# Cheatsheet

#### Venice 0.3.0 Cheat Sheet

Primitives	
Numbers	
Literals	Nil: nil Long: 1500 Double: 3.569 Boolean: true, false BigDecimal: 6.897M String: "abcde"
Arithmetic	± <u>- * / mod inc</u> dec min max abs
Compare	== != < > <= >=
Test	nil? some? zero? pos? neg? even? odd? number? long? double? decimal?
Random	rand-long rand- double
BigDecimal	dec/add dec/sub dec/mul dec/div dec/scale
Strings	
Create	str str/format

Collections	
Collections	
Generic	count empty? not- empty? empty-to-nil into conj remove repeat range group- by
Tests	coll? list? vector? set? map? seq? hash-map? ordered- map? sorted-map? bytebuf?
Lists	
Create	() list
Access	first second nth last peek rest nfirst nlast
Modify	cons conj rest pop into concat interpose interleave flatten reduce reverse sort sort- by take take-while drop drop-while

Use	count empty-to-nil str /index-of str/last-index- of str/replace-first str /replace-last str /replace-all str/lower- case str/upper-case str/join str/subs str /split str/truncate
Regex	match match-not
Trim	str/trim str/trim-to-nil
Test	string? empty? str /starts-with? str/ends- with? str/contains?

### Other

Keywords	keyword? keyword literals: :a :xyz
Symbols	symbol? symbol
Boolean	boolean? boolean true? false?

# Functions

Create	fn
Call	apply memoize
Test	fn?
Exception	throw
Misc	<u>class</u> <u>eval</u>
Other	

Vectors	
Create	[] vector
Access	first second nth last peek rest nfirst nlast subvec
Modify	cons conj rest pop into concat interpose interleave flatten reduce reverse sort sort- by take take-while drop drop-while
Test	contains?
Sets	

Create	set
Test	contains?

### Maps

Create	{} hash-map ordered- map sorted-map
Access	find get keys vals key val
Modify	cons conj assoc into concat flatten reduce-kv
Test	contains?

	version uuid time-ms time-ns coalesce
Meta	meta with-meta vary-meta

#### Macros

Create	defmacro
Branch	and or not when when-not if-let
Loop	list-comp dotimes while
Call	<u>doto</u> -≥ ->>
Test	macro? cond
Assert	<u>assert</u>
Util	comment gensym time

#### Ю

to	prn println
to-str	<u>pr-str</u>
from	readline read-string
file-io	slurp spit io/file io /file? io/exists-file? io /exists-dir? io/list-files io/delete-file io/copy-file io/tmp-dir io/user-dir
load	load-file load-string

## Other Types

#### ByteBuffer

Misc count empty? notempty? bytebuf bytebuf? subbytebuf

#### Atoms

Create	atom
Test	atom?
Access	deref reset! swap! compare-and-set!

### **Special Forms**

Forms def if do let fn loop defmacro recur try

### Java Interoperabilty

General .

Constructor: (. classname:

new args)

Method call: (. object method

args)

### **Embedding in Java**

#### Eval

```
import org.venice.Venice;

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

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

#### Passing parameters

#### Precompiled

```
import org.venice.Venice;
import org.venice.PreCompiled;

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

    PreCompiled precompiled = venice.precompile("(+ 1 x)");

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

#### Java Interop

#### Sandbox

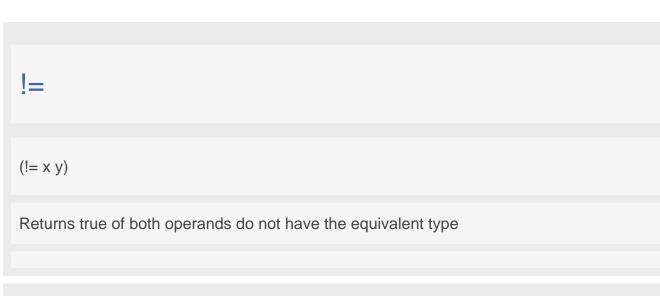
```
import org.venice.Venice;
import import org.venice.javainterop.*;
public class Example {
```

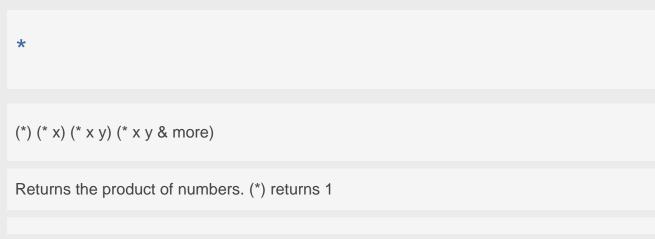
```
public static void main(String[] args) {
    Venice venice = new Venice();

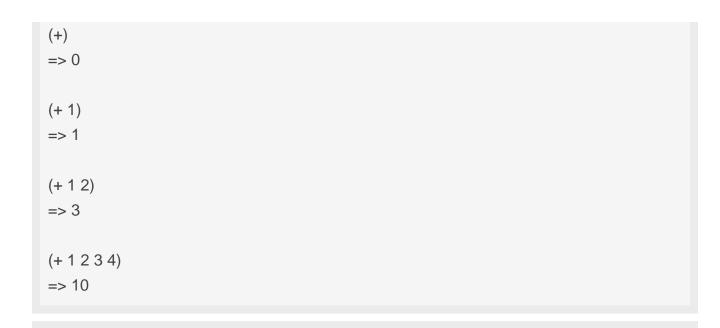
JavaInterceptor interceptor =
    new JavaSandboxInterceptor(
    WhiteList.create(
        "java.lang.Math:min",
        "java.lang.Math:max",
        "java.time.ZonedDateTime:*",
        "java.util.ArrayList:new"));

venice.eval("(. :java.lang.Math :min 20 30)"); // => OK
    venice.eval("(. (. :java.time.ZonedDateTime :now) :plusDays 5)"); // => OK
    venice.eval("(. :java.util.ArrayList :new)"); // => OK
    venice.eval("(. :java.lang.System :exit 0)"); // => Sandbox SecurityException
}
```

### **Function details**







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

->

->>

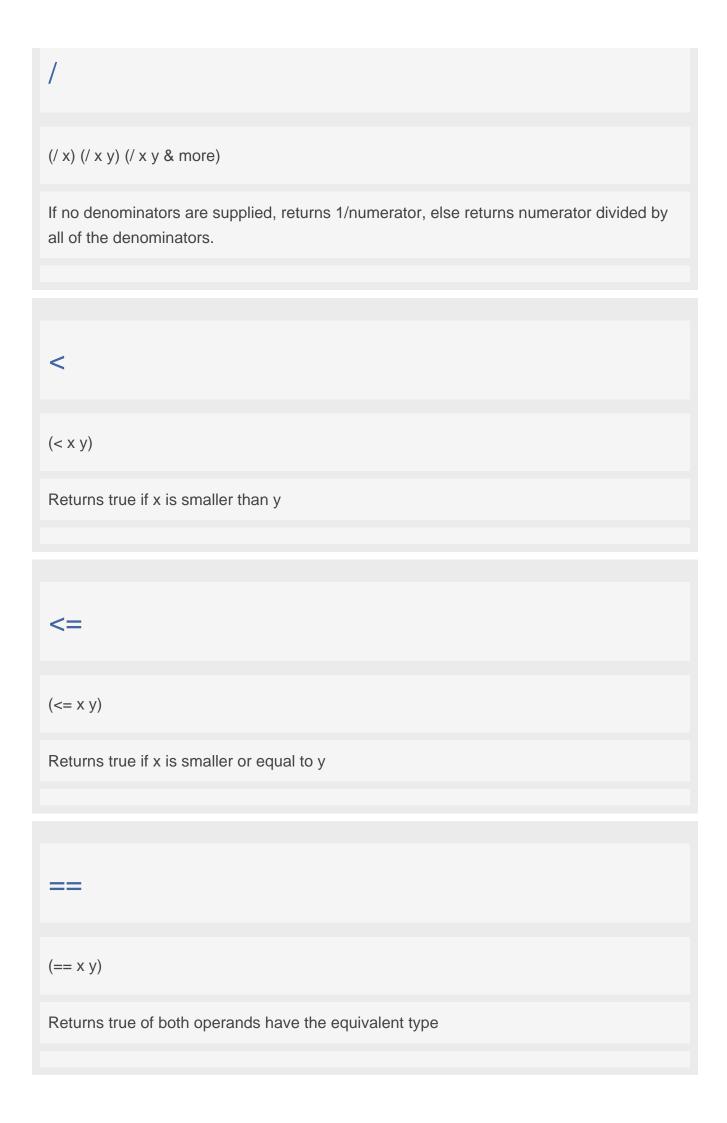
(-> x & forms)

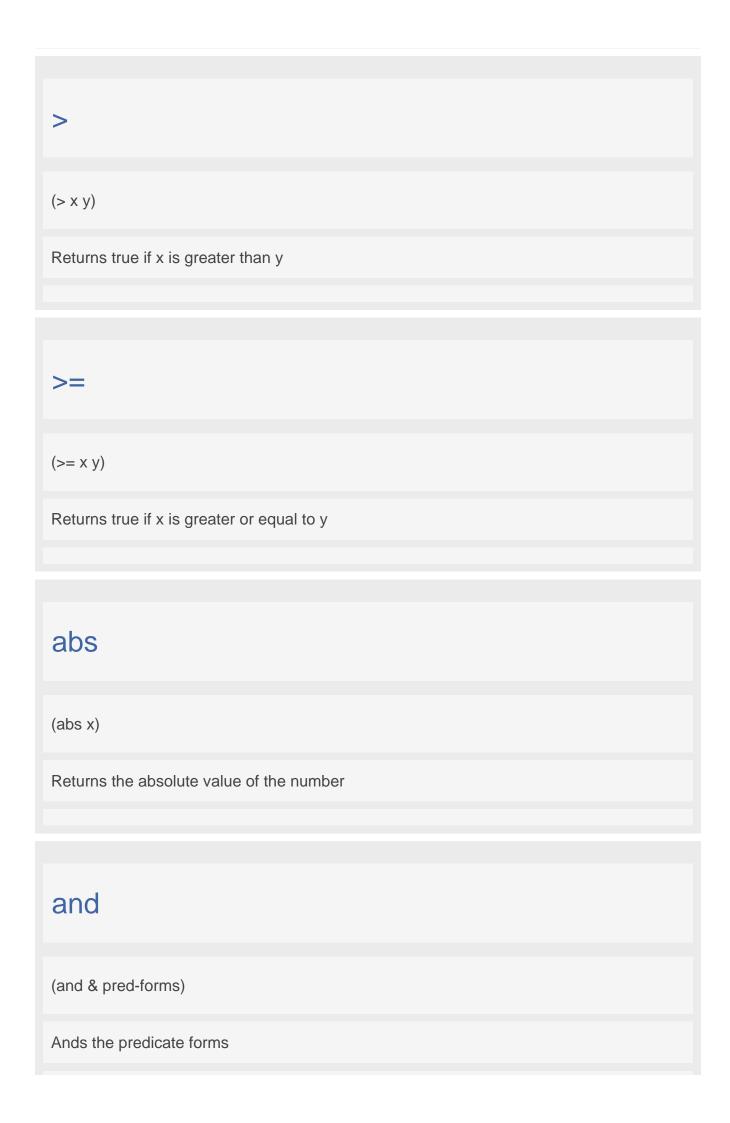
Threads the expr through the forms. Inserts x as the second item in the first form, making a list of it if it is not a list already. If there are more forms, inserts the first form as the second item in second form, etc.

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

Threads the expr 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.

```
(. classname :new args) (. object method args) (. classname :class) (. object :class)
Java interop. Calls a constructor or an object method. The function is sandboxed
(.: java.lang.Math:PI)
=> 3.141592653589793
(.: java.lang.Long:new 10)
=> 10
(. (. :java.lang.Long :new 10) :toString)
=> 10
(. :java.lang.Math :min 10 20)
=> 10
(.: java.lang.Math:class)
=> class java.lang.Math
(. "java.lang.Math" :class)
=> class java.lang.Math
(. (. :java.io.File :new "/temp") :class)
=> class java.io.File
```





# apply

(apply f args\* coll)

Applies f to all arguments composed of args and coll

#### assert

(assert expr) (assert expr message)

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

#### assoc

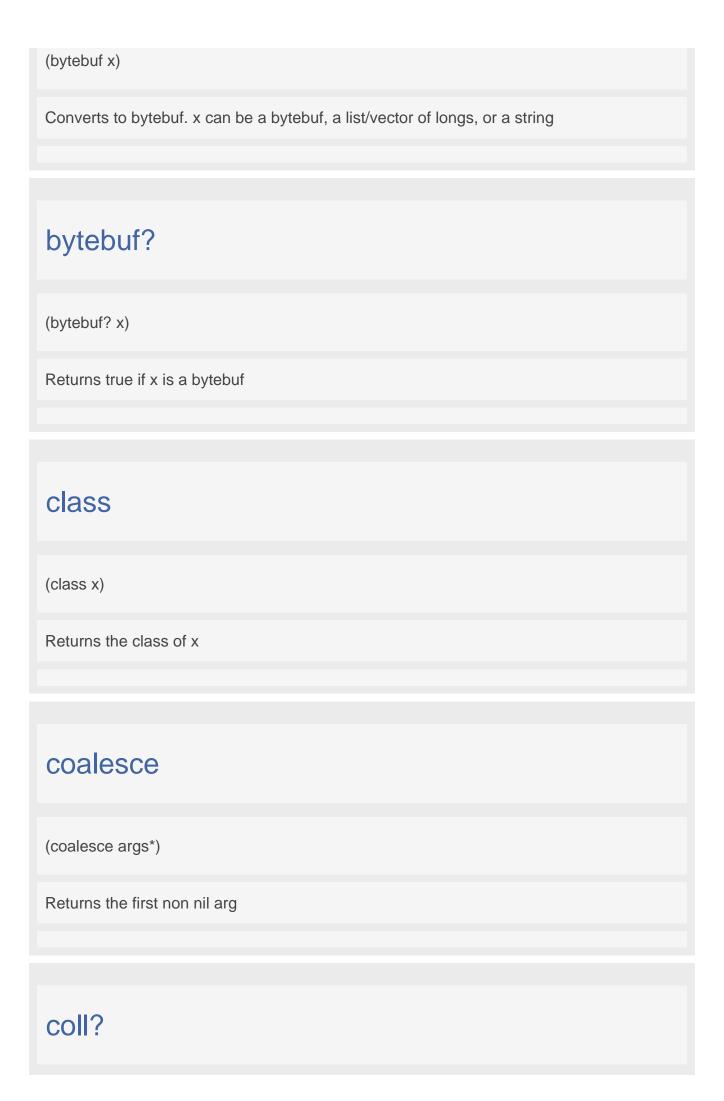
(assoc coll key val) (assoc coll key val & kvs)

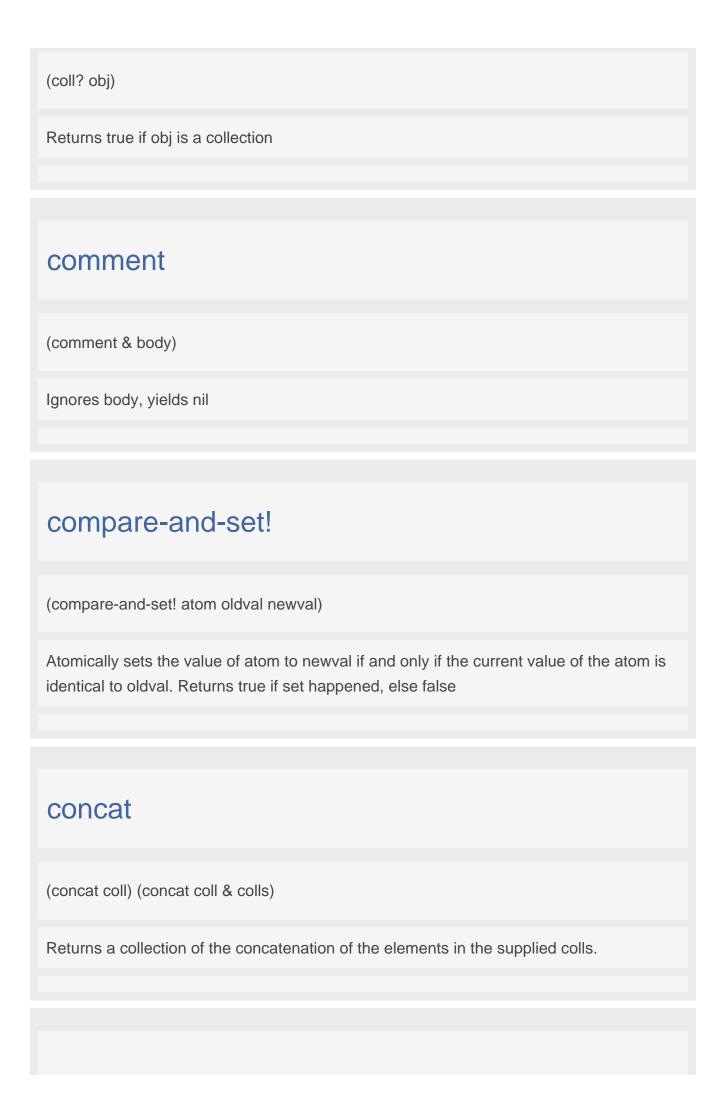
When applied to a map, returns a new map of the same type, that contains the mapping of key(s) to val(s). When applied to a vector, returns a new vector that contains val at index. Note - index must be <= (count vector).

#### atom

(atom x)

Creates an atom with the initial value x
atom?
atom:
(atom? x)
Returns true if x is an atom, otherwise false
boolean
(boolean x)
Converts to boolean. Everything except 'false' and 'nil' is true in boolean context.
boolean?
(boolean? n)
Returns true if n is a boolean
bytebuf





#### 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.

### conj

(conj coll x) (conj coll x & xs)

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item). The 'addition' may happen at different 'places' depending on the concrete type.

#### cons

(cons x coll)

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

#### contains?

(contains? coll key)

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

#### count

(count coll)

Returns the number of items in the collection. (count nil) returns 0. Also works on strings, and Java Collections

#### dec

(dec x)

Decrements the number x

#### 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, :UP)

#### 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, :UP)

#### 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, :UP)

#### dec/scale

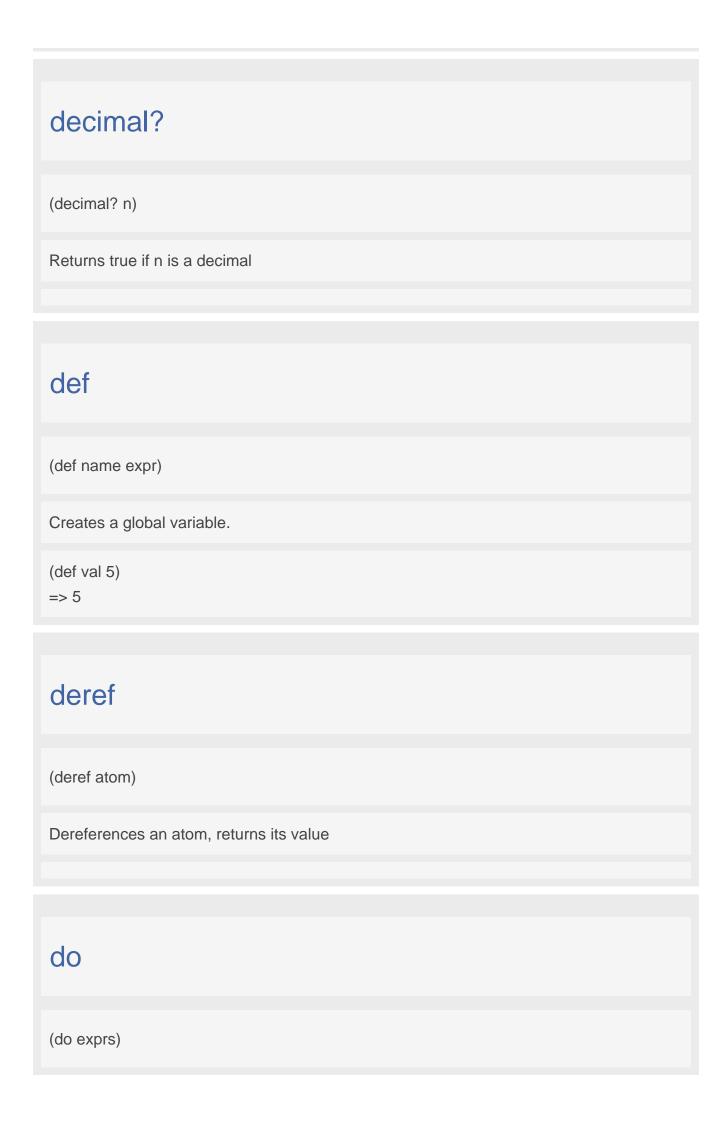
(dec/scale x scale rounding-mode)

Scales a decimal. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, : HALF\_DOWN, :HALF\_EVEN, :HALF\_UP, :UNNECESSARY, :UP)

#### 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, :UP)



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

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

#### dotimes

(dotimes bindings & body)

Repeatedly executes body with name bound to integers from 0 through n-1.

#### doto

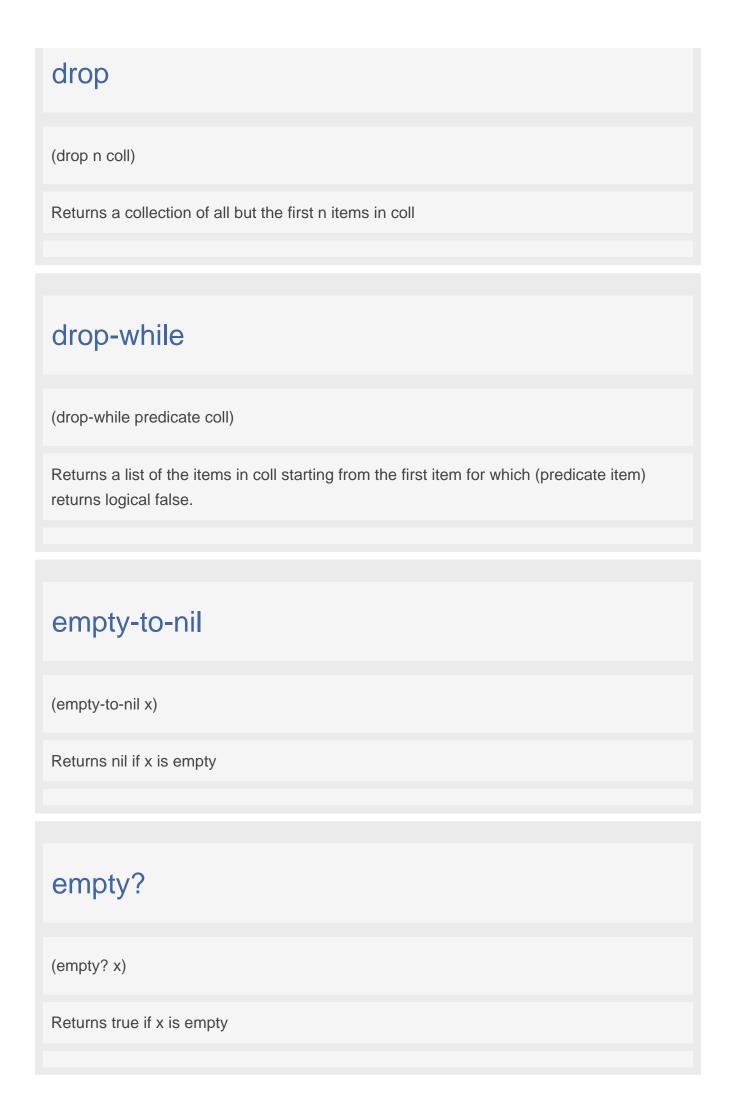
(doto x & forms)

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.

### double?

(double? n)

Returns true if n is a double



### 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))
```

#### even?

=> 6

(even? n)

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

### false?

(false? x)

Returns true if x is false, false otherwise

### 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)

=>

### first

(first coll)

Returns the first element of coll.

### 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.

### fn?

(fn? x)

Returns true if x is a function

### gensym

(gensym) (gensym prefix)

Generates a symbol.

### get

(get map key) (get map key not-found)

Returns the value mapped to key, not-found or nil if key not present.

### 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.

## hash-map

(hash-map & keyvals)

Creates a new hash map containing the items.

### hash-map?

(hash-map? obj)

Returns true if obj is a hash map

#### if

(if test true-expr false-expr)

Evaluates test.

(if (< 10 20) "yes" "no")

=> yes

### if-let

(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

#### inc

(inc x)

Increments the number x

#### interleave

(interleave c1 c2) (interleave c1 c2 & colls)

Returns a collection of the first item in each coll, then the second etc.

(interleave [:a :b :c] [1 2]) => (:a 1 :b 2)

### 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
```

### into

(into to-coll from-coll)

Returns a new coll consisting of to-coll with all of the items offrom-coll conjoined.

## io/copy-file

(io/copy input output)

Copies input to output. Returns nil or throws IOException. Input and output must be a java.io.File.

#### io/delete-file

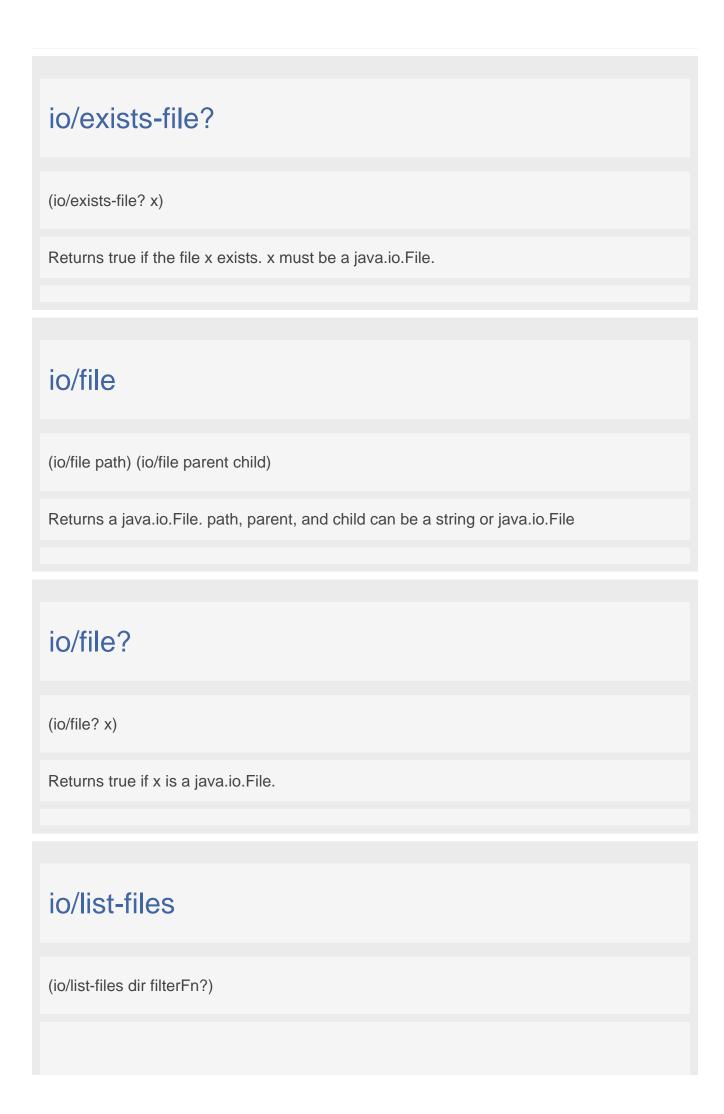
(io/delete-file x)

Deletes a file. x must be a java.io.File.

#### io/exists-dir?

(io/exists-dir? x)

Returns true if the file x exists and is a directory. x must be a java.io.File.



Lists files in a directory. dir must be a java.io.File. filterFn is an optional filter that filters the files found
io/tmp-dir
(io/tmp-dir)
Returns the tmp dir as a java.io.File.
io/user-dir
(io/user-dir)
Returns the user dir (current working dir) as a java.io.File.
key
(key e)
Returns the key of the map entry.
keys

(keys map)
Returns a collection of the map's keys.
keyword
(keyword name)
Returns a keyword from the given name
keyword?
(keyword? x)
(keyword? x)  Returns true if x is a keyword
Returns true if x is a keyword
Returns true if x is a keyword  last
Returns true if x is a keyword  last (last coll)

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

Evaluates the expressions and binds the values to symbols to new local context

(let [x 1] x))
=> 1
```

#### list

(list & items)

Creates a new list containing the items.

### list-comp

(list-comp seq-exprs body-expr)

List comprehension. Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a collection of evaluations of expr. Supported modifiers are: :when test.

```
(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 (list "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])

### list?

(list? obj)

Returns true if obj is a list

### load-file

(load-file name)

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

## load-string

(load-string s)

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

(load-string "(def x 1)")

=> 1

# long?

```
(long? n)

Returns true if n is a long
```

# loop

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

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

```
(loop [x 10]

(when (> x 1)

(println x)

(recur (- x 2))))
```

### macro?

(macro? x)

Returns true if x is a macro

# map?

(map? obj)

Returns true if obj is a map
match
(match s regex)
Returns true if the string s matches the regular expression regex
match-not
(match-not s regex)
Returns true if the string s does not match the regular expression regex
max
(max x) (max x y) (max x y & more)
Returns the greatest of the values
memoize
(memoize f)

Returns a memoized version of a referentially transparent function.

(do
 (def test (fn [a] (+ a 100)))
 (def test-memo (memoize test))
 (test-memo 1))
=> 101

#### meta

(meta obj)

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

#### min

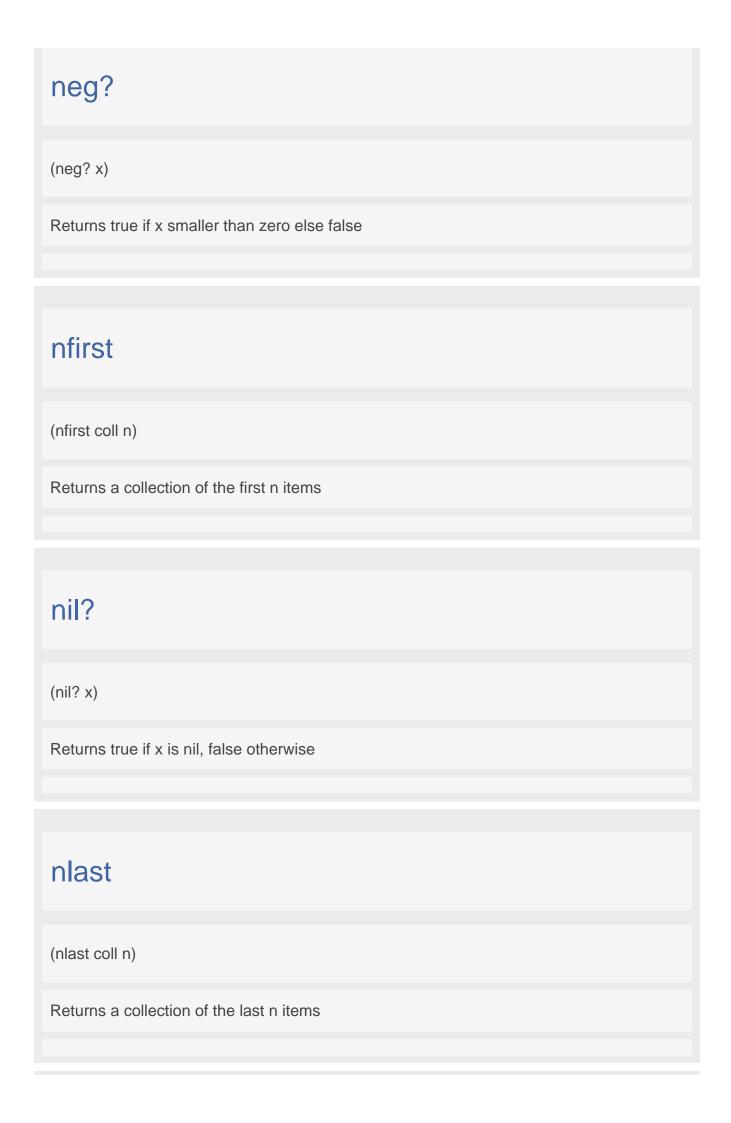
(min x) (min x y) (min x y & more)

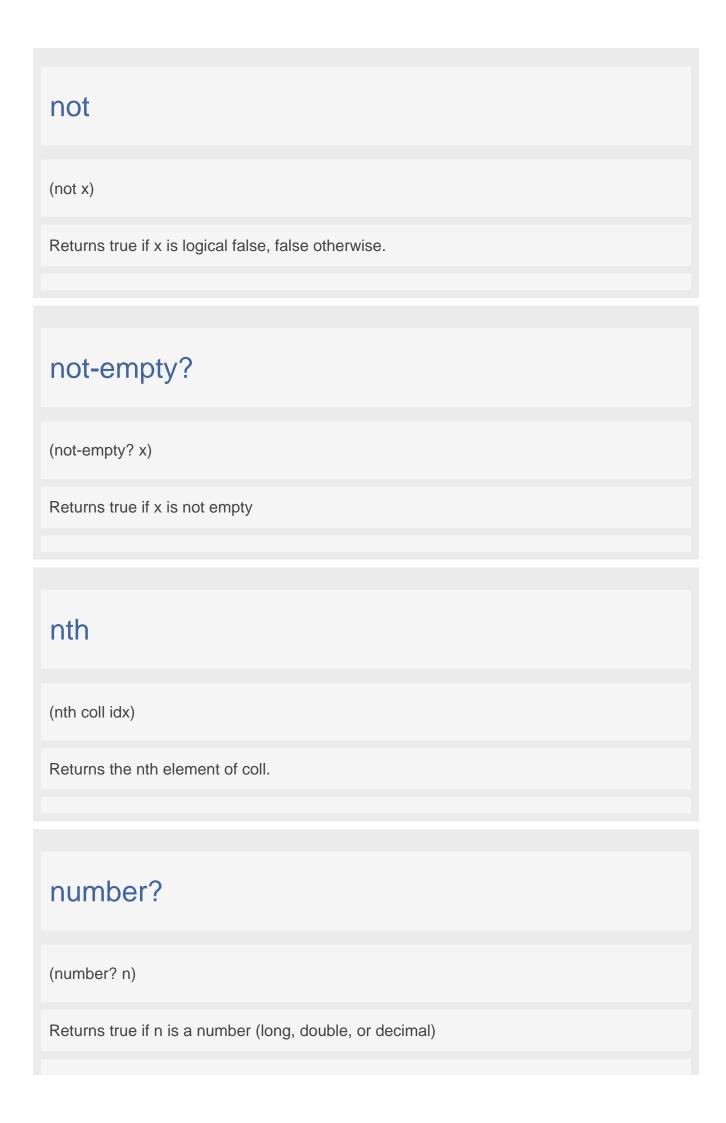
Returns the smallest of the values

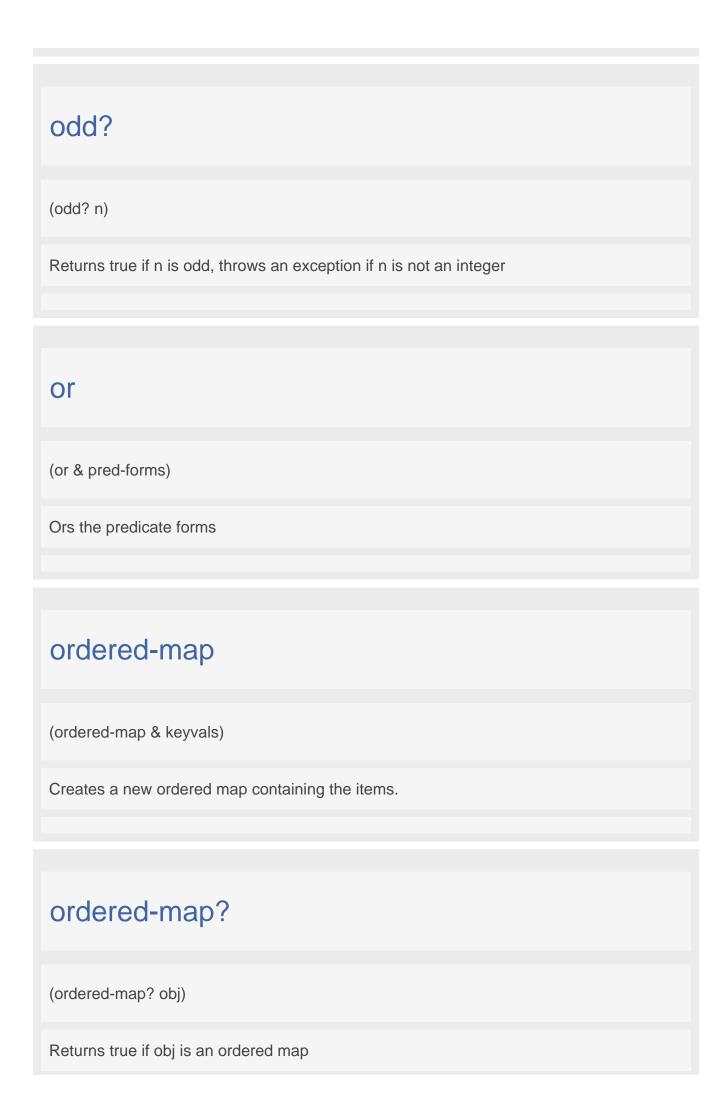
#### mod

(mod n d)

Modulus of n and d.









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 ' '.

# println

(println & xs)

Prints to stdout with a tailing linefeed, with no args, prints the empty string. With one arg x, prints x.toString(). With more than one arg, prints the concatenation of the str values of the args with delimiter ' '.The function is sandboxed.

#### prn

(prn & xs)

Prints to stdout, with no args, prints the empty string. With one arg x, prints x.toString(). With more than one arg, prints the concatenation of the str values of the args with delimiter ' '.The function is sandboxed.

#### rand-double

(rand-double) (rand-double max)

Without argument returns a double long between 0.0 and 1.0. Without argument max returns a random long between 0.0 and max.

# rand-long

(rand-long) (rand-long max)

Without argument returns a random long between 0 and MAX\_LONG. Without argument max returns a random long between 0 and max exclusive.

## 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.

# read-string

(read-string x)

Reads from x

# readline

(readline prompt)

Reads the next line from stdin. The function is sandboxed

#### recur

(recur expr\*)

Evaluates the exprs and rebinds the bindings of the recursion point to the values of the exprs.

#### 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-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.

#### remove

(remove predicate coll)

Returns a collection of the items in coll for which (predicate item) returns logical false.

# repeat

(repeat n x)

Returns a collection with the value x repeated n times

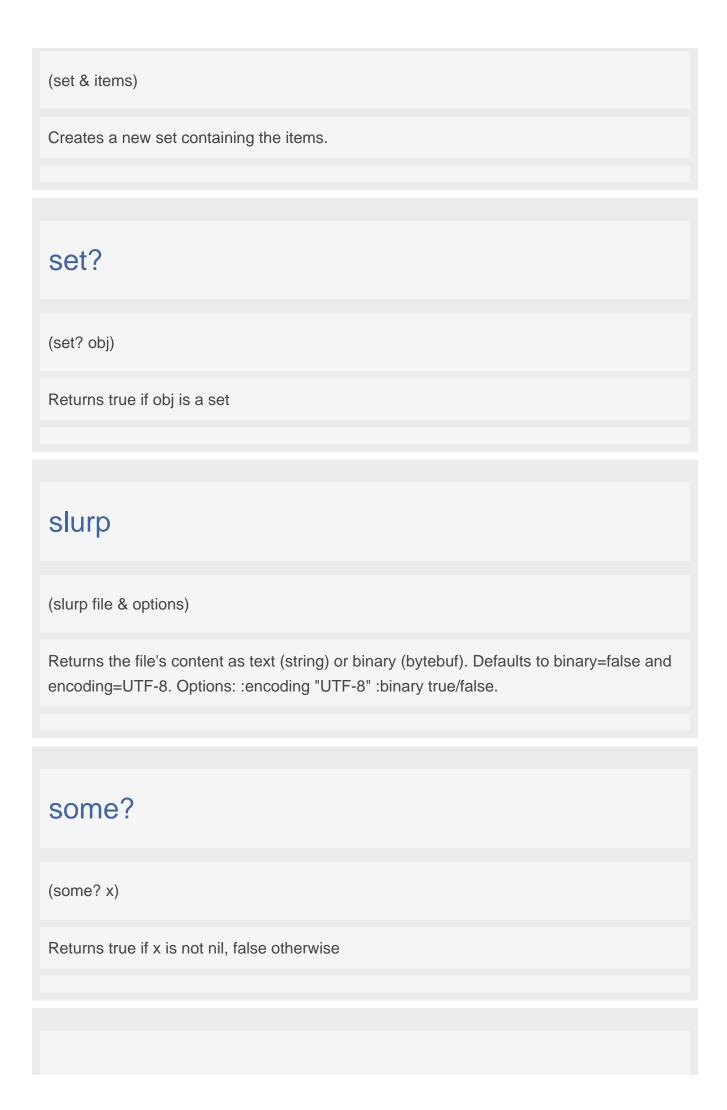
#### reset!

(reset! atom newval)

Sets the value of atom to newval without regard for the current value. Returns newval.

#### rest





#### sort

(sort coll) (sort compfn coll)

Returns a sorted list of the items in coll. If no compare function compfn is supplied, uses the natural compare. The compare function takes two arguments and returns -1, 0, or 1

# 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.

# sorted-map

(sorted-map & keyvals)

Creates a new sorted map containing the items.

# sorted-map?

(sorted-map? obj)

Returns true if obj is a sorted map

# spit

(spit f content & options)

Opens f, writes content, and then closes f. Defaults to append=true and encoding=UTF-8. Options: :append true/false, :encoding "UTF-8"

#### 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/contains?

(str/contains? s substr)

True if s contains with substr.

#### str/ends-with?

(str/ends-with? s substr)
True if s ends with substr.
str/format
(str/format s format args*)
Returns a formatted string using the specified format string and arguments.
str/index-of
(str/index-of s value) (str/index-of s value from-index)
(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.
Return index of value (string or char) in s, optionally searching forward from from-index.
Return index of value (string or char) in s, optionally searching forward from from-index.
Return index of value (string or char) in s, optionally searching forward from from-index.  Return nil if value not found.
Return index of value (string or char) in s, optionally searching forward from from-index. Return nil if value not found.  Str/join
Return index of value (string or char) in s, optionally searching forward from from-index.  Return nil if value not found.

# str/last-index-of

(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/lower-case

(str/lower-case s)

Converts s to lowercase

# str/replace-all

(str/replace-all s search replacement)

Replaces the all occurrances of search in s

# str/replace-first

(str/replace-first s search replacement)

Replaces the first occurrance of search in s

# str/replace-last (str/replace-last s search replacement) Replaces the last occurrance of search in s str/split (str/split s regex) Splits string on a regular expression. str/starts-with? (str/starts-with? s substr) True if s starts with substr. 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/trim

(str/trim s substr)

Trims leading and trailing spaces from s.

#### str/trim-to-nil

(str/trim-to-nil s substr)

Trims leading and trailing spaces from s. Returns nil if the rewsulting string is empry

#### str/truncate

(str/truncate s maxlen marker)

Truncates a string to the max length maxlen and adds the marker to the end if the string needs to be truncated

```
(str/truncate "abcdefghij" 20 "...")
```

=> abcdefghij

(str/truncate "abcdefghij" 9 "...")

=> abcdef...

(str/truncate "abcdefghij" 4 "...")
=> a...

# str/upper-case

(str/upper-case s)

Converts s to uppercase

# string?

(string? x)

Returns true if x is a string

# subbytebuf

(subbytebuf x start) (subbytebuf 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)

# 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)

# swap!

(swap! atom f & args)

Atomically swaps the value of 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 the value that was swapped in.

# symbol

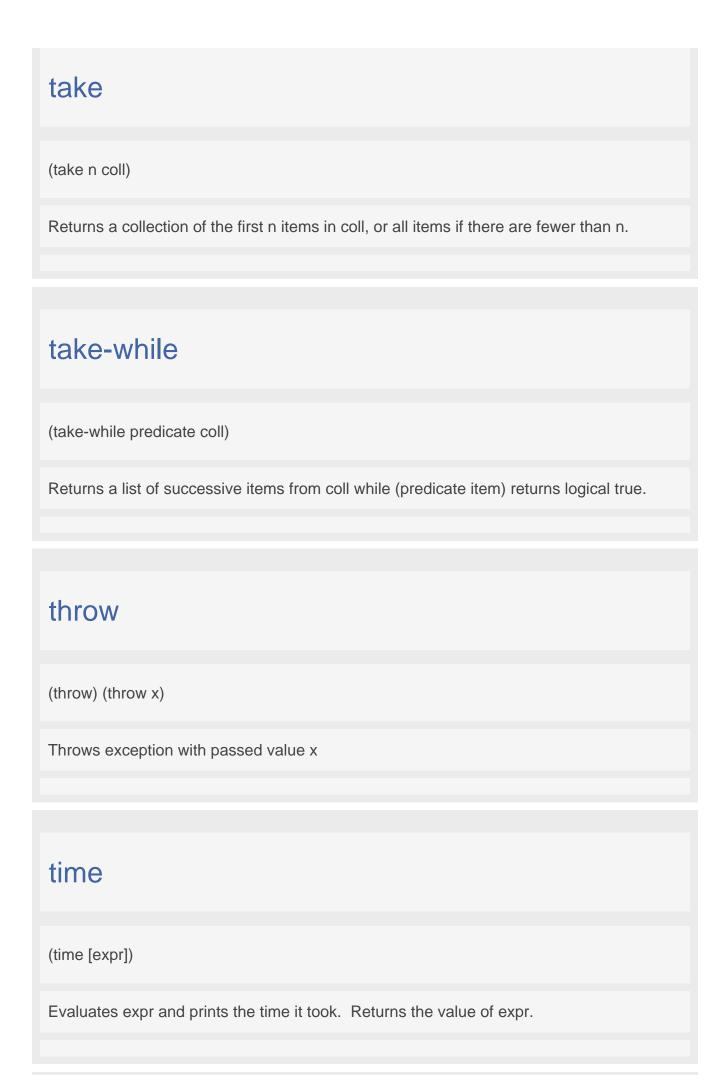
(symbol name)

Returns a symbol from the given name

# symbol?

(symbol? x)

Returns true if x is a symbol



# time-ms) (time-ms) Returns the current time in milliseconds (time-ms) => 1531853605119

#### time-ns

(time-ns)

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

(time-ns)

=> 413049646862939

# true?

(true? x)

Returns true if x is true, false otherwise

# try

```
(try (throw)) (try (throw expr)) (try (throw expr) (catch expr)) (try (throw expr) (catch expr))

Exception handling: try - catch -finally

(try (throw))

=> VncException: nil

(try (throw "test message"))

=> VncException: test message

(try (throw 100) (catch (do (+ 1 2) -1)))

=> -1

(try (throw 100) (finally -2))

=> -2

(try (throw 100) (catch (do (+ 1 2) -1)) (finally -2))

=> -2
```

# uuid

(uuid)

Generates a UUID.

#### val

(val e)

Returns the val of the map entry.



version
(version)
Returns the version.
when
(when test & body)
Evaluates test. If logical true, evaluates body in an implicit do.
when-not
(when-not test & body)
Evaluates test. If logical false, evaluates body in an implicit do.
while
(take-while pred) (take-while pred coll)

Repeatedly executes body while test expression is true. Presumes some side-effect will cause test to become false/nil. Returns nil

# with-meta

(with-meta obj m)

Returns a copy of the object obj, with a map m as its metadata.

# zero?

(zero? x)

Returns true if x zero else false