? v)
```



Returns true if the var is local else false

```
(var-local? +)
=> false

(var-local? '+)
=> false

(var-local? (symbol "+"))
=> false

(do
  (def x 10)
  (var-local? x))
=> false

(let [x 10]
  (var-local? x))
=> true
```

## SEE ALSO

[var-get](#)

Returns a var's value.

[var-ns](#)

Returns the namespace of the var's symbol

[var-name](#)

Returns the name 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

[top](#)

## var-name

```
(var-name v)
```

Returns the name of the var's symbol

```
(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

### [var-get](#)

Returns a var's value.

### [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

### [var-thread-local?](#)

Returns true if the var is thread-local else false

[top](#)

## var-ns

```
(var-ns v)
```

Returns the namespace of the var's symbol

```
(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.

### [var-get](#)

Returns a var's value.

### [var-name](#)

Returns the 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

[top](#)

## var-thread-local?

```
(var-thread-local? v)
```

Returns true if the var is thread-local else false

```
(binding [x 100]
  (var-local? x))
=> false
```

## SEE ALSO

### var-get

Returns a var's value.

### var-ns

Returns the namespace of the var's symbol

### var-name

Returns the 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

[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 :a 1))  
=> {:a 1 :line 21 :column 28 :file "example"}
```

[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"]
```

[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). For list, vectors and ordered maps the values are ...

[list\\*](#)

Creates a new list containing the items prepended to the rest, the last of which will be treated as a collection.

[top](#)

## vector?

```
(vector? obj)
```

Returns true if obj is a vector

```
(vector? (vector 1 2))
=> true
```

```
(vector? [1 2])
=> true
```

[top](#)

## version

```
(version)
```

Returns the Venice version.

```
(version)
=> "0.0.0"
```

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 ...

top

## volatile?

```
(volatile? x)
```

Returns true if x is a volatile, otherwise false

```
(do
  (def counter (volatile 0))
  (volatile? counter))
=> true
```

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.

[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 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 completeness 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.

[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.

[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.

[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
```

[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 ...

[top](#)

## with-meta

```
(with-meta obj m)
```

Returns a copy of the object `obj`, with a map `m` as its metadata.

[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 ...

[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.

[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.

[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.

[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"})
```

[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`, an `InputStream`, a `File`, or a string describing an URI.

Returns a tree of XML element maps with the keys `:tag`, `:attrs`, and `:content`.

[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"}
```

[top](#)

## 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"
```

[top](#)

## 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")
```

[top](#)

## zero?

(zero? x)

Returns true if x zero else false

```
(zero? 0)
=> true

(zero? 2)
=> false

(zero? (int 0))
=> 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

[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}
```

[top](#)

```
{}
```

Creates a hash map.

```
{:a 10 :b 20}
=> {:a 10 :b 20}
```