Literals

NUMBERS 2 1.3

STRINGS "abc"

BOOLEANS true false

KEYWORDS :a

NIL nil

LISTS '()

VECTORS []

SETS #{ }

MAPS { }

FUNCTIONS (fn [])

SYMBOLS ' def let

expressions (foo)

Numbers

CREATE 1 1.5 -1 -1.5

ARITHMETIC + - / * inc dec max

min

TEST even? odd?

COMPARE == > < >= <=

RANDOM rand rand-int

Booleans

CREATE true false boolean

TEST = not=

COMBINE and or not

BRANCH if when case cond

Strings

CREATE " " str

EXAMINE count get

CHANGE reverse join split

replace trim

CHANGE upper-case lower-case

CASE capitalize

TEST blank? includes?

REGEX #" " re-matches

re-find re-seq

re-pattern replace

Vectors

CREATE	[]	CREATE	'() range repeat
INSPECT	count	INSPECT	count
EXTRACT	first last rand-nth	EXTRACT	first last rand-nth
	get	REORDER	shuffle reverse sort
REORDER	shuffle reverse sort	LONGER	conj concat
LONGER	conj concat	SHORTER	rest drop drop-last
SHORTER	rest drop drop-last		take take-last
	take take-last		take-while filter
	take-while filter		remove
	remove	CHANGE	map map-indexed reduce
CHANGE	assoc map map-indexed	TEST	some every? empty?
	reduce		, , ,
TEST	some every? empty?		

Lists

Maps

CREATE { }

EXTRACT get select-keys keys

vals

TEST contains?

CHANGE assoc dissoc merge

update

Sets

CREATE

#{ } set

EXAMINE

get contains?

CHANGE

conj disj

OPS

union difference

intersection

TEST

subset? superset?

Nested Data Structures

CREATE

[]{}#{}

EXAMINE

get-in

CHANGE

assoc-in update-in

sort-by group-by

filter map reduce

Literals

LITERALS | NUMBERS

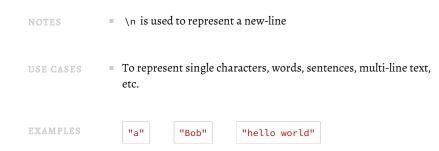
2 1.3

EXAMPLES 2 1.3 -2 -1.3

LITERALS | STRINGS

"abc"

Strings are used to represent text.



LITERALS | BOOLEANS

true false



LITERALS | KEYWORDS

: a

USE CASES

- Inside of maps, as keys
- To represent commonly reused values in data, ex. :male and :female

EXAMPLES

:a

:hello

LITERALS | NIL

nil

EXAMPLES

nil

LITERALS | LISTS



USE CASES

 Rarely written in code directly, but, they are returned by many functions that work with vectors and sets.

EXAMPLES

'(1 2 3 4)

'(:a :b :c)

LITERALS | VECTORS

NOTES

- Most functions that work with lists also work with vectors
- Vectors allow you to access a value at a certain index directly (whereas lists do not)

USE CASES

- To represent multiple values in a list
- Preferred to lists when writing code, because it is easier to write and distinguish [] than '()

EXAMPLES

[1 2 3 4]

[:a :b :c]

LITERALS | SETS

#{ }

Sets are sequences where each value is unique; a set will never contain the same value twice.

NOTES Most functions that work with lists also work with sets

 To represent a group of values, where order does not matter, and where there should only be one of each value

XAMPLES #{1 4 3 2} #{:c :b :a}

LITERALS | MAPS

{ }



```
{:email "bob@example.com"
:name "Bob"}
```

LITERALS | FUNCTIONS

(fn [])

EXAMPLES

(fn [x] (+ 1 x))

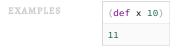
LITERALS | SYMBOLS

Labels to other values. Clojure comes with labels for many functions (ex. +, max) but you can create your own labels for functions and values too.



def

Labels to other values. Clojure comes with labels for many functions (ex. +, max) but you can create your own labels for functions and values too.



let

Labels to other values. Clojure comes with labels for many functions (ex. +, max) but you can create your own labels for functions and values too.



LITERALS | EXPRESSIONS

(foo)

EXAMPLES

(foo "bar")

(+ 1 2 (* 4 5))

Numbers

NUMBERS | CREATE

1 1.5 -1 -1.5

EXAMPLES 1 1.5 -1 -1.5

NUMBERS | ARITHMETIC

$$(+ x y \& more) (+ x y) (+ x) (+)$$

Add multiple numbers together.

NOTES = (+) returns 0

USE CASES With apply, can sum a list

With apply, can sum a list

 (+ 1 1 1)
 (apply + [1 1 1])

 3
 3

(reduce + [1 1 1])

$$(-xy \& more) (-xy) (-x)$$

Subract multiple numbers from x.

$$(/ \times y \& more) (/ \times y) (/ \times)$$

Divide x by one or more ys.

Multiplies one or more numbers.

(inc x)

Returns a number one greater than x.

USE CASES Most often inside of map, update, update-in, to increase a value by 1.

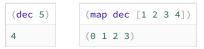
(dec x)

Returns a number one less than x.

NOTES The same as (- x 1)

USE CASES Most often inside of map, update, update-in, to decrease a

value by 1.



```
(update [2 4 6] 0 dec)
```

Returns the greatest of the input numbers.

USE CASES

- To find the maximum of some numbers
- To clamp a number above some threshold
- To find the maximum of a list of numbers (with apply)

EXAMPLES

Returns the least of the input numbers.

```
(apply min [1 5 1 -1 0])
```

NUMBERS | TEST

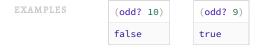
(even? n)

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



(odd? n)

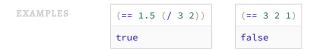
Returns true if n is odd, throws an exception if n is not an integer



NUMBERS | COMPARE

$$(== x y \& more) (== x y) (== x)$$

Returns true if all numbers have the same value, otherwise returns false.



Returns true if each number is greater-than the next number, otherwise returns false.



$$(\langle x y \& more) (\langle x y) (\langle x) \rangle$$

Returns true if each number is less-than the next number, otherwise returns false.

$$(>= x y \& more) (>= x y) (>= x)$$

Returns true if each number is greater-than-or-equal to the next number, otherwise returns false.

USE CASES
In an if or filter, to check if one value is >= to another.

In an if or filter, to check if set of values is descending.

EXAMPLES

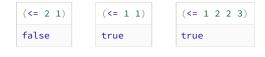
(>= 3 2 2 1)

(apply >= [3 2 2 1])
true

Returns true if each number is less-than-or-equal to the next number, otherwise returns false.

USE CASES

- In an if or filter, to check if one value is <= to another.
- In an if or filter, to check if a value is between a certain range.



```
(let [a 39]
  (<= 0 a 100))
true
```

NUMBERS | RANDOM

(rand n) (rand)

Returns a random decimal number between 0 (inclusive) and 1 (exclusive).

If n is provided, returns a random decimal number between 0 (inclusive) and n (exclusive).



(rand-int n)

Returns a random integer between 0 (inclusive) and 1 (exclusive).



Booleans

BOOLEANS | CREATE

true false



(boolean x)

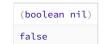
Converts an input to true or false

NOTES Input values of nil or false will return false. Everything

- Usually unnecessary, because most functions handle truthy values as true and falsey values as false.
- Sometimes necessary when some other part of the program is expecting exactly true or false.







BOOLEANS | TEST

$$(= x y \& more) (= x y) (= x)$$

Returns true if the values of all inputs are equal, otherwise, returns false.

NOTES • All input expressions will be evaluated.

use cases In an if or filter, to check if two or more values are equal.

To check if all values in a list are equal (with apply).



```
(apply = [4 (/ 8 2) (* 2 2)])
true
```

The opposite of =. Returns false when all values are equal, otherwise true.

NOTES Same as (not (= x y ...))

• All input expressions will be evaluated.

USE CASES • In an if or filter, to check if two values are not equal.

EXAMPLES

(not= 1 1)

(not= 1 0) true

BOOLEANS | COMBINE

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

Evaluates expressions one at a time, from left to right. If an expression returns falsey (nil or false), and returns that value and doesn't evaluate any of the other expressions, otherwise it returns the value of the last expression.

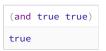
NOTES

- (and) returns true.
- and is a 'special form' not a function. Only the expressions until the first falsey value are evaluated (in a function, all input expressions are evaluated)

USE CASES

In an if or filter, to check if all of a set of conditions are true

EXAMPLES



(and true false)
false

(and false true)

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

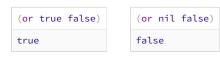
NOTES

• or is a 'special form' not a function. Only the expressions until the first truthy value are evaluated (in a function, all input expressions are evaluated)

USE CASES

- In an if or filter, to check if any of a set of conditions are true
- To provide a default value in situations where some value may be nil or false

EXAMPLES



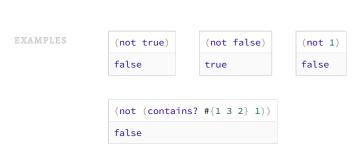
(or nil 3)

(not x)

Returns true if x is falsey, otherwise returns false.

USE CASES

Inside of a multi-part conditional statement, to negate a logical value



(if test-expr true-result-expr false-result-expr)

If test-expr is truthy, if will return true-result-expr, otherwise, it will return false-result-expr.

NOTES

• if is a 'special form', not a function; only one of true-result-expr or false-result-expr is evaluated (with a function, all input expressions are evaluated before the function is applied).







```
(if nil
"a"
"b")
```

```
(let [x 50]
  (if (= x 100)
   "equal"
   "different"))
"different"
```



(when test-expr result-expr)

If test-expr is truthy, when will return result-expr, otherwise, it will return nil

NOTES

- when is a 'special form', not a function; result-expr is only evaluated if test-expr is truthy (with a function, all input expressions are evaluated before the function is applied).
- Equivalent to: (if test-expr result-expr nil)

```
(when (= 1 1)
  true)
```

```
(when (not= 1 1)
    true)
nil
```

(case expression & test-constant result-expression)

Takes an expression and a set of of test-constant / result-expression pairs.

If expression is equal to a test-constant, the corresponding result-expression is returned.

NOTES

- case is a 'special form', not a function; only the matching result-expression is evaluated (with a function, all input expressions are evaluated before the function is applied).
- A default expression can be provided
- The test-constants can be of different types
- Can use a list as a test-constant to match on multiple things.
- If no default is provided, and no test-constant matches, an exception is thrown.

```
(let [letter "x"]
  (case letter
    "a" 10
    "b" 23
    "c" 15
    0))
```

```
(let [n 5]
  (case n
    (8 9 10) :high
    (5 6 7) :medium
    :low))
:medium
```

(cond & test expression)

Takes a set of test / expression pairs. It evaluates each test one at a time. If a test returns a truthy value, cond evaluates and returns the value of the corresponding expression and doesn't evaluate any of the other tests or expressions.

NOTES

- cond is a 'special form', not a function; only the matching expression is evaluated (with a function, all input expressions are evaluated before the function is applied).
- To provide a default, it is common to use :else as a final test (because :else is truthy)

Strings

STRINGS | CREATE

11 11

```
EXAMPLES "Hello World"
```

```
(str x & ys) (str x) (str)
```

With no args, returns the empty string. With one arg x, returns x.toString(). (str nil) returns the empty string. With more than one arg, returns the concatenation of the str values of the args.

```
use cases To convert a non-string to a string
```

To create strings from multiple values

```
EXAMPLES
```

```
(str 10)
```

```
(let [name "Bob"]
  (str "My name is" name))
```

```
(count coll)
```

Returns the number of items in the collection. (count nil) returns

1. Also works on strings, arrays, and Maps

```
(count "Hello World")

11
```

```
(get o k not-found) (get o k)
```

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

```
(get "Hello World" 4)
"o"
```

(reverse s)

Returns s with its characters reversed.

```
(clojure.string/reverse "Hello World")

"dlroW olleH"
```

```
(join separator coll) (join coll)
```

Returns a string of all elements in coll, as returned by (seq coll), separated by an optional separator.

```
(clojure.string/join ", " ["Alice" "Bob" "Cathy"])

"Alice, Bob, Cathy"
```

```
(split s re limit) (split s re)
```

Splits string on a regular expression. Optional argument limit is the maximum number of splits. Not lazy. Returns vector of the splits.

```
(clojure.string/split "Hello my name is Bob" " ")

["Hello" "my" "name" "is" "Bob"]
```

(replace s match replacement)

Replaces all instance of match with replacement in s. match/replacement can be: string / string pattern / (string or function of match).

```
(clojure.string/replace "Hello my name is Bob"
"Bob"
"Cathy")

"Hello my name is Cathy"
```

(trim s)

Removes whitespace from both ends of string.



STRINGS | CHANGE CASE

(upper-case s)

Converts string to all upper-case.

EXAMPLES

(clojure.string/upper-case "Hello World")
"HELLO WORLD"

(lower-case s)

Converts string to all lower-case.

```
(clojure.string/lower-case "Hello World")
"hello world"
```

(capitalize s)

Converts first character of the string to upper-case, all other characters to lower-case.

(clojure.string/capitalize "hello world")

"Hello world"

(blank? s)

True is s is nil, empty, or contains only whitespace.

```
(clojure.string/blank? " ")
true

(clojure.string/blank? "Hello World")
false
```

(includes? s substr)

True if s includes substr.

```
(clojure.string/includes? "Hello World" "e")
true

(clojure.string/includes? "Hello World" "xyz")
false
```

#" "

EXAMPLES #"^a[a-z]{3}\$"

(re-matches re s)

Returns the result of (re-find re s) if re fully matches s.

```
(re-matches #"^a[a-z]{3}$" "abcd")
"abcd"
```

```
(re-matches #"^a([a-z]{3})$" "abcd")
["abcd" "bcd"]
```

```
(re-matches #"^[a-z]{4}$" "1234")
nil
```

```
(re-find re s)
```

Returns the first regex match, if any, of s to re, using re.exec(s). Returns a vector, containing first the matching substring, then any capturing groups if the regular expression contains capturing groups.

```
(re-find #"[0-9]{3}" "abc123xyz456")

"123"

(re-find #"[a-z]{3}([0-9]{3})" "abc123xyz456")

["abc123" "123"]

(re-find #"[0-9]{3}" "abcdef")

nil
```

```
(re-seq re s)
```

Returns a lazy sequence of successive matches of re in s.

```
(re-seq #"[0-9]{3}" "abc123xyz456")
("123" "456")

(re-seq #"[0-9]{3}" "abcdefg")
nil

(re-seq #"([a-z]{3})([0-9]{3})" "abc123xyz456")
```

```
(re-pattern s)
```

Returns an instance of RegExp which has compiled the provided string.

(["abc123" "abc" "123"] ["xyz456" "xyz" "456"])

```
(re-pattern "abcdef")
#"abcdef"
```

```
(replace smap coll) (replace smap)
```

Given a map of replacement pairs and a vector/collection, returns a vector/seq with any elements = a key in smap replaced with the corresponding val in smap. Returns a transducer when no collection is provided.

```
(clojure.string/replace
"my postal code is a2c 3d4"
#"\w\d\w ?\d\w\d"
"XXX-XXXX")
my
```

Vectors

VECTORS | CREATE

Γ٦

EXAMPLES

[1 2 3 4 5]

["a" "b" "cde"]

VECTORS | INSPECT

(count coll)

See count under Lists

VECTORS | EXTRACT

(first coll)

See first under Lists

(last s)

See last under Lists

(rand-nth coll)

See rand-nth under Lists

```
(get o k not-found) (get o k)
```

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

```
(get [:a :b :c] 0)
```

```
(get [:a :b :c] 5)
nil
```

VECTORS | REORDER

```
(shuffle coll)
```

See shuffle under Lists

(reverse coll)

See reverse under Lists

(sort comp coll) (sort coll)

See sort under Lists

VECTORS | LONGER

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

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

NOTES

 conj works on both lists and vectors, but differently; for vectors, conj adds the value to the *end* of the vector, but with lists, conj adds the value to the *beginning*

EXAMPLES

```
(concat x y & zs) (concat x y) (concat x)
(concat)
```

See concat under Lists

VECTORS | SHORTER

```
(rest coll)
```

See rest under Lists

See drop under Lists

See drop-last under Lists

See take under Lists

```
(take-last n coll)
   See take-last under Lists
(take-while pred coll) (take-while pred)
   See take-while under Lists
(filter pred coll) (filter pred)
   See filter under Lists
```

(remove pred coll) (remove pred)

See remove under Lists

VECTORS | CHANGE

```
(assoc coll k v & kvs) (assoc coll k v)
```

assoc[iate]. When applied to a map, returns a new map of the same (hashed/sorted) 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.

```
NOTES = Index must be in bounds of existing vector

USE CASES = Rarely used

EXAMPLES (assoc [:a :b :c] 2 :x)

[:a :b :x]
```

```
(map f c1 c2 c3 & colls) (map f c1 c2 c3)
(map f c1 c2) (map f coll) (map f)
```

Returns a lazy sequence 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. Function f should accept number-of-colls arguments. Returns a transducer when no collection is provided.

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

Returns a lazy sequence consisting of the result of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item in coll, etc, until coll is exhausted. Thus function f should accept 2 arguments, index and item. Returns a stateful transducer when no collection is provided.

```
(reduce f val coll) (reduce f 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.

VECTORS | TEST

(some pred coll)

Returns the first logical true value of (pred x) for any x in coll, else nil. One common idiom is to use a set as pred, for example this will return: fred if: fred is in the sequence, otherwise nil: (some #{: fred} coll)

(every? pred coll)

Returns true if (pred x) is logical true for every x in coll, else false.

(empty? coll)

Returns true if coll has no items - same as (not (seq coll)). Please use the idiom (seq x) rather than (not (empty? x))

Lists

LISTS | CREATE

```
*( )

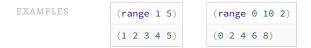
EXAMPLES

'(1 2 3 4 5)

'("a" "b" "cde")
```

```
(range start end step) (range start end)
(range end) (range)
```

Returns a lazy seq of nums from start (inclusive) to end (exclusive), by step, where start defaults to 0, step to 1, and end to infinity.



```
(repeat n x) (repeat x)
```

Returns a lazy (infinite!, or length n if supplied) sequence of xs.

```
(repeat 5 :a)
(:a :a :a :a :a)
```

LISTS | INSPECT

```
(count coll)
```

Returns the number of items in the collection. (count nil) returns

1. Also works on strings, arrays, and Maps

```
(count [:a :b :c :d :e]) (count [])
5 0
```

(first coll)

Returns the first item in the collection. Calls seq on its argument. If coll is nil, returns nil.

(last s)

Return the last item in coll, in linear time

```
(last [1 2 3 4])
4
```

```
(rand-nth coll)
```

Return a random element of the (sequential) collection. Will have the same performance characteristics as nth for the given collection.

```
(rand-nth [:a :b :c])

:b

(rand-nth [:a :b :c])
```

(shuffle coll)

Return a random permutation of coll

NOTES Returns a vector, even if given a list

(shuffle [:a :b :c :d :e])

[:a :c :d :e :b]

(reverse coll)

Returns a seq of the items in coll in reverse order. Not lazy.

NOTES Returns a list, even if given a vector

(reverse [:a :b :c :d :e])
(:e :d :c :b :a)

```
(sort comp coll) (sort coll)
```

Returns a sorted sequence of the items in coll. Comp can be boolean-valued comparison function, or a -/o/+ valued comparator. Comp defaults to compare.

NOTES Returns a list, even if given a vector

LISTS | LONGER

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

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

NOTES

 conj works on both lists and vectors, but differently; for vectors, conj adds the value to the *end* of the vector, but with lists, conj adds the value to the *beginning*

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

```
(concat x y & zs) (concat x y) (concat x)
(concat)
```

Returns a lazy seq representing the concatenation of the elements in the supplied colls.

NOTES Returns a list, even if given a vector

(concat [1 2 3] [4 5 6])
(1 2 3 4 5 6)

```
(rest coll)
```

Returns a possibly empty seq of the items after the first. Calls seq on its argument.

NOTES Returns a list, even if given a vector

(rest [1 2 3 4]) (2 3 4)

(drop n coll) (drop n)

Returns a lazy sequence of all but the first n items in coll. Returns a stateful transducer when no collection is provided.

NOTES Returns a list, even if given a vector

(drop 2 [:a :b :c :d :e])
(:c :d :e)

```
(drop-last n s) (drop-last s)
```

Return a lazy sequence of all but the last n (default 1) items in coll

```
Returns a list, even if given a vector

(drop-last 2 [:a :b :c :d :e])
(:a :b :c)
```

```
(take n coll) (take n)
```

Returns a lazy sequence of the first n items in coll, or all items if there are fewer than n. Returns a stateful transducer when no collection is provided.

```
Returns a list, even if given a vector

(take 2 [:a :b :c :d :e])
(:a :b)
```

```
(take-last n coll)
```

Returns a seq of the last n items in coll. Depending on the type of coll may be no better than linear time. For vectors, see also subvec.

```
Returns a list, even if given a vector

(take-last 2 [:a :b :c :d :e])
(:d :e)
```

```
(take-while pred coll) (take-while pred)
```

Returns a lazy sequence of successive items from coll while (pred item) returns true. pred must be free of side-effects. Returns a transducer when no collection is provided.

NOTES Returns a list, even if given a vector

(take-while (fn [x] (< x 3)) [1 2 0 1 2 3 2 1 5])

```
(filter pred coll) (filter pred)
```

Returns a lazy sequence of the items in coll for which (pred item) returns true. pred must be free of side-effects. Returns a transducer when no collection is provided.

NOTES

Returns a list, even if given a vector

```
(filter (fn [x]
(> x 3))
[1 2 3 4 5 6])
(4 5 6)
```

```
(remove pred coll) (remove pred)
```

Returns a lazy sequence of the items in coll for which (pred item) returns false. pred must be free of side-effects. Returns a transducer when no collection is provided.

NOTES

- The opposite of filter
- Returns a list, even if given a vector

```
(map f c1 c2 c3 & colls) (map f c1 c2 c3)
(map f c1 c2) (map f coll) (map f)
```

Returns a lazy sequence 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. Function f should accept number-of-colls arguments. Returns a transducer when no collection is provided.

NOTES Returns a list, even if given a vector

```
(map inc [1 2 3 4 5])
(2 3 4 5 6)
```

```
(map even? [1 2 3 4 5])
(false true false true false)
```

```
(map reverse ["Alice" "Bob" "Cathy" "Donald"])
("ecilA" "boB" "yhtaC" "dlanoD")
```

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

Returns a lazy sequence consisting of the result of applying f to 0 and the first item of coll, followed by applying f to 1 and the second item in coll, etc, until coll is exhausted. Thus function f should accept 2 arguments, index and item. Returns a stateful transducer when no collection is provided.

NOTES

- Like map, but with an additional argument to f that is the index
- Returns a list, even if given a vector

```
(reduce f val coll) (reduce f 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.

NOTES

 Can be used to re-implement most other array functions, like filter, map, etc.

```
(reduce + [1 2 3 4 5])
```

(some pred coll)

Returns the first logical true value of (pred x) for any x in coll, else nil. One common idiom is to use a set as pred, for example this will return: fred if: fred is in the sequence, otherwise nil: (some #{: fred} coll)

USE CASES

- Inside an if or filter, to check if any items in a collection meet a condition
- With a set, to check if collection contains a certain item

EXAMPLES

:x

```
(some even? [1 2 3])
true

(some even? [])
nil

(some #{:x} [:a :b :c :x])
:x

(some #{:y :z :x} [:a :b :c :x :y])
```

(every? pred coll)

Returns true if (pred x) is logical true for every x in coll, else false.



(empty? coll)

Returns true if coll has no items - same as (not (seq coll)). Please use the idiom (seq x) rather than (not (empty? x))



Maps

MAPS | CREATE

{ }

NOTES

keys and values; typically use keywords for keys

HISE CASES

To represent objects with various properties

```
{:a 10
:b 2
:c 33}
```

```
{:email "bob@example.com"
:name "Bob"}
```

```
{"a" 1
"b" 2
"c" 3}
```

```
{:key "value"
 [1 2] 5
 "foo" [2]}
```

```
(get o k not-found) (get o k)
```

Returns value at key, or nil. Optionally, can provide a default value.

USE CASES To get information out of a single map

```
(let [m {"a" 1
    "b" 5}]
  (get m "a"))
```

(select-keys map keyseq)

Returns a map containing only those entries in map whose key is in keys

EXAMPLES

(keys hash-map)

Returns a sequence of the map's keys.

(vals hash-map)

Returns a sequence of the map's values.

(contains? coll v)

Returns true if key is present in the given collection, otherwise returns false. Note that for numerically indexed collections like vectors and arrays, this tests if the numeric key is within the range of indexes. 'contains?' operates constant or logarithmic time; it will not perform a linear search for a value. See also 'some'.

NOTES

 For vectors and lists, you will want to use some or convert to a set with set

USE CASES

In an if or filter, to check if a map or set contains a value

```
(contains? {:a nil :b 2} :a) true
```

```
(assoc coll k v & kvs) (assoc coll k v)
```

Returns a new map with a new key and value assigned

EXAMPLES

```
(dissoc coll k & ks) (dissoc coll k)
(dissoc coll)
```

dissoc[iate]. Returns a new map of the same (hashed/sorted) type, that does not contain a mapping for key(s).

(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})
{:a 1
:b 2}
```

```
(update m k f x y z & more) (update m k f x y z)
(update m k f x y) (update m k f x)
(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 and any supplied args and return the new value, and returns a new structure. If the key does not exist, nil is passed as the old value.

Sets

SETS | CREATE

```
#{ }
```

```
EXAMPLES #{:c :b :a}
```

(set coll)

Converts a input collection into a set

```
USE CASES To remove duplicates from a list or vector
```

(set [1 2 3]) (set [1 2 2 2 3]) #{1 3 2}

SETS | EXAMINE

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

use cases = Rare, could be used instead of contains?

```
(get #{:c :b :a} :x)
nil
```

(contains? coll v)

Returns true if key is present in the given collection, otherwise returns false. Note that for numerically indexed collections like vectors and arrays, this tests if the numeric key is within the range of indexes. 'contains?' operates constant or logarithmic time; it will not perform a linear search for a value. See also 'some'.

For vectors and lists, you will want to use some or convert to a set with set

USE CASES

In an if or filter, to check if a map or set contains a value

(contains? #{1 3 2} 1)

true

false

(contains? #{1 3 2} 4)

SETS | CHANGE

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

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

```
(conj #{:b :a} :c) (conj #{:b :a} :a) (conj #{:b :a} :a) (conj #{:b :a} :a)
```

```
(disj coll k & ks) (disj coll k) (disj coll)
```

disj[oin]. Returns a new set of the same (hashed/sorted) type, that does not contain key(s).

```
(disj #{:c :b :a} :x)

#{:b :a}

(disj #{:c :b :a} :x)
```

```
(union s1 s2 & sets) (union s1 s2) (union s1)
(union)
```

Return a set that is the union of the input sets

```
(clojure.set/union #{:c :b :a} #{:c :b :d})

#{:c :b :d :a}
```

```
(difference s1 s2 & sets) (difference s1 s2)
(difference s1)
```

Return a set that is the first set without elements of the remaining sets

```
(clojure.set/difference #{:c :b :d :a} #{:c :b})
#{:d :a}
```

```
(intersection s1 s2 & sets) (intersection s1 s2)
(intersection s1)
```

Return a set that is the intersection of the input sets

```
(clojure.set/intersection #{:c :b :a} #{:c :b :d})
#{:c :b}
```

```
(subset? set1 set2)
```

Is set1 a subset of set2?

EXAMPLES

```
(clojure.set/subset? #{:b :a} #{:c :b :d :a})
true
```

```
(clojure.set/subset? #{:x} #{:c :b :d :a})
false
```

(superset? set1 set2)

Is set1 a superset of set2?

•

NOTES Like subset? but with reversed arguments

```
(clojure.set/superset? #{:c :b :d :a} #{:b :a})
true
```

```
(clojure.set/superset? #{:c :b :d :a} #{:x})
false
```

Nested Data Structures

NESTED DATA STRUCTURES | CREATE

```
[] { } #{ }

EXAMPLES
```

NESTED DATA STRUCTURES | EXAMINE

```
(get-in m ks not-found) (get-in m ks)
```

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.

```
EXAMPLES
```

```
(assoc-in m [k & 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 will be created.

```
(update-in m [k & ks] f a b c & args)
(update-in m [k & ks] f a b c)
(update-in m [k & ks] f a b)
(update-in m [k & ks] f a)
(update-in m [k & ks] f)
```

'Updates' a value in a nested associative structure, where ks is a sequence of keys and f is a function that will take the old value and any supplied args and return the new value, and returns a new nested structure. If any levels do not exist, hash-maps will be created.

```
(sort-by keyfn comp coll) (sort-by keyfn coll)
```

Returns a sorted sequence of the items in coll, where the sort order is determined by comparing (keyfn item). Comp can be boolean-valued comparison funcion, or a -/O/+ valued comparator. Comp defaults to compare.

(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.

```
(let [contacts [{:name "Alice"
                 :email "alice@example.com"
                 :company "Acme Inc."}
                {:name "Bob"
                 :email "bob@example.com"
                 :company "Evil Co."}
                {:name "Andy"
                 :email "andy@example.com"
                 :company "Acme Inc."}]]
  (group-by :company contacts))
{"Acme Inc." [{:company "Acme Inc."
               :email "alice@example.com"
               :name "Alice"}
              {:company "Acme Inc."
               :email "andv@example.com"
               :name "Andy"}]
"Evil Co." [{:company "Evil Co."
              :email "bob@example.com"
              :name "Bob"}]}
```

```
(filter pred coll) (filter pred)
```

Returns a lazy sequence of the items in coll for which (pred item) returns true. pred must be free of side-effects. Returns a transducer when no collection is provided.

```
(map f c1 c2 c3 & colls) (map f c1 c2 c3)
(map f c1 c2) (map f coll) (map f)
```

Returns a lazy sequence 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. Function f should accept number-of-colls arguments. Returns a transducer when no collection is provided.

```
(reduce f val coll) (reduce f 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.

NESTED DATA STRUCTURES | MULTIPLE

TRANSFORMATIONS

```
(let [contacts [{:name "Alice"
                 :company "Acme Inc."
                 :years-employed 5}
                {:name "Bob"
                 :company "Evil Co."
                 :years-employed 10}
                {:name "Andy"
                 :company "Acme Inc."
                 :years-employed 3}]]
  (->> contacts
       (filter (fn [contact]
                 (= "Acme Inc."
                    (contact :company))))
       (sort-by :years-employed)
       (map :name)))
("Andy" "Alice")
```