

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 | = != |
| COMBINE | and or not |
| BRANCH | if when case cond |

Vectors

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

Strings

| | |
|----------------|--|
| CREATE | " " str |
| EXAMINE | count get |
| CHANGE | reverse join split replace trim |
| CHANGE CASE | upper-case lower-case capitalize |
| TEST | blank? includes? |
| REGEX | #" " re-matches re-find re-seq re-pattern replace |

Lists

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

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.

NOTES

- `\n` is used to represent a new-line

USE CASES

- To represent single characters, words, sentences, multi-line text, etc.

EXAMPLES

"a"

"Bob"

"hello world"

LITERALS | BOOLEANS

true false

EXAMPLES

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

USE CASES

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

EXAMPLES

```
#{1 4 3 2}
```

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

LITERALS | MAPS

`{ }`

EXAMPLES

```
{"a" 100  
"b" 200  
"c" 300}
```

```
{: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.

EXAMPLES

```
+
```

```
max
```

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.

EXAMPLES

```
(def x 10)
```

```
11
```

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.

EXAMPLES

```
(let [x 10]  
  (+ x 5))
```

```
15
```

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

EXAMPLES

```
(+ 1 1 1)
```

```
3
```

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

```
3
```

```
(reduce + [1 1 1])
```

```
3
```

`(- x y & more)` `(- x y)` `(- x)`

Subtract multiple numbers from x.

NOTES

- `(- x)` is equivalent to `(- 0 x)`

EXAMPLES

```
(- 10 1 1)
```

```
8
```

```
(- 10)
```

```
-10
```

`(/ x y & more)` `(/ x y)` `(/ x)`

Divide x by one or more ys.

NOTES ■ `(/ x)` is equivalent to `(/ 1 x)`

EXAMPLES

| |
|-----------------------|
| <code>(/ 12 2)</code> |
| 4 |

| |
|-------------------------|
| <code>(/ 12 2 3)</code> |
| 2 |

| |
|--------------------|
| <code>(/ 5)</code> |
| 0.2 |

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

Multiplies one or more numbers.

NOTES ■ `(*)` returns 1

EXAMPLES

| |
|------------------------|
| <code>(* 2 2 2)</code> |
| 8 |

| |
|--------------------|
| <code>(* 2)</code> |
| 2 |

| |
|------------------|
| <code>(*)</code> |
| 1 |

(inc x)

Returns a number one greater than x.

NOTES

- The same as `(+ x 1)`

USE CASES

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

EXAMPLES

```
(inc 5)
```

```
6
```

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

```
(2 3 4 5)
```

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

```
[3 4 6]
```

(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.

EXAMPLES

```
(dec 5)
```

```
4
```

```
(map dec [1 2 3 4])
```

```
(0 1 2 3)
```

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

```
[1 4 6]
```

(max x y & more) (max x y) (max x)

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

```
(max 5 1 2 6 2)
```

```
6
```

```
(let [x 10]  
  (max 0 x))
```

```
10
```

```
(let [x -5]  
  (max 0 x))
```

```
0
```

```
(apply max [5 1 2 6 2])
```

```
6
```

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

Returns the least of the input numbers.

EXAMPLES

```
(min 1 5 1 -1 0)
```

```
-1
```

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

```
-1
```


NUMBERS | TEST

(even? n)

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

EXAMPLES

| |
|------------|
| (even? 10) |
| true |

| |
|-----------|
| (even? 9) |
| false |

(odd? n)

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

EXAMPLES

| |
|-----------|
| (odd? 10) |
| false |

| |
|----------|
| (odd? 9) |
| true |

NUMBERS | COMPARE

`(== x y & more)` `(== x y)` `(== x)`

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

EXAMPLES

```
(== 1.5 (/ 3 2))
```

```
true
```

```
(== 3 2 1)
```

```
false
```

`(> x y & more)` `(> x y)` `(> x)`

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

EXAMPLES

```
(> 1 2)
```

```
false
```

```
(> 2 2 1)
```

```
false
```

```
(> 3 2 1)
```

```
true
```

`(< x y & more)` `(< x y)` `(< x)`

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

EXAMPLES

```
(< 2 1)
```

```
false
```

```
(< 1 2 3)
```

```
true
```

`(>= 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

```
(>= 1 2)
```

```
false
```

```
(>= 1 1)
```

```
true
```

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

```
true
```

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

```
true
```

`(<= x y & more)` `(<= x y)` `(<= x)`

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.

EXAMPLES

```
(<= 2 1)
```

```
false
```

```
(<= 1 1)
```

```
true
```

```
(<= 1 2 2 3)
```

```
true
```

```
(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).

EXAMPLES

| | | | |
|--------|--------|----------|----------|
| (rand) | (rand) | (rand 5) | (rand 5) |
| 0.002 | 0.812 | 1.4 | 4.3 |

(rand-int n)

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

EXAMPLES

| | | |
|--------------|--------------|--------------|
| (rand-int 3) | (rand-int 3) | (rand-int 3) |
| 2 | 0 | 1 |

Booleans

BOOLEANS | CREATE

true false

EXAMPLES

```
true
```

```
false
```

(boolean x)

Converts an input to true or false

NOTES

- Input values of nil or false will return false. Everything else returns true.

USE CASES

- 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.

EXAMPLES

```
(boolean 3)
```

```
true
```

```
(boolean [])
```

```
true
```

```
(boolean nil)
```

```
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`).

EXAMPLES

```
(= 1 1)
```

```
true
```

```
(= 1 0)
```

```
false
```

```
(= (+ 1 1) 2 (- 3 1))
```

```
true
```

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

```
true
```

(not= x y & more) (not= x y) (not= x)

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)
```

```
false
```

```
(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 true)
```

```
true
```

```
(and true false)
```

```
false
```

```
(and false true)
```

```
false
```


`(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 true false)
```

```
true
```

```
(or nil false)
```

```
false
```

```
(or nil 3)
```

```
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

EXAMPLES

```
(not true)
```

```
false
```

```
(not false)
```

```
true
```

```
(not 1)
```

```
false
```

```
(not (contains? #{1 3 2} 1))
```

```
false
```

BOOLEANS | BRANCH

`(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).

EXAMPLES

```
(if true
  :a
  :b)
```

```
:a
```

```
(if false
  5
  10)
```

```
10
```

```
(if "hello"
  true
  false)
```

```
true
```

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

```
"b"
```

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

```
"different"
```

```
(if true
  (+ 2 3)
  (+ 1 2))
```

```
5
```

(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)

EXAMPLES

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

```
true
```

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

```
nil
```

(case expression & test-constant result-expression)

Takes an expression and a set 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.

EXAMPLES

```
(let [color :g]
  (case color
    :r "red"
    :g "green"
    :b "blue"))
```

"green"

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

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)

EXAMPLES

```
(let [grade 85]
  (cond (>= grade 90) :A
        (>= grade 80) :B
        (>= grade 70) :C
        (>= grade 60) :D
        :else :F))

:B
```

```
(let [grade 50]
  (cond (>= grade 90) :A
        (>= grade 80) :B
        (>= grade 70) :C
        (>= grade 60) :D
        :else :F))

:F
```

Strings

STRINGS | CREATE

" "

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)  
"10"
```

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

STRINGS | EXAMINE

(count coll)

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

1. Also works on strings, arrays, and Maps

EXAMPLES

```
(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.

EXAMPLES

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

```
"o"
```

STRINGS | CHANGE

(reverse s)

Returns `s` with its characters reversed.

EXAMPLES

```
(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.

EXAMPLES

```
(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.

EXAMPLES

```
(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).

EXAMPLES

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

(**trim** s)

Removes whitespace from both ends of string.

EXAMPLES

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

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.

EXAMPLES

```
(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.

EXAMPLES

```
(clojure.string/capitalize "hello world")  
"Hello world"
```

STRINGS | TEST

(blank? s)

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

EXAMPLES

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

```
true
```

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

```
false
```

(includes? s substr)

True if `s` includes `substr`.

EXAMPLES

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

```
true
```

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

```
false
```

STRINGS | REGEX

#" "

EXAMPLES

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

(re-matches re s)

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

EXAMPLES

```
(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.

EXAMPLES

```
(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`.

EXAMPLES

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

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

```
(re-seq #"([a-z]{3})([0-9]{3})" "abc123xyz456")  
(["abc123" "abc" "123"] ["xyz456" "xyz" "456"])
```

(re-pattern s)

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

EXAMPLES

```
(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.

EXAMPLES

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

Vectors

VECTORS | CREATE

[]

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.

EXAMPLES

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

```
:a
```

```
(get [:a :b :c] 2)
```

```
:c
```

```
(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

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

```
[1 2 3 4]
```

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

See `concat` under Lists

VECTORS | SHORTER

(rest coll)

See rest under Lists

(drop n coll) (drop n)

See drop under Lists

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

See drop-last under Lists

(take n coll) (take n)

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.

EXAMPLES

```
(range 1 5)
```

```
(1 2 3 4 5)
```

```
(range 0 10 2)
```

```
(0 2 4 6 8)
```

(repeat n x) (repeat x)

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

EXAMPLES

```
(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

EXAMPLES

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

```
5
```

```
(count [])
```

```
0
```

LISTS | EXTRACT

(first coll)

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

EXAMPLES

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

```
1
```

(last s)

Return the last item in coll, in linear time

EXAMPLES

```
(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.

EXAMPLES

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

```
:b
```

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

```
:a
```

LISTS | REORDER

(shuffle coll)

Return a random permutation of `coll`

NOTES

- Returns a vector, even if given a list

EXAMPLES

```
(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

EXAMPLES

```
(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 `-/0/+` valued comparator. `Comp` defaults to compare.

NOTES

- Returns a list, even if given a vector

EXAMPLES

```
(sort [5 1 2 4 3])
```

```
(1 2 3 4 5)
```

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*

EXAMPLES

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

```
[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

EXAMPLES

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

```
(1 2 3 4 5 6)
```

LISTS | SHORTER

(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

EXAMPLES

```
(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

EXAMPLES

```
(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*

NOTES

- Returns a list, even if given a vector

EXAMPLES

```
(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.

NOTES

- Returns a list, even if given a vector

EXAMPLES

```
(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`.

NOTES

- Returns a list, even if given a vector

EXAMPLES

```
(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

EXAMPLES

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

(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

EXAMPLES

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

```
(4 5 6)
```

```
(filter (fn [n]
         (> (count n) 3))
        ["Alice" "Bob" "Cathy" "Donald"])
```

```
("Alice" "Cathy" "Donald")
```

(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

EXAMPLES

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

```
(1 2 3)
```

LISTS | CHANGE

```
(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

EXAMPLES

```
(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 (fn [n]  
      (get n 2))  
     ["Alice" "Bob" "Cathy" "Donald"])
```

```
("i" "b" "t" "n")
```

`(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

EXAMPLES

```
(map-indexed (fn [index n]
               (get n index))
  (quote ["Alice" "Bob" "Cathy"
          "Donald"]))

("A" "o" "t" "a")
```

`(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.

EXAMPLES

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

```
15
```

```
(let [numbers [3 4 8 3 4 9 7 10 10]]  
  (reduce (fn [m i]  
            (if-let [v (get m i)]  
              (assoc m i (+ 1 v))  
              (assoc m i 1)))  
          {}  
          numbers))
```

```
{3 2  
 4 2  
 7 1  
 8 1  
 9 1}
```

LISTS | 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)

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

```
(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])
```

```
:x
```

(every? pred coll)

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

EXAMPLES

```
(every? even? [1 2 3])
```

```
false
```

```
(every? even? [2 4 6])
```

```
true
```

`(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))`

EXAMPLES

```
(empty? [])
```

```
true
```

```
(empty? [1 2 3])
```

```
false
```

Maps

MAPS | CREATE

{ }

NOTES

- keys and values; typically use keywords for keys

USE CASES

- To represent objects with various properties

EXAMPLES

```
{: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]}
```


MAPS | EXTRACT

(**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

EXAMPLES

```
(get {:foo "bar"  
      :baz "xyz"  
      :baz})
```

"xyz"

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

1

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

nil

```
(let [m {:a 1  
        :b 5}]  
  (get m :c 100))
```

100

(select-keys map keyseq)

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

EXAMPLES

```
(let [m {:a 1
        :b 2
        :c 3}]
  (select-keys m [:a :c]))

{:a 1
 :c 3}
```

```
(let [m {:a 1
        :b 2
        :c 3}]
  (select-keys m [:a :b :x]))

{:a 1
 :b 2}
```

(keys hash-map)

Returns a sequence of the map's keys.

EXAMPLES

```
(let [m {:a 1
        :b 2
        :c 3}]
  (keys m))

(:a :b :c)
```

(vals hash-map)

Returns a sequence of the map's values.

EXAMPLES

```
(let [m {:a 1  
        :b 2  
        :c 3}]  
  (vals m))  
  
(1 2 3)
```

(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

EXAMPLES

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

`true`

```
(contains? {:a 1
            :b 2}
           :c)
```

`false`

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

`true`

MAPS | CHANGE

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

Returns a new map with a new key and value assigned

EXAMPLES

```
(let [m {:a 1
        :b 5}]
  (assoc m :c 9))

{:a 1
 :b 5
 :c 9}
```

```
(let [m {:a 1
        :b 5}]
  (assoc m :b 9))

{:a 1
 :b 9}
```

(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).

EXAMPLES

```
(let [m {:a 1
        :b 5}]
  (dissoc m :a))

{:b 5}
```

```
(let [m {:a 1
        :b 5}]
  (dissoc m :c))

{:a 1
 :b 5}
```

(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.

EXAMPLES

```
(let [a {:i 1
        :j 2
        :k 3}
      b {:x 10
        :y 20
        :z 30}]
  (merge a b))

{:i 1
 :j 2
 :k 3
 :x 10
 :y 20
 :z 30}
```

```
(let [a {:i 1
        :j 2
        :k 3}
      b {:i 10
        :z 30}]
  (merge a b))

{:i 10
 :j 2
 :k 3
 :z 30}
```

```
(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.

EXAMPLES

```
(let [m {:a 1
        :b 2
        :c 3}]
  (update m :a dec))
```

```
{:a 0
 :b 2
 :c 3}
```

```
(let [m {:a 1
        :b 2
        :c 3}]
  (update m
    :x
    (fn [v]
      (if v
        (inc v)
        1)))))
```

```
{:a 1
 :b 2
 :c 3
 :x 1}
```

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

EXAMPLES

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

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


SETS | EXAMINE

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

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

USE CASES ■ Rare, could be used instead of contains?

EXAMPLES

| |
|----------------------|
| (get #{:c :b :a} :a) |
| :a |

| |
|----------------------|
| (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'.

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

EXAMPLES

| |
|------------------------|
| (contains? #{1 3 2} 1) |
| true |

| |
|------------------------|
| (contains? #{1 3 2} 4) |
| false |

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.

EXAMPLES

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

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

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

```
#{:b :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).

EXAMPLES

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

```
#{:b :a}
```

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

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

SETS | OPS

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

Return a set that is the union of the input sets

EXAMPLES

```
(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

EXAMPLES

```
(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

EXAMPLES

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

SETS | TEST

(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

EXAMPLES

```
(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

```
(let [contacts [{:name "Alice"
                  :email "alice@example.com"}
                {:name "Bob"
                  :email "bob@example.com"}]]
  (get-in contacts [1 :name]))

"Bob"
```

NESTED DATA STRUCTURES | CHANGE

(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.

EXAMPLES

```
(let [contacts [{:name "Alice"
                 :email "alice@example.com"}
               {:name "Bob"
                 :email "bob@example.com"}]]
  (assoc-in contacts [1 :name] "Robert"))

[{:email "alice@example.com"
  :name "Alice"}
 {:email "bob@example.com"
  :name "Robert"}]
```

`(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.

EXAMPLES

```
(let [contacts [{:name "Alice"
                 :email "alice@example.com"}
               {:name "Bob"
                 :email "bob@example.com"}]]
  (update-in contacts
    [1 :name]
    string/upper-case))

[{:email "alice@example.com"
  :name "Alice"}
 {:email "bob@example.com"
  :name "BOB"}]
```

(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 function, or a `-/0/+` valued comparator. `Comp` defaults to `compare`.

EXAMPLES

```
(let [contacts [{:name "Alice"
                  :email "alice@example.com"}
               {:name "Bob"
                  :email "bob@example.com"}
               {:name "Andy"
                  :email "andy@example.com"}]]
  (sort-by :name contacts))
```

```
({:email "alice@example.com"
  :name "Alice"}
 {:email "andy@example.com"
  :name "Andy"}
 {:email "bob@example.com"
  :name "Bob"})
```

```
(let [contacts [{:name "Alice"
                  :email "alice@example.com"}
               {:name "Bob"
                  :email "bob@example.com"}
               {:name "Andy"
                  :email "andy@example.com"}]]
  (sort-by (fn [contact]
             (count (:name contact)))
           contacts))
```

```
({:email "bob@example.com"
  :name "Bob"}
 {:email "andy@example.com"
  :name "Andy"}
 {:email "alice@example.com"
  :name "Alice"})
```


(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`.

EXAMPLES

```
(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 "andy@example.com"
                :name "Andy"}]}
"Evil Co." [{:company "Evil Co."
              :email "bob@example.com"
              :name "Bob"}]}
```

```
(let [contacts [{:name "Alice"
                  :email "alice@example.com"}
                {:name "Bob"
                  :email "bob@example.com"}
                {:name "Andy"
                  :email "andy@example.com"}]]
  (group-by (fn [contact]
              (first (contact :name)))
            contacts))

{"A" [{:email "alice@example.com"
        :name "Alice"}
      {:email "andy@example.com"
        :name "Andy"}]}
"B" [{: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.

EXAMPLES

```
(let [contacts [{:name "Alice"
                  :company "Acme Inc."}
                {:name "Bob"
                  :company "Evil Co."}
                {:name "Andy"
                  :company "Acme Inc."}]]
  (filter (fn [contact]
            (= (contact :company) "Acme Inc."))
    contacts))

({:company "Acme Inc."
  :name "Alice"}
 {:company "Acme Inc."
  :name "Andy"})
```

(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.

EXAMPLES

```
(let [contacts [{:name "Alice"
                  :company "Acme Inc."}
                {:name "Bob"
                  :company "Evil Co."}
                {:name "Andy"
                  :company "Acme Inc."}]]
  (map :name contacts))

("Alice" "Bob" "Andy")
```

(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.

EXAMPLES

```
(let [contacts [{:name "Alice"
                  :company "Acme Inc."}
               {:name "Bob"
                  :company "Evil Co."}
               {:name "Andy"
                  :company "Acme Inc."}]]
  (reduce (fn [memo contact]
            (if (memo (contact :company))
                (update memo (contact :company) inc)
                (assoc memo (contact :company) 1)))
          {}
          contacts))

{"Acme Inc." 2
 "Evil Co." 1}
```

NESTED DATA STRUCTURES | MULTIPLE TRANSFORMATIONS

EXAMPLES

```
(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")
```