Cheatsheet

Venice 0.8.2 Cheat Sheet

Primitives

Literals

Literals	Nil: nil Long: 1500 Double: 3.569 Boolean: true, false BigDecimal: 6.897M String: "abcde"

Numbers

Arithmetic	+ - * / mod inc 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 str/quote
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

Collections

Collections

Generic	count empty? not-empty? empty-to-nil empty into conj remove repeat range group-by get-in
Tests	coll? list? vector? set? map? sequential? hash-map? ordered-map? sorted-map? bytebuf?
Process	map filter keep docoll

Lists

Create	() list
Access	first second nth last peek rest nfirst nlast
Modify	cons conj rest pop into concat interpose interleave mapcat flatten seq reduce reverse sort sort-by take take-while drop drop-while
Test	every? any?

Vectors

Create	[] vector mapv
Access	first second nth last peek rest nfirst nlast subvec

	/split-lines str/strip-start str /strip-end str/strip-indent str /strip-margin str/repeat str /truncate
Regex	match match-not
Trim	str/trim str/trim-to-nil
Test	string? empty? str/blank? str /starts-with? str/ends-with? str /contains?

Byte Buffer

Create	bytebuf bytebuf-from-string
Test	bytebuf?
Use	bytebuf-to-string

Other

Keywords	keyword? keyword literals: :a :xyz
Symbols	symbol? symbol
Boolean	boolean not boolean? true? false?

Functions

Create	fn identity
Call	apply comp partial memoize
Test	fn?
Exception	throw
Misc	class type eval
Other	version uuid time-ms time-ns coalesce
Meta	meta with-meta vary-meta

Macros

Modify	cons conj rest pop into concat distinct dedupe partition interpose interleave mapcat flatten seq reduce reverse sort sort-by take take-while drop drop-while assoc-in get-in update update!
Test	contains? every? any?

Sets

Create	set
Modify	difference union intersection
Test	contains?

Maps

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

Other Types

ByteBuffer

Misc	count empty? not-empty? bytebuf bytebuf? subbytebuf

Atoms

Create	atom
Test	atom?

Create	defmacro
Branch	and or when when-not if-let
Loop	list-comp dotimes while
Call	doto -> ->>
Loading	load-string load-file load-module
Test	macro? cond
Assert	assert
Util	comment gensym time

IO

to	print println flush newline
to-str	pr-str
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/temp-file io/tmp-dir io/slurp-temp-file io/user-dir
load	load-file load-string

Access	deref reset! swap! compare-and-set!

Special Forms

Forms	def if do let fn loop defmacro recur try

Java Interoperability

General	. proxify Invoke constructors Invoke static or instance methods Access static or instance fields

Miscellaneous

JSON	json/pretty-print json/to-json json/to-pretty-json json/parse json/avail? json/avail-jdk8-module? Available if Jackson libs are on runtime classpath

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));
  }
}
```

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

    Venice venice = new Venice(interceptor);


    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

!=

(!= x y)

Returns true if both operands do not have the equivalent type

```
(!= 0 1)
=> true

(!= 0 0)
=> false

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

()

Creates a list.

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

*

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

Returns the product of numbers. (*) returns 1

```
(*)
=> 1

(* 4)
=> 4

(* 4 3)
=> 12

(* 4 3 2)
=> 24

(* 6.0 2)
=> 12.0

(* 6 1.5M)
=> 9.0
```

+

(+) (+ 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
```

-

(- 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

(- 8 2.5)
=> 5.5

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

->

```
(-> 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.

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

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

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

->>

```
(->> 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.

```
(->> 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)
```

.

(. classname :new args) (. object method args) (. classname :class) (. object :class)

Java interop. Calls a constructor or an object method. The function is sandboxed

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

;; invoke constructor
(. :java.time.ZonedDateTime :now)
=> 2018-08-16T14:43:19.635+02:00[Europe/Zurich]

;; invoke constructor with param
(. (. :java.lang.Long :new 10) :toString)
=> 10

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

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

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

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

/

(/ 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

(/ 6.0 2)
=> 3.0

(/ 6 1.5M)
=> 4.0000000000000000
```

<

```
(< x y)
```

Returns true if x is smaller than y

```
(< 2 3)
=> true

(< 2 3.0)
=> true

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

<=

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

Returns true if x is smaller or equal to y

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

(<= 3 3)
=> true

(<= 2 3.0)
=> true

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

==

(== x y)

Returns true if both operands have the equivalent type

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

(== 0 1)
=> false

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

>

(> x y)

Returns true if x is greater than y

```
(> 3 2)
=> true

(> 3 3)
=> false

(> 3.0 2)
=> true

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

>=

(>= x y)

Returns true if x is greater or equal to y

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

(>= 3 3)
```

```
=> true

(>= 3.0 2)
=> true

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

[]

Creates a vector

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

abs

(abs x)

Returns the absolute value of the number

```
(abs 10)
=> 10

(abs -10)
=> 10

(abs -10.1)
=> 10.1

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

and

(and x) (and x & next)

Ands the predicate forms

any?

(any? pred coll)

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

```
(any? (fn [x] (number? x)) nil)
=> false

(any? (fn [x] (number? x)) [])
=> false

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

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

(any? (fn [x] (>= x 10)) [1 5 10])
=> true
```

apply

(apply f args* coll)

Applies f to all arguments composed of args and coll

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

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).

```
(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-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))
=> [{:age 26 :name James} {:age 44 :name John}]

(do
  (def users [{:name "James" :age 26}  {:name "John" :age 43}])
  (assoc-in users [2] {:name "Jack" :age 19}) )
=> [{:age 26 :name James} {:age 43 :name John} {:age 19 :name Jack}]
```

atom

(atom x)

Creates an atom with the initial value x

```
(do
  (def counter (atom 0))
  (deref counter))
=> 0
```

atom?

(atom? x)

Returns true if x is an atom, otherwise false

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

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?

(boolean? n)

Returns true if n is a boolean

```
(boolean? true)
=> true

(boolean? false)
=> true

(boolean? nil)
=> false
```

```
(boolean? 0)
=> false
```

bytebuf

```
(bytebuf x)
```

Converts 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]
```

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]
```

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
```

bytebuf?

(bytebuf? x)

Returns true if x is a bytebuf

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

(bytebuf? [1 2])
=> false

(bytebuf? nil)
=> false
```

class

(class x)

Returns the class of x

```
(. :java.lang.Long :class)
=> class java.lang.Long
```

coalesce

(coalesce args*)

Returns the first non nil arg

```
(coalesce [])
=> []

(coalesce [1 2])
=> [1 2]

(coalesce [nil])
=> [nil]

(coalesce [nil 1 2])
=> [nil 1 2]
```

coll?

(coll? obj)

Returns true if obj is a collection

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

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

comment

(comment & body)

Ignores body, yields nil

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.

```
(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-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)
  (deref counter))
=> 4
```

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)
```

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.

```
(conj [1 2 3] 4)
=> [1 2 3 4]

(conj '(1 2 3) 4)
=> (4 1 2 3)
```

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 2] [4 5 6])
=> ([1 2] 4 5 6)
```

contains?

(contains? coll key)

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

```
(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
```

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
```

dec

```
(dec x)
```

Decrements the number x

```
(dec 10)
=> 9

(dec 10.1)
=> 9.1

(dec 10.12M)
=> 9.12
```

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/add 2.44697M 1.79882M 3 :HALF_UP)
=> 4.246

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/div 2.44697M 1.79882M 5 :HALF_UP)
=> 1.36032

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/mul 2.44697M 1.79882M 5 :HALF_UP)
=> 4.40166

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/scale 2.44697M 0 :HALF_UP)
=> 2

(dec/scale 2.44697M 1 :HALF_UP)
=> 2.4

(dec/scale 2.44697M 2 :HALF_UP)
=> 2.45

(dec/scale 2.44697M 3 :HALF_UP)
=> 2.447

```
(dec/scale 2.44697M 10 :HALF_UP)
=> 2.4469700000
```

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)

```
(dec/sub 2.44697M 1.79882M 3 :HALF_UP)
=> 0.648
```

decimal?

```
(decimal? n)
```

Returns true if n is a decimal

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

(decimal? 4.0)
=> false

(decimal? 3)
=> false
```

dedupe

```
(dedupe coll)
```

Returns a collection with all consecutive duplicates removed

```
(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)
```

def

(def name expr)

Creates a global variable.

```
(def val 5)
=> 5
```

defmacro

deref

(deref atom)

Dereferences an atom, returns its value

```
(do
  (def counter (atom 0))
  (deref counter))
=> 0
```

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}
```

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}
```

distinct

```
(distinct coll)
```

Returns a collection with all duplicates removed

```
(distinct [1 2 3 4 2 3 4])
=> [1 2 3 4]

(distinct '(1 2 3 4 2 3 4))
=> (1 2 3 4)
```

do

```
(do exprs)
```

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

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

docoll

(docoll f coll)

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

```
(docoll
  (fn [x] (println x))
  [1 2 3 4])
1
2
3
4
=>

(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
=>
```

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.

```
(doto (. :java.util.HashMap :new)
      (. :put :a 1)
      (. :put :b 2))
=> {a 1 b 2}
```

double?

(double? n)

Returns true if n is a double

(double? 4.0)
=> true

(double? 3)
=> 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

(drop 3 [1 2 3 4 5])
=> [5]

(drop 10 [1 2 3 4 5])
=> []

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.

(drop-while neg? [-2 -1 0 1 2 3])
=> [0 1 2 3]

empty

(empty coll)

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

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

(empty [1 2])
=> []

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

empty-to-nil

(empty-to-nil x)

Returns nil if x is empty

```
(empty-to-nil "")
=>

(empty-to-nil [])
=>

(empty-to-nil '())
=>

(empty-to-nil {})
=>
```

empty?

(empty? x)

Returns true if x is empty

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

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

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

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
```

even?

```
(even? n)
```

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

```
(odd? 4)
=> false

(odd? 3)
=> true
```

every?

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

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

```
(every? (fn [x] (number? x)) nil)
=> false

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

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

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

(every? (fn [x] (>= x 10)) [10 11 12])
=> true
```

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
```

filter

```
(filter predicate coll)
```

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

```
(filter even? [1 2 3 4 5 6 7])
=> [2 4 6]
```

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.

```
(first nil)
=>

(first [])
=>

(first [1 2 3])
=> 1

(first '())
=>

(first '(1 2 3))
=> 1
```

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.

```
(flatten [])
=> []

(flatten [[1 2 3] [4 5 6] [7 8 9]])
=> [1 2 3 4 5 6 7 8 9]
```

flush

(flush)

Flushes the output stream that is the current value of *out*

fn

(fn [params*] expr)

Creates a function.

```
(fn [x y] (+ x y))
=> anonymous-5f0d1e4e-7175-4dbc-998b-ba7b88c9105c

(def sum (fn [x y] (+ x y)))
=> anonymous-9f9a97dc-5afb-4d52-98fb-7c7b8b33df98
```

fn?

(fn? x)

Returns true if x is a function

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

gensym

(gensym) (gensym prefix)

Generates a symbol.

```
(gensym )
=> G__2
```

```
(gensym "prefix_")
=> prefix_3
```

get

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

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

```
(get {:a 1 :b 2} :b)
=> 2

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

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
```

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]}
```

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
```

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]
```

if

(if test true-expr false-expr)

Evaluates test.

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

if-let

(if-let bindings then)

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

(inc 10)
=> 11

(inc 10.1)
=> 11.1

(inc 10.12M)
=> 11.12

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
```

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))
=> #{}
```

into

```
(into to-coll from-coll)
```

Returns a new coll consisting of to-coll with all of the items offrom-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 [] {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]

(into (sorted-map) {:b 2 :c 3 :a 1})
=> {:a 1 :b 2 :c 3}
```

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.

```
(io/exists-dir? (io/file "/temp"))
=> false
```

io/exists-file?

(io/exists-file? x)

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

```
(io/exists-file? (io/file "/temp/test.txt"))
=> false
```

io/file

(io/file path) (io/file parent child)

Returns a java.io.File. path, parent, and child can be a string or java.io.File

```
(io/file "/temp/test.txt")
=> /temp/test.txt

(io/file "/temp" "test.txt")
=> /temp/test.txt

(io/file (io/file "/temp") "test.txt")
=> /temp/test.txt
```

io/file?

(io/file? x)

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

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

io/list-files

(io/list-files dir filterFn?)

Lists files in a directory. dir must be a java.io.File. filterFn is an optional filter that filters the files found

io/slurp-temp-file

(io/slurp-temp-file file & options)

slurps a previously created temp file

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (spit file "123456789" :append true)
    (io/slurp-temp-file file :binary false :remove true))
)
=> 123456789
```

io/temp-file

(io/temp-file prefix suffix)

Creates an empty temp file with prefix and suffix

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (spit file "123456789" :append true)
    (io/slurp-temp-file file :binary false :remove true))
)
=> 123456789
```

io/tmp-dir

(io/tmp-dir)

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

(io/tmp-dir)
=> /var/folders/rm/pjqr5pln3db4mxh5qq1j5yh80000gn/T

io/user-dir

(io/user-dir)

Returns the user dir (current working dir) as a java.io.File.

json/avail-jdk8-module?

(json/avail-jdk8-module?)

Checks if the Jackson JSON JDK8 module is on the classpath

json/avail?

(json/avail?)

Checks if the Jackson JSON libs are on the classpath

json/parse

(json/parse json)

Parse a JSON string

json/pretty-print

(json/pretty-print json)

Pretty print a JSON string

json/to-json

(json/to-json val)

Convert the value to JSON

json/to-pretty-json

(json/to-pretty-json val)

Convert the value to pretty-printed JSON

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.

(keep even? (range 1 4))
=> (false true false)

(keep (fn [x] (if (odd? x) x)) (range 4))
=> (1 3)

key

(key e)

Returns the key of the map entry.

```
(key (find {:a 1 :b 2} :b))
=> :b
```

keys

```
(keys map)
```

Returns a collection of the map's keys.

```
(keys {:a 1 :b 2 :c 3})
=> (:a :b :c)
```

keyword

```
(keyword name)
```

Returns a keyword from the given name

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

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

keyword?

```
(keyword? x)
```

Returns true if x is a keyword

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

(keyword? :a)
=> true

(keyword? nil)
=> false
```

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

last

```
(last coll)
```

Returns the last element of coll.

```
(last nil)
=>

(last [])
=>

(last [1 2 3])
=> 3

(last '())
=>

(last '(1 2 3))
=> 3
```

let

```
(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 )
=> ()
```

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

(list 1 2 3 [:a :b])
=> (1 2 3 [:a :b])
```

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

list?

```
(list? obj)
```

Returns true if obj is a list

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

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

load-file

(load-file name)

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

load-module

(load-module s)

Loads a Venice predefined extension module.

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
```

long?

(long? n)

Returns true if n is a long

```
(long? 4)
=> true

(long? 3.1)
=> false

(long? true)
=> false

(long? nil)
=> false
```

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

loop

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

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

```
(loop [x 10]
  (when (> x 1)
    (println x)
    (recur (- x 2))))
10
8
6
4
2
=>
```

macro?

```
(macro? x)
```

Returns true if x is a macro

map

```
(map f coll colls*)
```

Applys 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.

```
(map inc [1 2 3 4])
=> (2 3 4 5)
```

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 reverse [[3 2 1 0] [6 5 4] [9 8 7]])
=> (0 1 2 3 4 5 6 7 8 9)

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]

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
```

match-not

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

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

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

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

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 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)
=> 1.0

(max 1.0M 2.0M)
=> 2.0
```

```
(max 4.0M 3.0M 2.0M 1.0M)
=> 4.0

(max 1.0M 2)
=> 2
```

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
```

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)
=>
```

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

```
(min 1)
=> 1

(min 1 2)
=> 1

(min 4 3 2 1)
=> 1

(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)
=> 1.0

(min 1.0M 2.0M)
=> 1.0

(min 4.0M 3.0M 2.0M 1.0M)
=> 1.0

(min 1.0M 2)
=> 1.0
```

mod

(mod n d)

Modulus of n and d.

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

neg?

```
(neg? x)
```

Returns true if x smaller than zero else false

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

(neg? 3)
=> false

(neg? -3.2)
=> true

(neg? -3.2M)
=> true
```

newline

```
(newline)
```

Writes a platform-specific newline to *out*

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)
```

nil?

```
(nil? x)
```

Returns true if x is nil, false otherwise

```
(nil? nil)
=> true

(nil? 0)
=> false

(nil? false)
=> false
```

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)

not

(not x)

Returns true if x is logical false, false otherwise.

(not true)
=> false

(not (== 1 2))
=> true

not-empty?

(not-empty? x)

Returns true if x is not empty

(empty? {:a 1})
=> false

(empty? [1 2])
=> false

(empty? '(1 2))
=> false

nth

(nth coll idx)

Returns the nth element of coll.

```
(nth nil 1)
=>

(nth [1 2 3] 1)
=> 2

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

number?

```
(number? n)
```

Returns true if n is a number (long, double, or decimal)

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

(number? 4.0)
=> true

(number? 3)
=> true

(number? true)
=> false

(number? "a")
=> false
```

odd?

```
(odd? n)
```

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

```
(odd? 3)
=> true

(odd? 4)
=> false
```

or

(or x) (or x & next)

Ors the predicate forms

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}

ordered-map?

(ordered-map? obj)

Returns true if obj is an ordered map

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

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.

```
(do
  (def hundred-times (partial * 100))
  (hundred-times 5))
=> 500
```

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.

```
(partition 4 (range 20))
=> ((0 1 2 3) (4 5 6 7) (8 9 10 11) (12 13 14 15) (16 17 18 19))

(partition 4 6 (range 20))
=> ((0 1 2 3) (6 7 8 9) (12 13 14 15) (18 19))

(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))
```

peek

(peek coll)

For a list, same as first, for a vector, same as last

```
(peek '(1 2 3 4))
=> 1

(peek [1 2 3 4])
=> 4
```

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]
```

pos?

```
(pos? x)
```

Returns true if x greater than zero else false

```
(pos? 3)
=> true

(pos? -3)
=> false

(pos? 3.2)
=> true

(pos? 3.2M)
=> true
```

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
```

print

(print & 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 ' '. Returns nil.

```
(print [10 20 30])
[10 20 30]
=>
```

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 ' '. Returns nil.

```
(do (println 200) (println [10 20 30]))
200
[10 20 30]
=>
```

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.

rand-double

(rand-double) (rand-double max)

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

```
(rand-double)
=> 0.01629646125552997

(rand-double 100.0)
=> 82.96565153875201
```

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.

```
(rand-long)
=> 2761060298894380235

(rand-long 100)
=> 31
```

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.

```
(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 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)
=> (1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1)
```

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 (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-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 [x y z] (assoc x z y)) {} {:a 1 :b 2 :c 3})
=> {3 :c 1 :a 2 :b}
```

remove

```
(remove predicate coll)
```

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

```
(filter even? [1 2 3 4 5 6 7])
=> [2 4 6]
```

repeat

```
(repeat n x)
```

Returns a collection with the value x repeated n times

```
(repeat 5 [1 2])
=> ([1 2] [1 2] [1 2] [1 2] [1 2])
```

reset!

```
(reset! atom newval)
```

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

```
(do
  (def counter (atom 0))
  (reset! counter 99)
  (deref counter))
=> 99
```

rest

(rest coll)

Returns a collection with second to list element

```
(rest nil)
=> ()

(rest [])
=> []

(rest [1])
=> []

(rest [1 2 3])
=> [2 3]

(rest '())
=> ()

(rest '(1))
=> ()

(rest '(1 2 3))
=> (2 3)
```

reverse

(reverse coll)

Returns a collection of the items in coll in reverse order

```
(reverse [1 2 3 4 5 6])
=> [6 5 4 3 2 1]
```

second

(second coll)

Returns the second element of coll.

```
(second nil)
=>

(second [])
```

```
=>

(second [1 2 3])
=> 2

(second '())
=>

(second '(1 2 3))
=> 2
```

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.

```
(seq nil)
=>

(seq [1 2 3])
=> (1 2 3)

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

(seq "abcd")
=> (a b c d)
```

sequential?

```
(sequential? obj)
```

Returns true if obj 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? obj)
```

Returns true if obj is a set

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

slurp

```
(slurp file & options)
```

Returns the file's content as text (string) or binary (bytebuf). Defaults to binary=false and encoding=UTF-8. Options: : encoding "UTF-8" :binary true/false.

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
```

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 [3 2 5 4 1 6])
=> [1 2 3 4 5 6]

(sort {:c 3 :a 1 :b 2})
=> ([:a 1] [:b 2] [:c 3])
```

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 count ["aaa" "bb" "c"])
=> [c bb aaa]

(sort-by first [[1 2] [3 4] [2 3]])
=> [[1 2] [2 3] [3 4]]

(sort-by (fn [x] (get x :rank)) [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 1} {:rank 2} {:rank 3}]
```

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}
```

sorted-map?

```
(sorted-map? obj)
```

Returns true if obj is a sorted map

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

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 )
=>

(str 1 2 3)
=> 123
```

str/blank?

(str/blank? s)

True if s is blank.

```
(str/blank? nil)
=> true

(str/blank? "")
=> true

(str/blank? "  ")
=> true

(str/blank? "abc")
=> false
```

str/contains?

(str/contains? s substr)

True if s contains with substr.

```
(str/contains? "abc"  "ab")
=> true
```

str/ends-with?

(str/ends-with? s substr)

True if s ends with substr.

(str/starts-with? "abc" "bc")
=> false

str/format

(str/format format args*)

Returns a formatted string using the specified format string and arguments.

(str/format "%s: %d" "abc" 100)
=> abc: 100

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

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/last-index-of

(str/last-index-of s value) (str/last-index-of s value from-index)

Return last index of value (string or char) in s, optionally
searching backward from from-index. Return nil if value not found.

```
(str/last-index-of "abcdefabc" "ab")
=> 6
```

str/lower-case

(str/lower-case s)

Converts s to lowercase

```
(str/lower-case "aBcDeF")
=> abcdef
```

str/quote

(str/quote str q) (str/quote str start end)

Quotes a string.

```
(str/quote "abc" "-")
=> -abc-

(str/quote "abc" "<" ">")
=> <abc>
```

str/repeat

(str/repeat s n) (str/repeat s n sep)

Repeats s n times with an optional separator.

```
(str/repeat "abc" 0)
=>

(str/repeat "abc" 3)
=> abcabcabc

(str/repeat "abc" 3 "-")
=> abc-abc-abc
```

str/replace-all

```
(str/replace-all s search replacement)
```

Replaces the all occurrances of search in s

```
(str/replace-all "abcdefabc" "ab" "XYZ")
=> XYZcdefXYZc
```

str/replace-first

```
(str/replace-first s search replacement)
```

Replaces the first occurrance of search in s

```
(str/replace-first "abcdefabc" "ab" "XYZ")
=> XYZdefabc
```

str/replace-last

```
(str/replace-last s search replacement)
```

Replaces the last occurrance of search in s

```
(str/replace-last "abcdefabc" "ab" "XYZ")
=> abcdefXYZ
```

str/split

(str/split s regex)

Splits string on a regular expression.

```
(str/split "abc , def , ghi" "[ *],[ *]")
=> (abc def ghi)
```

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
```

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
  line2
  line3
```

str/strip-margin

(str/strip-margin s)

Strips leading whitespaces upto and including the margin '|' from each line in a multi-line string.

```
(str/strip-margin "line1
  line2
  line3")
=> line1
  line2
  line3
```

str/strip-start

(str/strip-start s substr)

Removes a substr only if it is at the beginning of a s, otherwise returns s.

```
(str/strip-start "abcdef" "abc")
=> def

(str/strip-start "abcdef" "def")
=> abcdef
```

str/subs

(str/subs s start) (str/subs s start end)

Returns the substring of s beginning at start inclusive, and ending at end (defaults to length of string), exclusive.

```
(str/subs "abcdef" 2)
=> cdef

(str/subs "abcdef" 2 5)
=> cde
```

str/trim

(str/trim s substr)

Trims leading and trailing spaces from s.

```
(str/trim " abc  ")
=> abc
```

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/trim "")
=>

(str/trim "    ")
=>

(str/trim nil)
=>

(str/trim " abc   ")
=> abc
```

str/truncate

(str/truncate s maxlen marker)

Truncates a string to the max lenght 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

```
(str/upper-case "aBcDeF")
=> ABCDEF
```

string?

```
(string? x)
```

Returns true if x is a string

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

(bytebuf? [1 2])
=> false

(bytebuf? nil)
=> false
```

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)

```
(subbytebuf (bytebuf [1 2 3 4 5 6]) 2)
=> [3 4 5 6]

(subbytebuf (bytebuf [1 2 3 4 5 6]) 4)
=> [5 6]
```

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] 4)
=> [5 6]
```

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.

```
(do
  (def counter (atom 0))
  (swap! counter inc)
  (deref counter))
=> 1
```

symbol

```
(symbol name)
```

Returns a symbol from the given name

```
(symbol "a")
=> a
```

```
(symbol 'a)
=> a
```

symbol?

```
(symbol? x)
```

Returns true if x is a symbol

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

(symbol? 'a)
=> true

(symbol? nil)
=> false

(symbol? :a)
=> false
```

take

```
(take n coll)
```

Returns a collection of the first n items in coll, or all items if there are fewer than n.

```
(take 3 [1 2 3 4 5])
=> [1 2 3]

(take 10 [1 2 3 4 5])
=> [1 2 3 4 5]
```

take-while

```
(take-while predicate coll)
```

Returns a list of successive items from coll while (predicate item) returns logical true.

```
(take-while neg? [-2 -1 0 1 2 3])
=> [-2 -1]
```

throw

(throw) (throw x)

Throws exception with passed value x

```
(try (throw 100) (catch (do (+ 1 2) -1)))
=> -1

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

time

(time expr)

Evaluates expr and prints the time it took. Returns the value of expr.

```
(time (println [100 200]))
[100 200]
Elapsed time: 0 msecs
=>
```

time-ms

(time-ms)

Returns the current time in milliseconds

```
(time-ms)
=> 1534423398831
```

time-ns

(time-ns)

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

```
(time-ns)
=> 206205254448458
```

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
```

try

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

type

```
(type x)
```

Retruns the type of x.

```
(type 5)
=> venice.Long

(type (. :java.time.ZonedDateTime :now))
=> venice.JavaObject(java.time.ZonedDateTime)
```

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}
```

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}
```

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.

```
(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}
```

uuid

```
(uuid)
```

Generates a UUID.

```
(uuid )
=> 46590b5a-a3ad-4884-b772-f2b02c8e6fe2
```

val

```
(val e)
```

Returns the val of the map entry.

```
(val (find {:a 1 :b 2} :b))
=> 2
```

vals

```
(vals map)
```

Returns a collection of the map's values.

```
(vals {:a 1 :b 2 :c 3})
=> (1 2 3)
```

vary-meta

```
(vary-meta obj f & args)
```

Returns a copy of the object obj, with (apply f (meta obj) args) as its metadata.

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?

```
(vector? obj)
```

Returns true if obj is a vector

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

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

version

```
(version)
```

Returns the version.

```
(version )
=> 0.8.2
```

when

```
(when test & body)
```

Evaluates test. If logical true, evaluates body in an implicit do.

when-not

```
(when test & body)
```

Evaluates test. If logical false, evaluates body in an implicit do.

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

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

```
(zero? 0)
=> true

(zero? 2)
=> false

(zero? 0.0)
=> true

(zero? 0.0M)
=> true
```

zipmap

(zipmap keys vals)

Returns a map with the keys mapped to the corresponding vals.

```
(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}
```

{}

Creates a hash map.

```
{:a 10 b: 20}
=> {:a 10 b: 20}
```