

# Cheatsheet

## Venice 0.8.1 Cheat Sheet

### Primitives

#### Literals

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

#### Numbers

Arithmetic	<a href="#">+</a> <a href="#">-</a> <a href="#">*</a> <a href="#">/</a> <a href="#">mod</a> <a href="#">inc</a> <a href="#">dec</a> <a href="#">min</a> <a href="#">max</a> <a href="#">abs</a>
Compare	<a href="#">==</a> <a href="#">!=</a> <a href="#">&lt;</a> <a href="#">&gt;</a> <a href="#">&lt;=</a> <a href="#">&gt;=</a>
Test	<a href="#">nil?</a> <a href="#">some?</a> <a href="#">zero?</a> <a href="#">pos?</a> <a href="#">neg?</a> <a href="#">even?</a> <a href="#">odd?</a> <a href="#">number?</a> <a href="#">long?</a> <a href="#">double?</a> <a href="#">decimal?</a>
Random	<a href="#">rand-long</a> <a href="#">rand-double</a>
BigDecimal	<a href="#">dec/add</a> <a href="#">dec/sub</a> <a href="#">dec</a> <a href="#">/mul</a> <a href="#">dec/div</a> <a href="#">dec/scale</a>

#### Strings

Create	<a href="#">str</a> <a href="#">str/format</a> <a href="#">str/quote</a>
Use	<a href="#">count</a> <a href="#">empty-to-nil</a> <a href="#">str/index-of</a> <a href="#">str/last-index-of</a> <a href="#">str/replace-first</a> <a href="#">str/replace-last</a> <a href="#">str/replace-all</a> <a href="#">str/lower-case</a> <a href="#">str/upper-case</a> <a href="#">str/join</a> <a href="#">str/subs</a> <a href="#">str/split</a> <a href="#">str</a>

### Collections

#### Collections

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

#### Lists

Create	<a href="#">()</a> <a href="#">list</a>
Access	<a href="#">first</a> <a href="#">second</a> <a href="#">nth</a> <a href="#">last</a> <a href="#">peek</a> <a href="#">rest</a> <a href="#">nfirst</a> <a href="#">nlast</a>
Modify	<a href="#">cons</a> <a href="#">conj</a> <a href="#">rest</a> <a href="#">pop</a> <a href="#">into</a> <a href="#">concat</a> <a href="#">interpose</a> <a href="#">interleave</a> <a href="#">mapcat</a> <a href="#">flatten</a> <a href="#">reduce</a> <a href="#">reverse</a> <a href="#">sort</a> <a href="#">sort-by</a> <a href="#">take</a> <a href="#">take-while</a> <a href="#">drop</a> <a href="#">drop-while</a>
Test	<a href="#">every?</a> <a href="#">any?</a>

#### Vectors

Create	<a href="#">[]</a> <a href="#">vector</a> <a href="#">mapv</a>
Access	<a href="#">first</a> <a href="#">second</a> <a href="#">nth</a> <a href="#">last</a> <a href="#">peek</a> <a href="#">rest</a> <a href="#">nfirst</a> <a href="#">nlast</a> <a href="#">subvec</a>

	<a href="#">/split-lines</a> <a href="#">str/strip-start</a> <a href="#">str/strip-end</a> <a href="#">str/strip-indent</a> <a href="#">str/strip-margin</a> <a href="#">str/repeat</a> <a href="#">str/truncate</a>
Regex	<a href="#">match</a> <a href="#">match-not</a>
Trim	<a href="#">str/trim</a> <a href="#">str/trim-to-nil</a>
Test	<a href="#">string?</a> <a href="#">empty?</a> <a href="#">str/blank?</a> <a href="#">str/starts-with?</a> <a href="#">str/ends-with?</a> <a href="#">str/contains?</a>
Other	
Keywords	<a href="#">keyword?</a> <a href="#">keyword</a> literals: :a :xyz
Symbols	<a href="#">symbol?</a> <a href="#">symbol</a>
Boolean	<a href="#">boolean</a> <a href="#">not</a> <a href="#">boolean?</a> <a href="#">true?</a> <a href="#">false?</a>

## Functions

Create	<a href="#">fn</a> <a href="#">identity</a>
Call	<a href="#">apply</a> <a href="#">comp</a> <a href="#">partial</a> <a href="#">memoize</a>
Test	<a href="#">fn?</a>
Exception	<a href="#">throw</a>
Misc	<a href="#">class</a> <a href="#">eval</a>
Other	<a href="#">version</a> <a href="#">uuid</a> <a href="#">time-ms</a> <a href="#">time-ns</a> <a href="#">coalesce</a>
Meta	<a href="#">meta</a> <a href="#">with-meta</a> <a href="#">vary-meta</a>

## Macros

Create	<a href="#">defmacro</a>
Branch	<a href="#">and</a> <a href="#">or</a> <a href="#">when</a> <a href="#">when-not</a> <a href="#">if-let</a>
Loop	<a href="#">list-comp</a> <a href="#">dotimes</a> <a href="#">while</a>
Call	<a href="#">doto</a> <a href="#">-&gt;</a> <a href="#">-&gt;&gt;</a>

Modify	<a href="#">cons</a> <a href="#">conj</a> <a href="#">rest</a> <a href="#">pop</a> <a href="#">into</a> <a href="#">concat</a> <a href="#">distinct</a> <a href="#">dedupe</a> <a href="#">partition</a> <a href="#">interpose</a> <a href="#">interleave</a> <a href="#">mapcat</a> <a href="#">flatten</a> <a href="#">reduce</a> <a href="#">reverse</a> <a href="#">sort</a> <a href="#">sort-by</a> <a href="#">take</a> <a href="#">take-while</a> <a href="#">drop</a> <a href="#">drop-while</a> <a href="#">assoc-in</a> <a href="#">get-in</a> <a href="#">update</a> <a href="#">update!</a>
Test	<a href="#">contains?</a> <a href="#">every?</a> <a href="#">any?</a>

## Sets

Create	<a href="#">set</a>
Modify	<a href="#">difference</a> <a href="#">union</a> <a href="#">intersection</a>
Test	<a href="#">contains?</a>

## Maps

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

## Other Types

### ByteBuffer

Misc	<a href="#">count</a> <a href="#">empty?</a> <a href="#">not-empty?</a> <a href="#">bytebuf</a> <a href="#">bytebuf?</a> <a href="#">subbytebuf</a>
------	--

## Atoms

Create	<a href="#">atom</a>
Test	<a href="#">atom?</a>

Loading	<a href="#">load-string</a> <a href="#">load-file</a> <a href="#">load-module</a>
Test	<a href="#">macro?</a> <a href="#">cond</a>
Assert	<a href="#">assert</a>
Util	<a href="#">comment</a> <a href="#">gensym</a> <a href="#">time</a>

Access	<a href="#">deref</a> <a href="#">reset!</a> <a href="#">swap!</a> <a href="#">compare-and-set!</a>
--------	---

## Special Forms

Forms	<a href="#">def</a> <a href="#">if</a> <a href="#">do</a> <a href="#">let</a> <a href="#">fn</a> <a href="#">loop</a> <a href="#">defmacro</a> <a href="#">recur</a> <a href="#">try</a>
-------	---

## IO

to	<a href="#">print</a> <a href="#">println</a> <a href="#">flush</a> <a href="#">newline</a>
to-str	<a href="#">pr-str</a>
from	<a href="#">readline</a> <a href="#">read-string</a>
file-io	<a href="#">slurp</a> <a href="#">spit</a> <a href="#">io/file</a> <a href="#">io/file?</a> <a href="#">io/exists-file?</a> <a href="#">io/exists-dir?</a> <a href="#">io/list-files</a> <a href="#">io/delete-file</a> <a href="#">io/copy-file</a> <a href="#">io/temp-file</a> <a href="#">io/tmp-dir</a> <a href="#">io/slurp-temp-file</a> <a href="#">io/user-dir</a>
load	<a href="#">load-file</a> <a href="#">load-string</a>

## Java Interoperability

General	<a href="#">.</a> <a href="#">proxify</a> Invoke constructors Invoke static or instance methods Access static or instance fields
---------	---

## Miscellaneous

JSON	<a href="#">json/pretty-print</a> <a href="#">json/to-json</a> <a href="#">json/to-pretty-json</a> <a href="#">json/parse</a> <a href="#">json/avail?</a> <a href="#">json/avail-jdk8-module?</a> 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

**()**

Creates a list.

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

**\***

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

Returns the product of numbers. (\*) returns 1

**+**

(+) (+ 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.

->

```
(-> 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)  
=> 2018-08-10T15:54:08.241+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.

<

(< x y)

Returns true if x is smaller than y

<=

(<= x y)

Returns true if x is smaller or equal to y

==

(== x y)

Returns true if both operands have the equivalent type

>

(> x y)

Returns true if x is greater than y

`>=`

`(>= x y)`

Returns true if x is greater or equal to y

`[]`

Creates a vector

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

`abs`

`(abs x)`

Returns the absolute value of the number

`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

## 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  $\leq$  (count vector).

## 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.

## atom

(atom x)

Creates an atom with the initial value x

## 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

(bytebuf x)

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

## bytebuf?

(bytebuf? x)

Returns true if x is a bytebuf

## class

(class x)

Returns the class of x

## coalesce

(coalesce args\*)

Returns the first non nil arg

## coll?

(coll? obj)

Returns true if obj is a collection

## 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

## concat

(concat coll) (concat coll & colls)

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

## 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)

## decimal?

(decimal? n)

Returns true if n is a decimal

## dedupe

(dedupe coll)

Returns a collection with all consecutive duplicates removed

## def

```
(def name expr)
```

Creates a global variable.

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

## defmacro

## deref

```
(deref atom)
```

Dereferences an atom, returns its value

## 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)

## distinct

(distinct coll)

Returns a collection with all duplicates removed

## 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

## drop

(drop n coll)

Returns a collection of all but the first n items in coll

## 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.

## empty

(empty coll)

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

## empty-to-nil

(empty-to-nil x)

Returns nil if x is empty

## empty?

(empty? x)

Returns true if x is empty

## 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

## 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

## filter

(filter predicate coll)

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

## 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.

## 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-e1d89b2e-d7ff-4bd5-a7ae-5211c605a5ea
```

```
(def sum (fn [x y] (+ x y)))  
=> anonymous-6b886d9a-5b09-49ce-b971-a64470bccbb6
```

## 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.

```
(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.

## hash-map

(hash-map & keyvals) (hash-map map)

Creates a new hash map containing the items.

## hash-map?

(hash-map? obj)

Returns true if obj is a hash map

## 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

## 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-file?

(io/exists-file? x)

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

## 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?

(io/file? x)

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

## 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/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)



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

## 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)
```

Returns true if x is a keyword

## last

```
(last coll)
```

Returns the last element of coll.

## 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-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

## 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

## 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\*)

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.

## map?

(map? obj)

Returns true if obj is a map

## 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.

## 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.

## 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
```

## 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

## mod

(mod n d)

Modulus of n and d.

## neg?

(neg? x)

Returns true if x smaller than zero else false

## newline

(newline)

Writes a platform-specific newline to \*out\*

## first

(first coll n)

Returns a collection of the first n items

## nil?

(nil? x)

Returns true if x is nil, false otherwise

## nlast

(nlast coll n)

Returns a collection of the last n items

## 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

## nth

```
(nth coll idx)
```

Returns the nth element of coll.

## number?

```
(number? n)
```

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

## odd?

```
(odd? n)
```

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

## 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?

(ordered-map? obj)

Returns true if obj is an ordered map

## 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 ofd6ofd6ofd6ofd tep apart. Ifd tep is not supplied, defaulofdto n, i.e. the

(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 ' '.

## 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 trailing 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)

Proxies 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-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

(rest coll)

Returns a collection with second to list element

## reverse

(reverse coll)

Returns a collection of the items in coll in reverse order

## second

(second coll)

Returns the second element of coll.

## 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

## 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) (sorted-map map)

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/blank?

(str/blank? s)

True if s is blank.

## 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 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)

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

## str/join

(str/join coll) (str/join separator coll)

Joins all elements in coll separated by an optional separator.

## 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/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)

=> abcabcab

(str/repeat "abc" 3 "-")

=> abc-abc-abc

## str/replace-all

(str/replace-all s search replacement)

Replaces the all occurrences of search in s

## str/replace-first

(str/replace-first s search replacement)

Replaces the first occurrence of search in s

## str/replace-last

(str/replace-last s search replacement)

Replaces the last occurrence of search in s

## str/split

(str/split s regex)

Splits string on a regular expression.

## str/split-lines

(str/split-lines s)

Splits s into lines.

## str/starts-with?

(str/starts-with? s substr)

True if s starts with substr.

## 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/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 empty

## 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



## 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

## take

(take n coll)

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

## take-while

(take-while predicate coll)

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

## throw

(throw) (throw x)

Throws exception with passed value x

## time

```
(time expr)
```

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

```
(time (println [100 200]))  
[100 200]  
Elapsed time: 1 msecs  
=>
```

## time-ms

```
(time-ms)
```

Returns the current time in milliseconds

```
(time-ms)  
=> 1533909247843
```

## time-ns

```
(time-ns)
```

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

```
(time-ns)  
=> 63990899439507
```

## 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) (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
```

## 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.

## val

(val e)

Returns the val of the map entry.

## vals

(vals map)

Returns a collection of the map's values.

## 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? obj)

Returns true if obj is a vector

## version

(version)

Returns the version.

## when

(when test & body)

Evaluates 1 0 1 two logicals tru, eEvaluates bod in an implicit don.

(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

## 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}