



FREE eBook

LEARNING clojure

Free unaffiliated eBook created from
Stack Overflow contributors.

#clojure

Table of Contents

About.....	1
Chapter 1: Getting started with clojure.....	2
Remarks.....	2
Versions.....	2
Examples.....	3
Installation and Setup.....	3
Option 1: Leiningen.....	3
Linux.....	3
OS X.....	3
Install with Homebrew.....	3
Install with MacPorts.....	4
Windows.....	4
Option 2: Official Distribution.....	4
Option 3: Boot.....	4
"Hello, world!" in the REPL.....	4
Create a new application.....	5
"Hello, world!" using Boot.....	5
Create a new application (with boot).....	6
Chapter 2: Atom.....	7
Introduction.....	7
Examples.....	7
Define an atom.....	7
Read an atom's value.....	7
Update an atom's value.....	7
Chapter 3: clj-time.....	9
Introduction.....	9
Examples.....	9
Creating a Joda Time.....	9
Getting Day Month Year Hour Minute Second from your date-time.....	9
Comparing two date-time.....	9

Checking whether a time is within a time interval.....	10
Adding joda date-time from other time types.....	10
Adding date-time to other date-times.....	10
Chapter 4: Clojure destructuring	12
Examples.....	12
Destructuring a vector.....	12
Destructuring a map.....	12
Destructuring remaining elements into a sequence.....	13
Destructuring nested vectors.....	13
Destructuring a map with default values.....	13
Destructuring params of a fn.....	14
Converting the rest of a sequence to a map.....	14
Overview.....	14
Tips:.....	15
Destructuring and binding to keys' name.....	15
Destructuring and giving a name to the original argument value.....	16
Chapter 5: clojure.core	17
Introduction.....	17
Examples.....	17
Defining functions in clojure.....	17
Assoc - updating map/vector values in clojure.....	17
Comparison Operators in Clojure.....	17
Dissoc - disassociating a key from a clojure map.....	18
Chapter 6: clojure.spec	19
Syntax.....	19
Remarks.....	19
Examples.....	19
Using a predicate as a spec.....	19
fdef: writing a spec for a function.....	19
Registering a spec.....	20
clojure.spec/and & clojure.spec/or.....	20
Record specs.....	21

Map specs.....	21
Collections.....	22
Sequences.....	24
Chapter 7: clojure.test.....	25
Examples.....	25
is.....	25
Grouping related tests with the testing macro.....	25
Defining a test with deftest.....	25
are.....	26
Wrap each test or all tests with use-fixtures.....	26
Running tests with Leiningen.....	27
Chapter 8: Collections and Sequences.....	28
Syntax.....	28
Examples.....	28
Collections.....	28
Lists.....	28
Sequences.....	31
Vectors.....	35
Sets.....	39
Maps.....	41
Chapter 9: core.async.....	47
Examples.....	47
basic channel operations: creating, putting, taking, closing, and buffers.....	47
Creating channels with chan.....	47
Putting values into channels with >!! and >!.....	47
Taking values from channels with <!!.....	48
Closing channels.....	48
Asynchronous puts with put!.....	49
Asynchronous takes with take!.....	49
Using dropping and sliding buffers.....	49
Chapter 10: Emacs CIDER.....	51
Introduction.....	51

Examples.....	51
Function Evaluation.....	51
Pretty Print.....	51
Chapter 11: File Operations.....	53
Examples.....	53
Overview.....	53
Notes:.....	53
Chapter 12: Functions.....	54
Examples.....	54
Defining Functions.....	54
Functions are defined with five components:.....	54
Parameters and Arity.....	54
Arity.....	55
Defining Variadic Functions.....	55
Defining anonymous functions.....	56
Full Anonymous Function Syntax.....	56
Shorthand Anonymous Function Syntax.....	56
When To Use Each.....	56
Supported Syntax.....	56
Chapter 13: Getting started with web development.....	58
Examples.....	58
Create new Ring application with http-kit.....	58
New web application with Luminus.....	58
Web Servers.....	59
databases.....	59
miscellaneous.....	59
Chapter 14: Java interop.....	61
Syntax.....	61
Remarks.....	61
Examples.....	61
Calling an instance method on a Java object.....	61

Referencing an instance field on a Java Object.....	61
Creating a new Java object.....	61
Calling a static method.....	62
Calling a Clojure function from Java.....	62
Chapter 15: Macros.....	63
Syntax.....	63
Remarks.....	63
Examples.....	63
Simple Infix Macro.....	63
Syntax quoting and unquoting.....	64
Chapter 16: Parsing logs with clojure.....	65
Examples.....	65
Parse a line of log with record & regex.....	65
Chapter 17: Pattern Matching with core.match.....	66
Remarks.....	66
Examples.....	66
Matching Literals.....	66
Matching a Vector.....	66
Matching a Map.....	66
Matching a literal symbol.....	66
Chapter 18: Performing Simple Mathematical Operations.....	68
Introduction.....	68
Remarks.....	68
Examples.....	68
Math Examples.....	68
Chapter 19: Setting up your development environment.....	69
Examples.....	69
Light Table.....	69
Emacs.....	70
Atom.....	70
IntelliJ IDEA + Cursive.....	70
Spacemacs + CIDER.....	71

Vim.....	72
Chapter 20: Threading Macros.....	73
Introduction.....	73
Examples.....	73
Thread Last (->>).....	73
Thread First (->).....	73
Thread as (as->).....	73
Chapter 21: Transducers.....	75
Introduction.....	75
Remarks.....	75
Examples.....	75
Small transducer applied to a vector.....	75
Applying transducers.....	76
Creating/Using Transducers.....	76
Chapter 22: Truthiness.....	78
Examples.....	78
Truthiness.....	78
Booleans.....	78
Chapter 23: Vars.....	79
Syntax.....	79
Remarks.....	79
Examples.....	79
Types of Variables.....	79
Credits.....	80

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [clojure](#)

It is an unofficial and free clojure ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official clojure.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with clojure

Remarks



Clojure is a dynamically-typed general-purpose programming language with Lisp syntax.

Its features support the functional style of programming with first-class functions and immutable values by default. Using reassignable variables is not as easy in Clojure as in many mainstream languages, since variables have to be created and updated like container objects. This encourages use of pure values that will stay the way they were at the moment they were last seen. This typically makes code much more predictable, testable and concurrency-capable. This works for collections too, since Clojure's built-in data structures are persistent.

For performance, Clojure supports type-hinting to eliminate unnecessary reflection where possible. Also, groups of changes to persistent collections can be done to *transient* versions, reducing the amount of objects involved. This is not necessary most of the time, since persistent collections fast to copy since they share most of their data. Their performance guarantees are not far from their mutable counterparts.

Among other features, Clojure also has:

- software transactional memory (STM)
- several concurrency primitives not involving manual locking (atom, agent)
- composable sequence transformers (transducers),
- functional tree manipulation facilities (zippers)

Due to its simple syntax and high extensibility (via macros, implementation of core interfaces and reflection), some commonly-seen language features can be added to Clojure with libraries. For instance, `core.typed` brings a static type checker, `core.async` brings simple channel-based concurrency mechanisms, `core.logic` brings logic programming.

Designed as a hosted language, it can interoperate with the platform it runs on. While the primary target is JVM and the entire ecosystem behind Java, alternative implementations can run in other environments too, such as ClojureCLR running on the Common Language Runtime or ClojureScript running on JavaScript runtimes (including web browsers). While alternative implementations may lack some of the functionality from the JVM version, they are still considered one family of languages.

Versions

Version	Change log	Release Date
1.8	Latest change log	2016-01-19
1.7	Change log 1.7	2015-06-30
1.6	Change log 1.6	2014-03-25
1.5.1	Change log 1.5.1	2013-03-10
1.4	Change log 1.4	2012-04-15
1.3	Change log 1.3	2011-09-23
1.2.1		2011-03-25
1.2		2010-08-19
1.1		2010-01-04
1.0		2009-05-04

Examples

Installation and Setup

Option 1: [Leiningen](#)

Requires JDK 6 or newer.

The easiest way to get started with Clojure is to download and install Leiningen, the de facto standard tool to manage Clojure projects, then run `lein repl` to open a [REPL](#).

Linux

```
curl https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein > ~/bin/lein
export PATH=$PATH:~/bin
chmod 755 ~/bin/lein
```

OS X

Follow Linux steps above or install with macOS package managers.

Install with [Homebrew](#)

```
brew install leiningen
```

Install with MacPorts

First install Clojure

```
sudo port -R install clojure
```

Install Leiningen, a build tool for Clojure

```
sudo port -R install leiningen
```

```
lein self-install
```

Windows

See [the official documentation](#).

Option 2: Official Distribution

Requires JRE 6 or newer.

Clojure releases are published as simple [JAR](#) files to be run on the JVM. This is what typically happens inside the Clojure build tools.

1. Go to <http://clojure.org> and download the latest Clojure archive
2. Extract the downloaded [ZIP](#) file into a directory of your choice
3. Run `java -cp clojure-1.8.0.jar clojure.main` in that directory

You may have to substitute the `clojure-1.8.0.jar` in that command for the name of the JAR file that you actually downloaded.

For a better command-line REPL experience (e.g. cycling through your previous commands), you might want to install [rlwrap](#): `rlwrap java -cp clojure-1.8.0.jar clojure.main`

Option 3: Boot

Requires JDK 7 or newer.

Boot is a multi-purpose Clojure build tool. Understanding it requires some knowledge of Clojure, so it may not be the best option for beginners. See [the website](#) (click *Get Started* there) for installation instructions.

Once it's installed and in your `PATH`, you can run `boot repl` anywhere to start a Clojure REPL.

"Hello, world!" in the REPL

The Clojure community puts a large emphasis on interactive development, so quite a lot of interaction with Clojure happens within a [REPL \(read-eval-print-loop\)](#). When you input an expression into it, Clojure **reads** it, **evaluates** it, and **prints** the result of the evaluation, all in a **loop**.

You should be able to launch a Clojure REPL by now. If you don't know how, follow the **Installation and Setup** section in this topic. Once you've got it running, type the following into it:

```
(println "Hello, world!")
```

Then hit `Enter`. This should print out `Hello, world!`, followed by this expression's return value, `nil`.

If you want to run some clojure instantly, try online REPL. For example <http://www.tryclj.com/>.

Create a new application

After following the instructions above and installing Leiningen, start a new project by running:

```
lein new <project-name>
```

This will setup a Clojure project with the default Leiningen template within the `<project-name>` folder. There are several templates for Leiningen, which affect the project's structure. Most commonly is the template "app" used, which adds a main-function and prepares the project to be packed into a jar-file (which the main-function being the entrypoint of the application). This can be achieved with this by running:

```
lein new app <project-name>
```

Assuming you used the app-template to create a new application, you can test that everything was setup correctly, by entering the created directory and running the application using:

```
lein run
```

If you see `Hello, World!` on your console, you're all set and ready to start building your application.

You can pack this simple application into two jar-files with the following command:

```
lein uberjar
```

"Hello, world!" using Boot

Note: you need to install Boot before trying this example out. See the **Installation and Setup** section if you haven't installed it yet.

Boot allows making executable Clojure files using [shebang](#) (`#!`) line. Place the following text into a file of your choice (this example assumes it's in the "current working directory" and is named `hello.clj`).

```
#!/usr/bin/env boot

(defn -main [& args]
  (println "Hello, world!"))
```

Then mark it as executable (if applicable, typically by running `chmod +x hello.clj`).
...and run it (`./hello.clj`).

The program should output "Hello, world!" and finish.

Create a new application (with boot)

```
boot -d seancorfield/boot-new new -t app -n <appname>
```

This command will tell boot to grab the task `boot-new` from <https://github.com/seancorfield/boot-new> and execute the task with the `app` template (see link for other templates). The task will create a new directory called `<appname>` with a typical Clojure application structure. See the generated README for more information.

To run the application: `boot run`. Other commands are specified in `build.boot` and described in the README.

Read *Getting started with clojure* online: <https://riptutorial.com/clojure/topic/827/getting-started-with-clojure>

Chapter 2: Atom

Introduction

An atom in Clojure is a variable that can be changed throughout your program (namespace). Because most data types in Clojure are immutable (or unchangeable) - you can't change a number's value without redefining it - atoms are essential in Clojure programming.

Examples

Define an atom

To define an atom, use an ordinary `def`, but add an `atom` function before it, like so:

```
(def counter (atom 0))
```

This creates an `atom` of value `0`. Atoms can be of any type:

```
(def foo (atom "Hello"))  
(def bar (atom ["W" "o" "r" "l" "d"]))
```

Read an atom's value

To read an atom's value, simply put the name of the atom, with a `@` before it:

```
@counter ; => 0
```

A bigger example:

```
(def number (atom 3))  
(println (inc @number))  
;; This should output 4
```

Update an atom's value

There are two commands to change an atom, `swap!` and `reset!`. `swap!` is given commands, and changes the atom based on its current state. `reset!` changes the atom's value completely, regardless of what the original atom's value was:

```
(swap! counter inc) ; => 1  
(reset! counter 0) ; => 0
```

This example outputs the first 10 powers of 2 using atoms:

```
(def count (atom 0))
```

```
(while (< @atom 10)
  (swap! atom inc)
  (println (Math/pow 2 @atom)))
```

Read Atom online: <https://riptutorial.com/clojure/topic/7519/atom>

Chapter 3: clj-time

Introduction

This documents deals with how to manipulate Date and Time in clojure.

To use this in your application go to your project.clj file and include [clj-time "<version_number>"] in your :dependencies section.

Examples

Creating a Joda Time

```
(clj-time/date-time 2017 1 20)
```

Gives you a Joda time of 20th Jan 2017 at 00:00:00.

Hours, minutes and seconds can also be specified as

```
(clj-time/date-time year month date hour minute second millisecond)
```

Getting Day Month Year Hour Minute Second from your date-time

```
(require '[clj-time.core :as t])

(def example-time (t/date-time 2016 12 5 4 3 27 456))

(t/year example-time) ;; 2016
(t/month example-time) ;; 12
(t/day example-time) ;; 5
(t/hour example-time) ;; 4
(t/minute example-time) ;; 3
(t/second example-time) ;; 27
```

Comparing two date-time

```
(require '[clj-time.core :as t])

(def date1 (t/date-time 2016 12 5))
(def date2 (t/date-time 2016 12 6))

(t/equal? date1 date2) ;; false
(t/equal? date1 date1) ;; true

(t/before? date1 date2) ;; true
(t/before? date2 date1) ;; false

(t/after? date1 date2) ;; false
(t/after? date2 date1) ;; true
```


Checking whether a time is within a time interval

This function tells whether a given time lies within a a given time interval.

```
(require '[clj-time.core :as t])

(def date1 (t/date-time 2016 11 5))
(def date2 (t/date-time 2016 12 5))

(def test-date1 (t/date-time 2016 12 20))
(def test-date2 (t/date-time 2016 11 15))

(t/within? (t/interval date1 date2) test-date1) ;; false
(t/within? (t/interval date1 date2) test-date2) ;; true
```

The interval function is used is **exclusive**, which means that is does not include the second argument of the function within the interval. As an example:

```
(t/within? (t/interval date1 date2) date2) ;; false
(t/within? (t/interval date1 date2) date1) ;; true
```

Adding joda date-time from other time types

The `clj-time.coerce` library can help converting other date-time formats to joda time format (`clj-time.core/date-time`). The other formats include **Java long** format, **String**, **Date**, **SQL Date**.

To convert time from other time formats, include the library and use the `from-` function, e.g.

```
(require '[clj-time.coerce :as c])

(def string-time "1990-01-29")
(def epoch-time 633571200)
(def long-time 633551400)

(c/from-string string-time) ;; #<DateTime 1990-01-29T00:00:00.000Z>
(c/from-epoch epoch-time) ;; #<DateTime 1990-01-29T00:00:00.000Z>
(c/from-long 633551400) ;; #<DateTime 1990-01-29T00:00:00.000Z>
```

Adding date-time to other date-times

`clj-time` gives us option to add/subtract date-times to other date-times. The date times added subtracted should be in form of days, months, years, hours etc.

```
(require '[clj-time.core :as t])

(def example-date (t/date-time 2016 1 1)) ;; #<DateTime 2016-01-01T00:00:00.000Z>

;; Addition
(t/plus example-date (t/months 1)) ;; #<DateTime 2016-02-01T00:00:00.000Z>
(t/plus example-date (t/years 1)) ;; #<DateTime 2017-01-01T00:00:00.000Z>

;; Subtraction
(t/minus example-date (t/days 1)) ;; #<DateTime 2015-12-31T00:00:00.000Z>
```

```
(t/minus example-date (t/hours 12))      ;; #<DateTime 2015-12-31T12:00:00.000Z>
```

Read clj-time online: <https://riptutorial.com/clojure/topic/9127/clj-time>

Chapter 4: Clojure destructuring

Examples

Destructuring a vector

Here's how you can destructure a vector:

```
(def my-vec [1 2 3])
```

Then, for example within a `let` block, you can extract values from the vector very succinctly as follows:

```
(let [[x y] my-vec]
  (println "first element:" x ", second element: " y))
;; first element: 1 , second element: 2
```

Destructuring a map

Here's how you can destructure a map:

```
(def my-map {:a 1 :b 2 :c 3})
```

Then, for example, within a `let` block you can extract values from the map very succinctly as follows:

```
(let [{x :a y :c} my-map]
  (println ":a val:" x ", :c val: " y))
;; :a val: 1 , :c val: 3
```

Notice that the values being extracted in each mapping are on the left and the keys they are associated with are on the right.

If you want to destructure values to bindings with the same names as the keys you can use this shortcut:

```
(let [{:keys [a c]} my-map]
  (println ":a val:" a ", :c val: " c))
;; :a val: 1 , :c val: 3
```

If your keys are strings you can use almost the same structure:

```
(let [{:strs [foo bar]} {"foo" 1 "bar" 2}]
  (println "FOO:" foo "BAR: " bar ))
;; FOO: 1 BAR: 2
```

And similarly for symbols:

```
(let [{:syms [foo bar]} {'foo 1 'bar 2}]
  (println "FOO:" foo "BAR:" bar))
;; FOO: 1 BAR: 2
```

If you want to destructure a nested map, you can nest binding-forms explained above:

```
(def data
  {:foo {:a 1
         :b 2}
   :bar {:a 10
         :b 20}})

(let [{{:keys [a b]} :foo
      {a2 :a b2 :b} :bar} data]
  [a b a2 b2])
;; => [1 2 10 20]
```

Destructuring remaining elements into a sequence

Let's say you have a vector like so:

```
(def my-vec [1 2 3 4 5 6])
```

And you want to extract the first 3 elements and get the remaining elements as a sequence. This can be done as follows:

```
(let [[x y z & remaining] my-vec]
  (println "first:" x ", second:" y "third:" z "rest:" remaining))
;= first: 1 , second: 2 third: 3 rest: (4 5 6)
```

Destructuring nested vectors

You can destructure nested vectors:

```
(def my-vec [[1 2] [3 4]])

(let [[[a b][c d]] my-vec]
  (println a b c d))
;; 1 2 3 4
```

Destructuring a map with default values

Sometimes you want to destructure key under a map which might not be present in the map, but you want a default value for the destructured value. You can do that this way:

```
(def my-map {:a 3 :b 4})
(let [{a :a
      b :b
      :keys [c d]}
```

```

      :or {a 1
           c 2}} my-map]
(println a b c d))
;= 3 4 2 nil

```

Destructuring params of a fn

Destructuring works in many places, as well as in the param list of an fn:

```

(defn my-func [_ a b]
  (+ a b))

(my-func [1 2 3]) ;= 5
(my-func (range 5)) ;= 3

```

Destructuring also works for the `& rest` construct in the param list:

```

(defn my-func2 [& [_ a b]]
  (+ a b))

(my-func2 1 2 3) ;= 5
(apply my-func2 (range 5)) ;= 3

```

Converting the rest of a sequence to a map

Destructuring also gives you the ability to interpret a sequence as a map:

```

(def my-vec [:a 1 :b 2])
(def my-lst '("smthg else" :c 3 :d 4))

(let [[& {:keys [a b]}] my-vec
      [s & {:keys [c d]} my-lst]
  (+ a b c d)) ;= 10

```

It is useful for defining functions with **named parameters**:

```

(defn my-func [a b & {:keys [c d] :or {c 3 d 4}}]
  (println a b c d))

(my-func 1 2) ;= 1 2 3 4
(my-func 3 4 :c 5 :d 6) ;= 3 4 5 6

```

Overview

Destructuring allows you to extract data from various objects into distinct variables. In each example below, each variable is assigned to its own string (a="a", b="b", &c.)

Type	Example	Value of <code>data</code> / <i>comment</i>
<code>vec</code>	<code>(let [[a b c] data ...])</code>	<code>["a" "b" "c"]</code>

Type	Example	Value of <code>data</code> / <i>comment</i>
nested <code>vec</code>	<code>(let [[[a b] [c d]] data ...])</code>	<code>[["a" "b"] ["c" "d"]]</code>
<code>map</code>	<code>(let [{:a :a :b :b :c :c} data ...])</code>	<code>{:a "a" :b "b" :c "c"}</code>
— alternative:	<code>(let [{:keys [a b c]} data ...])</code>	<i>When variables are named after the keys.</i>

Tips:

- Default values can be provided using `:or`, otherwise the default is `nil`
- Use `& rest` to store a seq of any extra values in `rest`, otherwise the extra values are ignored
- A common and useful use of destructuring is for function parameters
- You can assign unwanted parts to a throw-away variable (conventionally: `_`)

Destructuring and binding to keys' name

Sometimes when destructuring maps, you would like to bind the destructured values to their respective key name. Depending on the granularity of the data structure, using the *standard* destructuring scheme may be a little bit *verbose*.

Let's say, we have a map based record like so :

```
(def john {:lastname "McCarthy" :firstname "John" :country "USA"})
```

we would normally destructure it like so :

```
(let [{:lastname :lastname :firstname :firstname :country :country} john]
  (str firstname " " lastname ", " country))
;; "John McCarthy, USA"
```

here, the data structure is quite simple with only 3 slots (*firstname*, *lastname*, *country*) but imagine how cumbersome it would be if we had to repeat all the key names twice for more granular data structure (having way more slots than just 3).

Instead, a better way of handling this is by using `:keys` (since our keys are *keywords* here) and selecting the key name we would like to bind to like so :

```
(let [{:keys [firstname lastname country]} john]
  (str firstname " " lastname ", " country))
;; "John McCarthy, USA"
```

The same *intuitive logic* applies for other key types like *symbols* (using `:syms`) and plain old *strings* (using `:strs`)

```
;; using strings as keys
(def john {"lastname" "McCarthy" "firstname" "John" "country" "USA"})
;; #'user/john
```

```
;; destructuring string-keyed map
(let [{:strs [lastname firstname country]} john]
  (str firstname " " lastname ", " country))
;"John McCarthy, USA"

;; using symbols as keys
(def john {'lastname "McCarthy" 'firstname "John" 'country "USA"})

;; destructuring symbol-keyed map
(let [{:syms [lastname firstname country]} john]
  (str firstname " " lastname ", " country))
;"John McCarthy, USA"
```

Destructuring and giving a name to the original argument value

```
(defn print-some-items
  [[a b :as xs]]
  (println a)
  (println b)
  (println xs))

(print-some-items [2 3])
```

This example prints the output

```
2
3
[2 3]
```

The argument is destructured and the items `2` and `3` are assigned to the symbols `a` and `b`. The original argument, the entire vector `[2 3]`, is also assigned to the symbol `xs`.

Read Clojure destructuring online: <https://riptutorial.com/clojure/topic/1786/clojure-destructuring>

Chapter 5: clojure.core

Introduction

This document gives various basic functionalities offered by clojure. There is no explicit dependency needed for this and comes as a part of org.clojure.

Examples

Defining functions in clojure

```
(defn x [a b]
  (* a b)) ;; public function

=> (x 3 2) ;; 6
=> (x 0 9) ;; 0

(defn- y [a b]
  (+ a b)) ;; private function

=> (x (y 1 2) (y 2 3)) ;; 15
```

Assoc - updating map/vector values in clojure

When applied on a map, returns a new map with new or updated key val pairs.

It can be used to add new information in existing map.

```
(def userData {:name "Bob" :userID 2 :country "US"})
(assoc userData :age 27) ;; { :name "Bob" :userID 2 :country "US" :age 27}
```

It replaces old information value if existing key is supplied.

```
(assoc userData :name "Fred") ;; { :name "Fred" :userID 2 :country "US" }
(assoc userData :userID 3 :age 27) ;; { :name "Bob" :userID 3 :country "US" :age 27}
```

It can also be used on a vector for replacing value at the specified index.

```
(assoc [3 5 6 7] 2 10) ;; [3 5 10 7]
(assoc [1 2 3 4] 6 6) ;; java.lang.IndexOutOfBoundsException
```

Comparison Operators in Clojure

Comparisons are functions in clojure. What that means in $(2 > 1)$ is $(> 2 1)$ in clojure. Here are all the comparison operators in clojure.

1. Greater Than


```
(> 2 1) ;; true
(> 1 2) ;; false
```

2. Less Than

```
(< 2 1) ;; false
```

3. Greater Than or Equal To

```
(>= 2 1) ;; true
(>= 2 2) ;; true
(>= 1 2) ;; false
```

4. Less Than or Equal To

```
(<= 2 1) ;; false
(<= 2 2) ;; true
(<= 1 2) ;; true
```

5. Equal To

```
(= 2 2) ;; true
(= 2 10) ;; false
```

6. Not Equal To

```
(not= 2 2) ;; false
(not= 2 10) ;; true
```

Dissoc - disassociating a key from a clojure map

This returns a map without the key-value pairs for the keys mentioned in the function argument. It can be used to remove information from existing map.

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

It can also be used for dissocing multiple keys as:

```
(dissoc {:a 1 :b 2 :c 3} :a :b) ;; {:c 3}
```

Read [clojure.core](https://riptutorial.com/clojure/topic/9585/clojure-core) online: <https://riptutorial.com/clojure/topic/9585/clojure-core>

Chapter 6: clojure.spec

Syntax

- `::` is a shorthand a namespace-qualified keyword. E.g. if we are in the namespace `user`: `::foo` is a shorthand for `:user/foo`
- `#:` or `#` - map-literal syntax for qualifying keys in a map by a namespace

Remarks

Clojure `spec` is a new specification/contracts library for clojure available as of version 1.9.

Specs are leveraged in a number of ways including being included in documentation, data validation, generating data for testing and more.

Examples

Using a predicate as a spec

Any predicate function can be used as a spec. Here's a simple example:

```
(clojure.spec/valid? odd? 1)
;;=> true

(clojure.spec/valid? odd? 2)
;;=> false
```

the `valid?` function will take a spec and a value and return true if the value conforms to the spec and false otherwise.

One other interesting predicate is set membership:

```
(s/valid? #{:red :green :blue} :red)
;;=> true
```

fdef: writing a spec for a function

Let's say we have the following function:

```
(defn nat-num-count [nums] (count (remove neg? nums)))
```

We can write a spec for this function by defining a function spec of the same name:

```
(clojure.spec/fdef nat-num-count
  :args (s/cat :nums (s/coll-of number?))
  :ret integer?)
```

```
:fn #(<= (:ret %) (-> % :args :nums count)))
```

`:args` takes a regex spec which describes the sequence of arguments by a keyword label corresponding to the argument name and a corresponding spec. The reason the spec required by `:args` is a regex spec is to support multiple arities for a function. `:ret` specifies a spec for the return value of the function.

`:fn` is a spec which constrains the relationship between the `:args` and the `:ret`. It is used as a property when run through `test.check`. It is called with a single argument: a map with two keys: `:args` (the conformed arguments to the function) and `:ret` (the function's conformed return value).

Registering a spec

In addition to predicates functioning as specs, you can register a spec globally using `clojure.spec/def`. `def` requires that a spec being registered is named by a namespace-qualified keyword:

```
(clojure.spec/def ::odd-nums odd?)
;;=> :user/odd-nums

(clojure.spec/valid? ::odd-nums 1)
;;=> true
(clojure.spec/valid? ::odd-nums 2)
;;=> false
```

Once registered, a spec can be referenced globally anywhere in a Clojure program.

The `::odd-nums` syntax is a shorthand for `:user/odd-nums`, assuming we are in the `user` namespace. `::` will qualify the symbol it precedes with the current namespace.

Rather than pass in the predicate, we can pass in the spec name to `valid?`, and it will work the same way.

clojure.spec/and & clojure.spec/or

`clojure.spec/and` & `clojure.spec/or` can be used to create more complex specs, using multiple specs or predicates:

```
(clojure.spec/def ::pos-odd (clojure.spec/and odd? pos?))

(clojure.spec/valid? ::pos-odd 1)
;;=> true

(clojure.spec/valid? ::pos-odd -3)
;;=> false
```

`or` works similarly, with one significant difference. When defining an `or` spec, you must tag each possible branch with a keyword. This is used in order to provide specific branches which fail in error messages:

```
(clojure.spec/def ::big-or-small (clojure.spec/or :small #(< % 10) :big #(> % 100)))

(clojure.spec/valid? ::big-or-small 1)
;;=> true

(clojure.spec/valid? ::big-or-small 150)
;;=> true

(clojure.spec/valid? ::big-or-small 20)
;;=> false
```

When conforming a spec using `or`, the applicable spec will be returned which made the value conform:

```
(clojure.spec/conform ::big-or-small 5)
;; => [:small 5]
```

Record specs

You can spec a record as follows:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(defrecord Person [name age occupation])

(clojure.spec/def ::person (clojure.spec/keys :req-un [::name ::age ::occupation]))

(clojure.spec/valid? ::person (->Person "john doe" 25 "programmer"))
;;=> true

(clojure.spec/valid? ::person (->Person "john doe" "25" "programmer"))
;;=> false
```

At some point in the future, a reader syntax or built-in support for qualifying record keys by the records' namespace may be introduced. This support already exists for maps.

Map specs

You can spec a map by specifying which keys should be present in the map:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req [::name ::age ::occupation]))

(clojure.spec/valid? ::person {::name "john" ::age 25 ::occupation "programmer"})
;; => true
```

`:req` is a vector of keys required to be present in the map. You can specify additional options such as `:opt`, a vector of keys which are optional.

The examples so far require that the keys in the name are namespace-qualified. But it's common for map keys to be unqualified. For this case, `clojure.spec` provides `:req` and `:opt` equivalents for unqualified keys: `:req-un` and `:opt-un`. Here's the same example, with unqualified keys:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req-un [::name ::age ::occupation]))

(clojure.spec/valid? ::person {:name "john" :age 25 :occupation "programmer"})
;; => true
```

Notice how the specs provided in the `:req-un` vector are still qualified. `clojure.spec`, will automatically confirm the unqualified versions in the map when conforming the values.

namespace map literal syntax allows you to qualify all the keys of a map by a single namespace succinctly. For example:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req [::name ::age ::occupation]))

(clojure.spec/valid? ::person #user{:name "john" :age 25 :occupation "programmer"})
;; => true
```

Notice the special `#:` reader syntax. We follow this with the namespace we wish to qualify all the map keys by. These will then be checked against the specs corresponding to the provided namespace.

Collections

You can spec collections in a number of ways. `coll-of` allows you to spec collections and provide some additional constraints. Here's a simple example:

```
(clojure.spec/valid? (clojure.spec/coll-of int?) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int?) '(1 2 3))
;; => true
```

Constraint options follow the main spec/predicate for the collection. You can constrain the collection type with `:kind` like this:

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector?) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector?) '(1 2 3))
;; => false
```

The above is false because the collection passed in is not a vector.

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind list?) '(1 2 3))
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind set?) #{1 2 3})
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind set?) #{1 "2" 3})
;; => false
```

The above is false because not all elements in the set are ints.

You can also constrain the size of the collection in a few ways:

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector? :count 3) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector? :count 3) [1 2])
;; => false

(clojure.spec/valid? (clojure.spec/coll-of int? :min-count 3 :max-count 5) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :min-count 3 :max-count 5) [1 2])
;; => false
```

You can also enforce uniqueness of the elements in the collection with `:distinct`:

```
(clojure.spec/valid? (clojure.spec/coll-of int? :distinct true) [1 2])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :distinct true) [2 2])
;; => false
```

`coll-of` ensures all elements in a sequence are checked. For large collections, this can be very inefficient. `every` behaves just like `coll-of`, except it only samples a relatively small number of the sequences' elements from for conformance. This works well for large collections. Here's an example:

```
(clojure.spec/valid? (clojure.spec/every int? :distinct true) [1 2 3 4 5])
;; => true
```

`map-of` is similar to `coll-of`, but for maps. Since maps have both keys and values, you must supply both a spec for the key and a spec for the value:

```
(clojure.spec/valid? (clojure.spec/map-of keyword? string?) {:red "red" :green "green"})
;; => true
```

Like `coll-of`, `map-of` checks conformance of all map key/values. For large maps this will be inefficient. Like `coll-of`, `map-of` supplies `every-kv` for efficiently sampling a relatively small number of values from a large map:

```
(clojure.spec/valid? (clojure.spec/every-kv keyword? string?) {:red "red" :green "green"})  
;; => true
```

Sequences

spec can describe and be used with arbitrary sequences. It supports this via a number of regex spec operations.

```
(clojure.spec/valid? (clojure.spec/cat :text string? :int int?) ["test" 1])  
;;=> true
```

`cat` requires labels for each spec used to describe the sequence. `cat` describes a sequence of elements and a spec for each one.

`alt` is used to choose among a number of possible specs for a given element in a sequence. For example:

```
(clojure.spec/valid? (clojure.spec/cat :text-or-int (clojure.spec/alt :text string? :int int?)) ["test"])  
;;=> true
```

`alt` also requires that each spec is labeled by a keyword.

Regex sequences can be composed in some very interesting and powerful ways to create arbitrarily complex sequence-describing specs. Here's a slightly more complex example:

```
(clojure.spec/def ::complex-seq (clojure.spec/+ (clojure.spec/cat :num int? :foo-map  
(clojure.spec/map-of keyword? int?))))  
(clojure.spec/valid? ::complex-seq [0 {:foo 3 :baz 1} 4 {:foo 4}])  
;;=> true
```

Here `::complex-seq` will validate a sequence of one or more pairs of elements, the first being an int and the second being a map of keyword to int.

Read `clojure.spec` online: <https://riptutorial.com/clojure/topic/2325/clojure-spec>

Chapter 7: clojure.test

Examples

is

The `is` macro is the core of the `clojure.test` library. It returns the value of its body expression, printing an error message if the expression returns a falsey value.

```
(defn square [x]
  (+ x x))

(require '[clojure.test :as t])

(t/is (= 0 (square 0)))
;;=> true

(t/is (= 1 (square 1)))
;;
;; FAIL in () (foo.clj:1)
;; expected: (= 1 (square 1))
;; actual: (not (= 1 2))
;;=> false
```

Grouping related tests with the testing macro

You can group related assertions in `deftest` unit tests within a context using the `testing` macro:

```
(deftest add-nums
  (testing "Positive cases"
    (is (= 2 (+ 1 1)))
    (is (= 4 (+ 2 2))))
  (testing "Negative cases"
    (is (= -1 (+ 2 -3)))
    (is (= -4 (+ 8 -12)))))
```

This will help clarify test output when run. Note that `testing` must occur inside a `deftest`.

Defining a test with deftest

`deftest` is a macro for defining a unit test, similar to unit tests in other languages.

You can create a test as follows:

```
(deftest add-nums
  (is (= 2 (+ 1 1)))
  (is (= 3 (+ 1 2))))
```

Here we are defining a test called `add-nums`, which tests the `+` function. The test has two assertions.

You can then run the test like this in your current namespace:

```
(run-tests)
```

Or you can just run the tests for the namespace the test is in:

```
(run-tests 'your-ns)
```

are

The `are` macro is also part of the `clojure.test` library. It allows you to make multiple assertions against a template.

For example:

```
(are [x y] (= x y)
     4 (+ 2 2)
     8 (* 2 4))
=> true
```

Here, `(= x y)` acts as a template which takes each argument and creates an `is` assertion out of it.

This expands to multiple `is` assertions:

```
(do
  (is (= 4 (+ 2 2)))
  (is (= 8 (* 2 4))))
```

Wrap each test or all tests with use-fixtures

`use-fixtures` allows to wrap each `deftest` in namespace with code that runs before and after test. It can be used for fixtures or stubbing.

Fixtures are just functions that take test function and run it with other necessary steps (before/after, wrap).

```
(ns myapp.test
  (require [clojure.test :refer :all])

(defn stub-current-thing [body]
  ;; with-redefs stubs things/current-thing function to return fixed
  ;; value for duration of each test
  (with-redefs [things/current-thing (fn [] {:foo :bar})]
    ;; run test body
    (body)))

(use-fixtures :each stub-current-thing)
```

When used with `:once`, it wraps whole run of tests in current namespace with function

```
(defn database-for-tests [all-tests]
  (setup-database)
  (all-tests)
  (drop-database))

(use-fixtures :once database-for-tests)
```

Running tests with Leiningen

If you are using Leiningen and your tests are located in the test directory in your project root then you can run your tests using `lein test`

Read `clojure.test` online: <https://riptutorial.com/clojure/topic/1901/clojure-test>

Chapter 8: Collections and Sequences

Syntax

- `()` → `()`
- `(1 2 3 4 5)` → `(1 2 3 4 5)`
- `(1 foo 2 bar 3)` → `(1 'foo 2 'bar 3)`
- `(list 1 2 3 4 5)` → `(1 2 3 4 5)`
- `(list* [1 2 3 4 5])` → `(1 2 3 4 5)`
- `[]` → `[]`
- `[1 2 3 4 5]` → `[1 2 3 4 5]`
- `(vector 1 2 3 4 5)` → `[1 2 3 4 5]`
- `(vec '(1 2 3 4 5))` → `[1 2 3 4 5]`
- `{}` ⇒ `{}`
- `{:keyA 1 :keyB 2}` → `{:keyA 1 :keyB 2}`
- `{:keyA 1, :keyB 2}` → `{:keyA 1 :keyB 2}`
- `(hash-map :keyA 1 :keyB 2)` → `{:keyA 1 :keyB 2}`
- `(sorted-map 5 "five" 1 "one")` → `{1 "one" 5 "five"}` (entries are sorted by key when used as a sequence)
- `#{} → #{}`
- `#{1 2 3 4 5} → #{4 3 2 5 1} (unordered)`
- `(hash-set 1 2 3 4 5) → #{2 5 4 1 3} (unordered)`
- `(sorted-set 2 5 4 3 1) → #{1 2 3 4 5}`

Examples

Collections

All built-in Clojure collections are immutable and heterogeneous, have literal syntax, and support the `conj`, `count`, and `seq` functions.

- `conj` returns a new collection that is equivalent to an existing collection with an item "added", in either "constant" or logarithmic time. What exactly this means depends on the collection.
- `count` returns the number of items in a collection, in constant time.
- `seq` returns `nil` for an empty collection, or a sequence of items for a non-empty collection, in constant time.

Lists

A list is denoted by parentheses:

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

A Clojure list is a [singly linked list](#). `conj` "conjoins" a new element to the collection in the most efficient location. For lists, this is at the beginning:

```
(conj () :foo)
;;=> (:foo)

(conj (conj () :bar) :foo)
;;=> (:foo :bar)
```

Unlike other collections, non-empty lists are evaluated as calls to special forms, macros, or functions when evaluated. Therefore, while `(:foo)` is the literal representation of the list containing `:foo` as its only item, evaluating `(:foo)` in a REPL will cause an `IllegalArgumentException` to be thrown because a keyword cannot be invoked as a [nullary function](#).

```
(:foo)
;; java.lang.IllegalArgumentException: Wrong number of args passed to keyword: :foo
```

To prevent Clojure from evaluating a non-empty list, you can [quote](#) it:

```
'(:foo)
;;=> (:foo)

'(:foo :bar)
;;=> (:foo :bar)
```

Unfortunately, this causes the elements to not be evaluated:

```
(+ 1 1)
;;=> 2

'(1 (+ 1 1) 3)
;;=> (1 (+ 1 1) 3)
```

For this reason, you'll usually want to use `list`, a [variadic function](#) that evaluates all of its arguments and uses those results to construct a list:

```
(list)
;;=> ()

(list :foo)
;;=> (:foo)

(list :foo :bar)
;;=> (:foo :bar)

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

`count` returns the number of items, in constant time:

```
(count ())
;;=> 0
```

```
(count (conj () :foo))  
;;=> 1  
  
(count '(:foo :bar))  
;;=> 2
```

You can test whether something is a list using the `list?` predicate:

```
(list? ())  
;;=> true  
  
(list? '(:foo :bar))  
;;=> true  
  
(list? nil)  
;;=> false  
  
(list? 42)  
;;=> false  
  
(list? :foo)  
;;=> false
```

You can get the first element of a list using `peek`:

```
(peek ())  
;;=> nil  
  
(peek '(:foo))  
;;=> :foo  
  
(peek '(:foo :bar))  
;;=> :foo
```

You can get a new list without the first element using `pop`:

```
(pop '(:foo))  
;;=> ()  
  
(pop '(:foo :bar))  
;;=> (:bar)
```

Note that if you try to `pop` an empty list, you'll get an `IllegalStateException`:

```
(pop ())  
;; java.lang.IllegalStateException: Can't pop empty list
```

Finally, all lists are sequences, so you can do everything with a list that you can do with any other sequence. Indeed, with the exception of the empty list, calling `seq` on a list returns the exact same object:

```
(seq ())  
;;=> nil
```

```
(seq '(:foo))
;;=> (:foo)

(seq '(:foo :bar))
;;=> (:foo :bar)

(let [x '(:foo :bar)]
  (identical? x (seq x)))
;;=> true
```

Sequences

A sequence is very much like a list: it is an immutable object that can give you its [first](#) element or the [rest](#) of its elements in constant time. You can also [construct](#) a new sequence from an existing sequence and an item to stick at the beginning.

You can test whether something is a sequence using the [seq?](#) predicate:

```
(seq? nil)
;;=> false

(seq? 42)
;;=> false

(seq? :foo)
;;=> false
```

As you already know, lists are sequences:

```
(seq? ())
;;=> true

(seq? '(:foo :bar))
;;=> true
```

Anything you get by calling [seq](#) or [rseq](#) or [keys](#) or [vals](#) on a non-empty collection is also a sequence:

```
(seq? (seq ()))
;;=> false

(seq? (seq '(:foo :bar)))
;;=> true

(seq? (seq []))
;;=> false

(seq? (seq [[:foo :bar]]))
;;=> true

(seq? (rseq []))
;;=> false

(seq? (rseq [[:foo :bar]]))
```

```

;;=> true

(seq? (seq {}))
;;=> false

(seq? (seq {:foo :bar :baz :qux}))
;;=> true

(seq? (keys {}))
;;=> false

(seq? (keys {:foo :bar :baz :qux}))
;;=> true

(seq? (vals {}))
;;=> false

(seq? (vals {:foo :bar :baz :qux}))
;;=> true

(seq? (seq #{}))
;;=> false

(seq? (seq #{:foo :bar}))
;;=> true

```

Remember that all lists are sequences, but not all sequences are lists. While lists support `peek` and `pop` and `count` in constant time, in general, a sequence does not need to support any of those functions. If you try to call `peek` or `pop` on a sequence that doesn't also support Clojure's stack interface, you'll get a `ClassCastException`:

```

(peek (seq [:foo :bar]))
;; java.lang.ClassCastException: clojure.lang.PersistentVector$ChunkedSeq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq [:foo :bar]))
;; java.lang.ClassCastException: clojure.lang.PersistentVector$ChunkedSeq cannot be cast to
clojure.lang.IPersistentStack

(peek (seq #{:foo :bar}))
;; java.lang.ClassCastException: clojure.lang.APersistentMap$KeySeq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq #{:foo :bar}))
;; java.lang.ClassCastException: clojure.lang.APersistentMap$KeySeq cannot be cast to
clojure.lang.IPersistentStack

(peek (seq {:foo :bar :baz :qux}))
;; java.lang.ClassCastException: clojure.lang.PersistentArrayMap$Seq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq {:foo :bar :baz :qux}))
;; java.lang.ClassCastException: clojure.lang.PersistentArrayMap$Seq cannot be cast to
clojure.lang.IPersistentStack

```

If you call `count` on a sequence that doesn't implement `count` in constant time, you won't get an error; instead, Clojure will traverse the entire sequence until it reaches the end, then return the number of elements that it traversed. This means that, for general sequences, `count` is linear, not

constant, time. You can test whether something supports constant-time `count` using the `counted?` predicate:

```
(counted? '(:foo :bar))
;;=> true

(counted? (seq '(:foo :bar)))
;;=> true

(counted? [ :foo :bar ])
;;=> true

(counted? (seq [ :foo :bar ]))
;;=> true

(counted? { :foo :bar :baz :qux })
;;=> true

(counted? (seq { :foo :bar :baz :qux }))
;;=> true

(counted? #{ :foo :bar })
;;=> true

(counted? (seq #{ :foo :bar }))
;;=> false
```

As mentioned above, you can use `first` to get the first element of a sequence. Note that `first` will call `seq` on their argument, so it can be used on anything "seqable", not just actual sequences:

```
(first nil)
;;=> nil

(first '(:foo))
;;=> :foo

(first '(:foo :bar))
;;=> :foo

(first [ :foo ])
;;=> :foo

(first [ :foo :bar ])
;;=> :foo

(first { :foo :bar })
;;=> [ :foo :bar ]

(first #{ :foo })
;;=> :foo
```

Also as mentioned above, you can use `rest` to get a sequence containing all but the first element of an existing sequence. Like `first`, it calls `seq` on its argument. However, it does *not* call `seq` on its result! This means that, if you call `rest` on a sequence that contains fewer than two items, you'll get back `()` instead of `nil`:


```

(rest nil)
;;=> ()

(rest '(:foo))
;;=> ()

(rest '(:foo :bar))
;;=> (:bar)

(rest [ :foo ])
;;=> ()

(rest [ :foo :bar ])
;;=> (:bar)

(rest { :foo :bar })
;;=> ()

(rest #{ :foo })
;;=> ()

```

If you want to get back `nil` when there aren't any more elements in a sequence, you can use `next` instead of `rest`:

```

(next nil)
;;=> nil

(next '(:foo))
;;=> nil

(next [ :foo ])
;;=> nil

```

You can use the `cons` function to create a new sequence that will return its first argument for `first` and its second argument for `rest`:

```

(cons :foo nil)
;;=> (:foo)

(cons :foo (cons :bar nil))
;;=> (:foo :bar)

```

Clojure provides a large [sequence library](#) with many functions for dealing with sequences. The important thing about this library is that it works with anything "seqable", not just lists. That's why the concept of a sequence is so useful; it means that a single function, like `reduce`, works perfectly on any collection:

```

(reduce + '(1 2 3))
;;=> 6

(reduce + [1 2 3])
;;=> 6

(reduce + #{1 2 3})
;;=> 6

```

The other reason that sequences are useful is that, since they don't mandate any particular implementation of `first` and `rest`, they allow for lazy sequences whose elements are only realized when necessary.

Given an expression that would create a sequence, you can wrap that expression in the `lazy-seq` macro to get an object that acts like a sequence, but will only actually evaluate that expression when it is asked to do so by the `seq` function, at which point it will cache the result of the expression and forward `first` and `rest` calls to the cached result.

For finite sequences, a lazy sequence usually acts the same as an equivalent eager sequence:

```
(seq [:foo :bar])
;;=> (:foo :bar)

(lazy-seq [:foo :bar])
;;=> (:foo :bar)
```

However, the difference becomes apparent for infinite sequences:

```
(defn eager-fibonacci [a b]
  (cons a (eager-fibonacci b (+ a b))))

(defn lazy-fibonacci [a b]
  (lazy-seq (cons a (lazy-fibonacci b (+ a b)))))

(take 10 (eager-fibonacci 0 1))
;; java.lang.StackOverflowError:

(take 10 (lazy-fibonacci 0 1))
;;=> (0 1 1 2 3 5 8 13 21 34)
```

Vectors

A vector is denoted by square brackets:

```
[]
;;=> []

[:foo]
;;=> [:foo]

[:foo :bar]
;;=> [:foo :bar]

[1 (+ 1 1) 3]
;;=> [1 2 3]
```

In addition using to the literal syntax, you can also use the `vector` function to construct a vector:

```
(vector)
;;=> []

(vector :foo)
```

```
;;=> [:foo]

(vector :foo :bar)
;;=> [:foo :bar]

(vector 1 (+ 1 1) 3)
;;=> [1 2 3]
```

You can test whether something is a vector using the `vector?` predicate:

```
(vector? [])
;;=> true

(vector? [:foo :bar])
;;=> true

(vector? nil)
;;=> false

(vector? 42)
;;=> false

(vector? :foo)
;;=> false
```

`conj` adds elements to the end of a vector:

```
(conj [] :foo)
;;=> [:foo]

(conj (conj [] :foo) :bar)
;;=> [:foo :bar]

(conj [] :foo :bar)
;;=> [:foo :bar]
```

`count` returns the number of items, in constant time:

```
(count [])
;;=> 0

(count (conj [] :foo))
;;=> 1

(count [:foo :bar])
;;=> 2
```

You can get the last element of a vector using `peek`:

```
(peek [])
;;=> nil

(peek [:foo])
;;=> :foo

(peek [:foo :bar])
```

```
;;=> :bar
```

You can get a new vector without the last element using `pop`:

```
(pop [:foo])  
;;=> []  
  
(pop [:foo :bar])  
;;=> [:foo]
```

Note that if you try to pop an empty vector, you'll get an `IllegalStateException`:

```
(pop [])  
;; java.lang.IllegalStateException: Can't pop empty vector
```

Unlike lists, vectors are indexed. You can get an element of a vector by index in "constant" time using `get`:

```
(get [:foo :bar] 0)  
;;=> :foo  
  
(get [:foo :bar] 1)  
;;=> :bar  
  
(get [:foo :bar] -1)  
;;=> nil  
  
(get [:foo :bar] 2)  
;;=> nil
```

In addition, vectors themselves are functions that take an index and return the element at that index:

```
([:foo :bar] 0)  
;;=> :foo  
  
([:foo :bar] 1)  
;;=> :bar
```

However, if you call a vector with an invalid index, you'll get an `IndexOutOfBoundsException` instead of `nil`:

```
([:foo :bar] -1)  
;; java.lang.IndexOutOfBoundsException:  
  
([:foo :bar] 2)  
;; java.lang.IndexOutOfBoundsException:
```

You can get a new vector with a different value at a particular index using `assoc`:

```
(assoc [:foo :bar] 0 42)  
;;=> [42 :bar]
```

```
(assoc [:foo :bar] 1 42)
;;=> [:foo 42]
```

If you pass an index equal to the `count` of the vector, Clojure will add the element as if you had used `conj`. However, if you pass an index that is negative or greater than the `count`, you'll get an [IndexOutOfBoundsException](#):

```
(assoc [:foo :bar] 2 42)
;;=> [:foo :bar 42]

(assoc [:foo :bar] -1 42)
;; java.lang.IndexOutOfBoundsException:

(assoc [:foo :bar] 3 42)
;; java.lang.IndexOutOfBoundsException:
```

You can get a sequence of the items in a vector using `seq`:

```
(seq [])
;;=> nil

(seq [:foo])
;;=> (:foo)

(seq [:foo :bar])
;;=> (:foo :bar)
```

Since vectors are indexed, you can also get a reversed sequence of a vector's items using `rseq`:

```
(rseq [])
;;=> nil

(rseq [:foo])
;;=> (:foo)

(rseq [:foo :bar])
;;=> (:bar :foo)
```

Note that, although all lists are sequences, and sequences are displayed in the same way as lists, not all sequences are lists!

```
('(:foo :bar))
;;=> (:foo :bar)

(seq [:foo :bar])
;;=> (:foo :bar)

(list? '(:foo :bar))
;;=> true

(list? (seq [:foo :bar]))
;;=> false

(list? (rseq [:foo :bar]))
```

```
;;=> false
```

Sets

Like maps, sets are associative and unordered. Unlike maps, which contain mappings from keys to values, sets essentially map from keys to themselves.

A set is denoted by curly braces preceded by an octothorpe:

```
#{}  
;;=> #{}  
  
#{:foo}  
;;=> #{:foo}  
  
#{:foo :bar}  
;;=> #{:bar :foo}
```

As with maps, the order in which elements appear in a literal set doesn't matter:

```
(= #{:foo :bar} #{:bar :foo})  
;;=> true
```

You can test whether something is a set using the `set?` predicate:

```
(set? #{})  
;;=> true  
  
(set? #{:foo})  
;;=> true  
  
(set? #{:foo :bar})  
;;=> true  
  
(set? nil)  
;;=> false  
  
(set? 42)  
;;=> false  
  
(set? :foo)  
;;=> false
```

You can test whether a map contains a given item in "constant" time using the `contains?` predicate:

```
(contains? #{} :foo)  
;;=> false  
  
(contains? #{:foo} :foo)  
;;=> true  
  
(contains? #{:foo} :bar)  
;;=> false
```

```
(contains? #{} nil)
;;=> false

(contains? #{nil} nil)
;;=> true
```

In addition, sets themselves are functions that take an element and return that element if it is present in the set, or `nil` if it isn't:

```
(#{ } :foo)
;;=> nil

(#{:foo} :foo)
;;=> :foo

(#{:foo} :bar)
;;=> nil

(#{ } nil)
;;=> nil

(#{nil} nil)
;;=> nil
```

You can use `conj` to get a set that has all the elements of an existing set, plus one additional item:

```
(conj #{} :foo)
;;=> #{:foo}

(conj (conj #{} :foo) :bar)
;;=> #{:bar :foo}

(conj #{:foo} :foo)
;;=> #{:foo}
```

You can use `disj` to get a set that has all the elements of an existing set, minus one item:

```
(disj #{} :foo)
;;=> #{}

(disj #{:foo} :foo)
;;=> #{}

(disj #{:foo} :bar)
;;=> #{:foo}

(disj #{:foo :bar} :foo)
;;=> #{:bar}

(disj #{:foo :bar} :bar)
;;=> #{:foo}
```

`count` returns the number of elements, in constant time:

```
(count #{})
;;=> 0
```

```
(count (conj #{} :foo))
;;=> 1

(count #{:foo :bar})
;;=> 2
```

You can get a sequence of all elements in a set using `seq`:

```
(seq #{})
;;=> nil

(seq #{:foo})
;;=> (:foo)

(seq #{:foo :bar})
;;=> (:bar :foo)
```

Maps

Unlike the list, which is a sequential data structure, and the vector, which is both sequential and associative, the map is exclusively an associative data structure. A map consists of a set of mappings from keys to values. All keys are unique, so maps support "constant"-time lookup from keys to values.

A map is denoted by curly braces:

```
{}
```

```
;;=> {}

{:foo :bar}
```

```
;;=> {:foo :bar}
```

```
{:foo :bar :baz :qux}
```

```
;;=> {:foo :bar, :baz :qux}
```

Each pair of two elements is a key-value pair. So, for instance, the first map above has no mappings. The second has one mapping, from the key `:foo` to the value `:bar`. The third has two mappings, one from the key `:foo` to the value `:bar`, and one from the key `:baz` to the value `:qux`. Maps are inherently unordered, so the order in which the mappings appear doesn't matter:

```
(= {:foo :bar :baz :qux}
   {:baz :qux :foo :bar})
;;=> true
```

You can test whether something is a map using the `map?` predicate:

```
(map? {})
;;=> true

(map? {:foo :bar})
;;=> true
```



```
(map? {:foo :bar :baz :qux})  
;;=> true  
  
(map? nil)  
;;=> false  
  
(map? 42)  
;;=> false  
  
(map? :foo)  
;;=> false
```

You can test whether a map contains a given *key* in "constant" time using the `contains?` predicate:

```
(contains? {:foo :bar :baz :qux} 42)  
;;=> false  
  
(contains? {:foo :bar :baz :qux} :foo)  
;;=> true  
  
(contains? {:foo :bar :baz :qux} :bar)  
;;=> false  
  
(contains? {:foo :bar :baz :qux} :baz)  
;;=> true  
  
(contains? {:foo :bar :baz :qux} :qux)  
;;=> false  
  
(contains? {:foo nil} :foo)  
;;=> true  
  
(contains? {:foo nil} :bar)  
;;=> false
```

You can get the value associated with a key using `get`:

```
(get {:foo :bar :baz :qux} 42)  
;;=> nil  
  
(get {:foo :bar :baz :qux} :foo)  
;;=> :bar  
  
(get {:foo :bar :baz :qux} :bar)  
;;=> nil  
  
(get {:foo :bar :baz :qux} :baz)  
;;=> :qux  
  
(get {:foo :bar :baz :qux} :qux)  
;;=> nil  
  
(get {:foo nil} :foo)  
;;=> nil  
  
(get {:foo nil} :bar)  
;;=> nil
```

In addition, maps themselves are functions that take a key and return the value associated with that key:

```
({:foo :bar :baz :qux} 42)
;;=> nil

({:foo :bar :baz :qux} :foo)
;;=> :bar

({:foo :bar :baz :qux} :bar)
;;=> nil

({:foo :bar :baz :qux} :baz)
;;=> :qux

({:foo :bar :baz :qux} :qux)
;;=> nil

({:foo nil} :foo)
;;=> nil

({:foo nil} :bar)
;;=> nil
```

You can get an entire map entry (key and value together) as a two-element vector using `find`:

```
(find {:foo :bar :baz :qux} 42)
;;=> nil

(find {:foo :bar :baz :qux} :foo)
;;=> [:foo :bar]

(find {:foo :bar :baz :qux} :bar)
;;=> nil

(find {:foo :bar :baz :qux} :baz)
;;=> [:baz :qux]

(find {:foo :bar :baz :qux} :qux)
;;=> nil

(find {:foo nil} :foo)
;;=> [:foo nil]

(find {:foo nil} :bar)
;;=> nil
```

You can extract the key or value from a map entry using `key` or `val`, respectively:

```
(key (find {:foo :bar} :foo))
;;=> :foo

(val (find {:foo :bar} :foo))
;;=> :bar
```

Note that, although all Clojure map entries are vectors, not all vectors are map entries. If you try to call `key` or `val` on anything that's not a map entry, you'll get a `ClassCastException`:

```
(key [:foo :bar])  
;; java.lang.ClassCastException:  
  
(val [:foo :bar])  
;; java.lang.ClassCastException:
```

You can test whether something is a map entry using the `map-entry?` predicate:

```
(map-entry? (find {:foo :bar} :foo))  
;;=> true  
  
(map-entry? [:foo :bar])  
;;=> false
```

You can use `assoc` to get a map that has all the same key-value pairs as an existing map, with one mapping added or changed:

```
(assoc {} :foo :bar)  
;;=> {:foo :bar}  
  
(assoc (assoc {} :foo :bar) :baz :qux)  
;;=> {:foo :bar, :baz :qux}  
  
(assoc {:baz :qux} :foo :bar)  
;;=> {:baz :qux, :foo :bar}  
  
(assoc {:foo :bar :baz :qux} :foo 42)  
;;=> {:foo 42, :baz :qux}  
  
(assoc {:foo :bar :baz :qux} :baz 42)  
;;=> {:foo :bar, :baz 42}
```

You can use `dissoc` to get a map that has all the same key-value pairs as an existing map, with possibly one mapping removed:

```
(dissoc {:foo :bar :baz :qux} 42)  
;;=> {:foo :bar :baz :qux}  
  
(dissoc {:foo :bar :baz :qux} :foo)  
;;=> {:baz :qux}  
  
(dissoc {:foo :bar :baz :qux} :bar)  
;;=> {:foo :bar :baz :qux}  
  
(dissoc {:foo :bar :baz :qux} :baz)  
;;=> {:foo :bar}  
  
(dissoc {:foo :bar :baz :qux} :qux)  
;;=> {:foo :bar :baz :qux}  
  
(dissoc {:foo nil} :foo)  
;;=> {}
```

`count` returns the number of mappings, in constant time:

```
(count {})
;;=> 0

(count (assoc {} :foo :bar))
;;=> 1

(count {:foo :bar :baz :qux})
;;=> 2
```

You can get a sequence of all entries in a map using `seq`:

```
(seq {})
;;=> nil

(seq {:foo :bar})
;;=> ([:foo :bar])

(seq {:foo :bar :baz :qux})
;;=> ([:foo :bar] [:baz :qux])
```

Again, maps are unordered, so the ordering of the items in a sequence that you get by calling `seq` on a map is undefined.

You can get a sequence of just the keys or just the values in a map using `keys` or `vals`, respectively:

```
(keys {})
;;=> nil

(keys {:foo :bar})
;;=> (:foo)

(keys {:foo :bar :baz :qux})
;;=> (:foo :baz)

(vals {})
;;=> nil

(vals {:foo :bar})
;;=> (:bar)

(vals {:foo :bar :baz :qux})
;;=> (:bar :qux)
```

Clojure 1.9 adds a literal syntax for more concisely representing a map where the keys share the same namespace. Note that the map in either case is identical (the map does not "know" the default namespace), this is merely a syntactic convenience.

```
;; typical map syntax
(def p {:person/first "Darth" :person/last "Vader" :person/email "darth@death.star"})

;; namespace map literal syntax
(def p #:person{:first "Darth" :last "Vader" :email "darth@death.star"})
```

Read Collections and Sequences online: <https://riptutorial.com/clojure/topic/1389/collections-and-sequences>

Chapter 9: core.async

Examples

basic channel operations: creating, putting, taking, closing, and buffers.

`core.async` is about making *processes* that take values from and put values into *channels*.

```
(require [clojure.core.async :as a])
```

Creating channels with `chan`

You create a channel using the `chan` function:

```
(def chan-0 (a/chan)) ;; unbuffered channel: acts as a rendez-vous point.
(def chan-1 (a/chan 3)) ;; channel with a buffer of size 3.
(def chan-2 (a/chan (a/dropping-buffer 3)) ;; channel with a *dropping* buffer of size 3
(def chan-3 (a/chan (a/sliding-buffer 3)) ;; channel with a *sliding* buffer of size 3
```

Putting values into channels with `>!!` and `>!`

You put values into a channel with `>!!`:

```
(a/>!! my-channel :an-item)
```

You can put any value (Strings, numbers, maps, collections, objects, even other channels, etc.) into a channel, except `nil`:

```
;; WON'T WORK
(a/>!! my-channel nil)
=> IllegalArgumentException Can't put nil on channel
```

Depending on the channel's buffer, `>!!` may block the current thread.

```
(let [ch (a/chan)] ;; unbuffered channel
  (a/>!! ch :item)
  ;; the above call blocks, until another process
  ;; takes the item from the channel.
)
(let [ch (a/chan 3)] ;; channel with 3-size buffer
  (a/>!! ch :item-1) ;; => true
  (a/>!! ch :item-2) ;; => true
  (a/>!! ch :item-3) ;; => true
  (a/>!! ch :item-4)
  ;; now the buffer is full; blocks until :item-1 is taken from ch.
)
```

From inside a `(go ...)` block, you can - and should - use `a/>!` instead of `a/>!!`:

```
(a/go (a/>! ch :item))
```

The logical behaviour will be the same as `a/>!!`, but only the logical process of the goroutine will block instead of the actual OS thread.

Using `a/>!!` inside of a `(go ...)` block is an anti-pattern:

```
;; NEVER DO THIS
(a/go
  (a/>!! ch :item))
```

Taking values from channels with `<!!`

You take a value from a channel using `<!!`:

```
;; creating a channel
(def ch (a/chan 3))
;; putting some items in it
(do
  (a/>!! ch :item-1)
  (a/>!! ch :item-2)
  (a/>!! ch :item-3))
;; taking a value
(a/<!! ch) ;; => :item-1
(a/<!! ch) ;; => :item-2
```

If no item is available in the channel, `a/<!!` will block the current Thread until a value is put in the channel (or the channel is closed, see later):

```
(def ch (a/chan))
(a/<!! ch) ;; blocks until another process puts something into ch or closes it
```

From inside a `(go ...)` block, you can - and should - use `a/<!` instead of `a/<!!`:

```
(a/go (let [x (a/<! ch)] ...))
```

The logical behaviour will be the same as `a/<!!`, but only the logical process of the goroutine will block instead of the actual OS thread.

Using `a/<!!` inside of a `(go ...)` block is an anti-pattern:

```
;; NEVER DO THIS
(a/go
  (a/<!! ch))
```

Closing channels

You *close* a channel with `a/close!`:

```
(a/close! ch)
```

Once a channel is closed, and the all data in the channel has been exhausted, `takes` will always return `nil`:

```
(def ch (a/chan 5))

;; putting 2 values in the channel, then closing it
(a/>!! ch :item-1)
(a/>!! ch :item-2)
(a/close! ch)

;; taking from ch will return the items that were put in it, then nil
(a/<!! ch) ;; => :item-1
(a/<!! ch) ;; => :item-2
(a/<!! ch) ;; => nil
(a/<!! ch) ;; => nil
(a/<!! ch) ;; => nil

;; once the channel is closed, >!! will have no effect on the channel:
(a/>!! ch :item-3)
=> false ;; false means the put did not succeed
(a/<!! ch) ;; => nil
```

Asynchronous puts with `put!`

As an alternative to `a/>!!` (which may block), you can call `a/put!` to put a value in a channel in another thread, with an optional callback.

```
(a/put! ch :item)
(a/put! ch :item (fn once-put [closed?] ...)) ;; callback function, will receive
```

In ClojureScript, since blocking the current Thread is not possible, `a/>!!` is not supported, and `put!` is the only way to put data into a channel from outside of a `(go)` block.

Asynchronous takes with `take!`

As an alternative to `a/<!!` (which may block the current thread), you may use `a/take!` to take a value from a channel asynchronously, passing it to a callback.

```
(a/take! ch (fn [x] (do something with x)))
```

Using dropping and sliding buffers

With dropping and sliding buffers, puts never block, however, when the buffer is full, you lose data. Dropping buffer lose the last data added, whereas sliding buffers lose the first data added.

Dropping buffer example:

```
(def ch (a/chan (a/dropping-buffer 2)))

;; putting more items than buffer size
(a/>!! ch :item-1)
=> true ;; put succeeded
(a/>!! ch :item-2)
=> true
(a/>!! ch :item-3)
=> false ;; put failed

;; now we take from the channel
(a/<!! ch)
=> :item-1
(a/<!! ch)
=> :item-2
(a/<!! ch)
=> blocks! :item-3 is lost
```

Sliding buffer example:

```
(def ch (a/chan (a/sliding-buffer 2)))

;; putting more items than buffer size
(a/>!! ch :item-1)
=> true
(a/>!! ch :item-2)
=> true
(a/>!! ch :item-3)
=> true

;; now when we take from the channel:
(a/<!! ch)
=> :item-2
(a/<!! ch)
=> :item-3
;; :item-1 was lost
```

Read core.async online: <https://riptutorial.com/clojure/topic/5496/core-async>

Chapter 10: Emacs CIDER

Introduction

CIDER is the acronym for **C**lojure(script) **I**nteractive **D**evelopment **E**nvironment that **R**ocks. It is an extension to emacs. CIDER aims to provide an interactive development environment to the programmer. CIDER is built on top of nREPL, a networked REPL server and SLIME served as the principle inspiration for CIDER.

Examples

Function Evaluation

CIDER function `cider-eval-last-sexp` can be used to execute the the code while editing the code inside the buffer. This function is by default binded to `C-x C-e` or `C-x C-e`.

CIDER manual says `C-x C-e` or `C-c C-e` will:

Evaluate the form preceding point and display the result in the echo area and/or in an buffer overlay.

For example:

```
(defn say-hello
  [username]
  (format "Hello, my name is %s" username))

(defn introducing-bob
  []
  (say-hello "Bob")) => "Hello, my name is Bob"
```

Performing `C-x C-e` or `C-c C-e` while your cursor is just ahead of the ending paren of `say-hello` function call will output the string `Hello, my name is Bob`.

Pretty Print

CIDER function `cider-insert-last-sexp-in-repl` can be used to execute the the code while editing the code inside the buffer and get the output pretty printed in a different buffer. This function is by default binded to `C-c C-p`.

CIDER manual says `C-c C-p` will

Evaluate the form preceding point and pretty-print the result in a popup buffer.

For example

```

(def databases {:database1 {:password "password"
                             :database "test"
                             :port "5432"
                             :host "localhost"
                             :user "username"}

               :database2 {:password "password"
                             :database "different_test_db"
                             :port "5432"
                             :host "localhost"
                             :user "vader"}}})

(defn get-database-config
  []
  databases)

(get-database-config)

```

Performing `C-c C-p` while your cursor is just ahead of the ending paren of `get-database-config` function call will output the pretty printed map in a new popup buffer.

```

{:database1
 {:password "password",
  :database "test",
  :port "5432",
  :host "localhost",
  :user "username"},
 :database2
 {:password "password",
  :database "different_test_db",
  :port "5432",
  :host "localhost",
  :user "vader"}}}

```

Read Emacs CIDER online: <https://riptutorial.com/clojure/topic/8847/emacs-cider>

Chapter 11: File Operations

Examples

Overview

Read a file all at once (not recommended for large files):

```
(slurp "./small_file.txt")
```

Write data to a file all at once:

```
(spit "./file.txt" "Ocelots are Awesome!") ; overwrite existing content
(spit "./log.txt" "2016-07-26 New entry." :append true)
```

Read a file line-by-line:

```
(use 'clojure.java.io)
(with-open [rdr (reader "./file.txt")]
  (line-seq rdr) ; returns lazy-seq
) ; with-open macro calls (.close rdr)
```

Write a file line-by-line:

```
(use 'clojure.java.io)
(with-open [wtr (writer "./log.txt" :append true)]
  (.write wtr "2016-07-26 New entry.")
) ; with-open macro calls (.close wtr)
```

Write to a file, replacing existing content:

```
(use 'clojure.java.io)
(with-open [wtr (writer "./file.txt")]
  (.write wtr "Everything in file.txt has been replaced with this text.")
) ; with-open macro calls (.close wtr)
```

Notes:

- You can specify URLs as well as files
- Options to `(slurp)` and `(spit)` are passed to `clojure.java.io/reader` and `/writer`, respectively.

Read File Operations online: <https://riptutorial.com/clojure/topic/3922/file-operations>

Chapter 12: Functions

Examples

Defining Functions

Functions are defined with five components:

The header, which includes the `defn` keyword, the name of the function.

```
(defn welcome ....)
```

An optional Docstring that explains and document what the function does.

```
(defn welcome
  "Return a welcome message to the world"
  ...)
```

Parameters listed in brackets.

```
(defn welcome
  "Return a welcome message"
  [name]
  ...)
```

The body, which describes the procedures the function carries out.

```
(defn welcome
  "Return a welcome message"
  [name]
  (str "Hello, " name "!"))
```

Calling it:

```
=> (welcome "World")

"Hello, World!"
```

Parameters and Arity

Clojure functions can be defined with zero or more parameters.

```
(defn welcome
  "Without parameters"
  []
  "Hello!")
```

```
(defn square
  "Take one parameter"
  [x]
  (* x x))

(defn multiplier
  "Two parameters"
  [x y]
  (* x y))
```

Arity

The number of arguments a function takes. Functions support *arity overloading*, which means that functions in Clojure allow for more than one "set" of arguments.

```
(defn sum-args
  ;; 3 arguments
  ([x y z]
   (+ x y z))
  ;; 2 arguments
  ([x y]
   (+ x y))
  ;; 1 argument
  ([x]
   (+ x 1)))
```

The arities don't have to do the same job, each arity can do something unrelated:

```
(defn do-something
  ;; 2 arguments
  ([first second]
   (str first " " second))
  ;; 1 argument
  ([x]
   (* x x x)))
```

Defining Variadic Functions

A Clojure function can be defined to take an arbitrary number of arguments, using the symbol **&** in its argument list. All remaining arguments are collected as a sequence.

```
(defn sum [& args]
  (apply + args))

(defn sum-and-multiply [x & args]
  (* x (apply + args)))
```

Calling:

```
=> (sum 1 11 23 42)
77
```

```
=> (sum-and-multiply 2 1 2 3)  ;; 2*(1+2+3)
12
```

Defining anonymous functions

There are two ways to define an anonymous function: the full syntax and a shorthand.

Full Anonymous Function Syntax

```
(fn [x y] (+ x y))
```

This expression evaluates to a function. Any syntax you can use with a function defined with `defn` (&, argument destructuring, etc.), you can also do with the `fn` form. `defn` is actually a macro that just does `(def (fn ...))`.

Shorthand Anonymous Function Syntax

```
#(+ %1 %2)
```

This is the shorthand notation. Using the shorthand notation, you don't have to name arguments explicitly; they'll be assigned the names `%1`, `%2`, `%3` and so on according to the order they're passed in. If the function only has one argument, its argument is called just `%`.

When To Use Each

The shorthand notation has some limitations. You can't destructure an argument, and you can't nest shorthand anonymous functions. The following code throws an error:

```
(def f #(map #(+ %1 2) %1))
```

Supported Syntax

You *can* use varargs with shorthand anonymous functions. This is completely legal:

```
 #(every? even? %&)
```

It takes a variable number of arguments and returns true if every one of them is even:

```
(#(every? even? %&) 2 4 6 8)
;; true
(#(every? even? %&) 1 2 4 6)
;; false
```

Despite the apparent contradiction, it is possible to write a named anonymous function by

including a name, as in the following example. This is especially useful if the function needs to call itself but also in stack traces.

```
(fn addition [& addends] (apply + addends))
```

Read Functions online: <https://riptutorial.com/clojure/topic/3078/functions>

Chapter 13: Getting started with web development

Examples

Create new Ring application with http-kit

[Ring](#) is de-facto standard API for clojure HTTP applications, similar to Ruby's Rack and Python's WSGI.

We're going to use it with [http-kit](#) webserver.

Create new Leiningen project:

```
lein new app myapp
```

Add http-kit dependency to `project.clj`:

```
:dependencies [[org.clojure/clojure "1.8.0"]
               [http-kit "2.1.18"]]
```

Add `:require` for http-kit to `core.clj`:

```
(ns test.core
  (:gen-class)
  (:require [org.httpkit.server :refer [run-server]]))
```

Define ring request handler. Request handlers are just functions from request to response and response is just a map:

```
(defn app [req]
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body "hello HTTP!"})
```

Here we just return 200 OK with the same content for any request.

Start the server in `-main` function:

```
(defn -main
  [& args]
  (run-server app {:port 8080}))
```

Run with `lein run` and open `http://localhost:8080/` in browser.

New web application with Luminus

Luminus is a Clojure micro-framework based on a set of lightweight libraries. It aims to provide a robust, scalable, and easy to use platform. With Luminus you can focus on developing your app the way you want without any distractions. It also has very good documentation that covers some of the major topics

It is very easy to start with luminus. Just create a new project with the following commands:

```
lein new luminus my-app
cd my-app
lein run
```

Your server will start on the port 3000

Running `lein new luminus myapp` will create an application using the default profile template. However, if you would like to attach further functionality to your template you can append profile hints for the extended functionality.

Web Servers

- `+aleph` - adds Aleph server support to the project
- `+jetty` - adds Jetty support to the project
- `+http-kit` - adds the HTTP Kit web server to the project

databases

- `+h2` - adds db.core namespace and H2 db dependencies
- `+sqlite` - adds db.core namespace and SQLite db dependencies
- `+postgres` - adds db.core namespace and add PostgreSQL dependencies
- `+mysql` - adds db.core namespace and add MySQL dependencies
- `+mongodb` - adds db.core namespace and MongoDB dependencies
- `+datomic` - adds db.core namespace and Datomic dependencies

miscellaneous

- `+auth` - adds Buddy dependency and authentication middleware
- `+auth-jwe` - adds Buddy dependency with the JWE backend
- `+cider` - adds support for CIDER using CIDER nREPL plugin
- `+cljs` - adds [ClojureScript][cljs] support with [Reagent](#)
- `+re-frame` - adds [ClojureScript][cljs] support with [re-frame](#)
- `+cucumber` - a profile for cucumber with clj-webdriver
- `+swagger` - adds support for Swagger-UI using the compojure-api library
- `+sassc` - adds support for SASS/SCSS files using SassC command line compiler
- `+service` - create a service application without the front-end boilerplate such as HTML templates
- `+war` - add support of building WAR archives for deployment to servers such as Apache Tomcat (should NOT be used for Immutant apps running on WildFly)

- `+site` - creates template for site using the specified database (H2 by default) and ClojureScript

To add a profile simply pass it as an argument after your application name, eg:

```
lein new luminus myapp +cljs
```

You can also mix multiple profiles when creating the application, eg:

```
lein new luminus myapp +cljs +swagger +postgres
```

Read [Getting started with web development online](https://riptutorial.com/clojure/topic/2323/getting-started-with-web-development):

<https://riptutorial.com/clojure/topic/2323/getting-started-with-web-development>

Chapter 14: Java interop

Syntax

- `.` let's you access instance methods
- `.-` let's you access instance fields
- `..` macro expanding to multiple nested invocations of `.`

Remarks

As a hosted language, Clojure provides excellent interoperability support with Java. Clojure code can also be called directly from Java.

Examples

Calling an instance method on a Java object

You can call an instance method using the `.` special form:

```
(.trim " hello ")  
;;=> "hello"
```

You can call instance methods with arguments like this:

```
(.substring "hello" 0 2)  
;;=> "he"
```

Referencing an instance field on a Java Object

You can call an instance field using the `.-` syntax:

```
(def p (java.awt.Point. 0 1))  
(.-x p)  
;;=> 0  
(.-y p)  
;;=> 1
```

Creating a new Java object

You can create instance of objects in one of two ways:

```
(java.awt.Point. 0 1)  
;;=> => #object[java.awt.Point 0x3776d535 "java.awt.Point[x=0,y=1]" ]
```

Or

```
(new java.awt.Point 0 1)
;;=> => #object[java.awt.Point 0x3776d535 "java.awt.Point[x=0,y=1]"]
```

Calling a static method

You can call static methods like this:

```
(System/currentTimeMillis)
;;=> 1469493415265
```

Or pass in arguments, like this:

```
(System/setProperty "foo" "42")
;;=> nil
(System/getProperty "foo")
;;=> "42"
```

Calling a Clojure function from Java

You can call a Clojure function from Java code by looking up the function and invoking it:

```
IFn times = Clojure.var("clojure.core", "*");
times.invoke(2, 2);
```

This looks up the `*` function from the `clojure.core` namespace and invokes it with the arguments 2 & 2.

Read Java interop online: <https://riptutorial.com/clojure/topic/4036/java-interop>

Chapter 15: Macros

Syntax

- The `'` symbol used in the `macroexpand` example is just syntactic sugar for the `quote` operator. You could have written `(macroexpand (quote (infix 1 + 2)))` instead.

Remarks

Macros are just functions that run at compile time, i.e. during the `eval` step in a `read-eval-print-loop`.

Reader macros are another form of macro that gets expanded at read time, rather than compile time.

Best practice when defining macro.

- alpha-renaming, Since macro is expand binding name conflict could arise. Binding conflict is not very intuitive to solve when using the macro. This is why, whenever a macro adds a binding to the scope, it is mandatory to use the `#` at the end of each symbol.

Examples

Simple Infix Macro

Clojure uses prefix notation, that is: The operator comes before its operands.

For example, a simple sum of two numbers would be:

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

Macros allow you to manipulate the Clojure language to a certain degree. For example, you could implement a macro that let you write code in infix notation (e.g., `1 + 2`):

```
(defmacro infix [first-operand operator second-operand]
  "Converts an infix expression into a prefix expression"
  (list operator first-operand second-operand))
```

Let's break down what the code above does:

- `defmacro` is a *special form* you use to define a macro.
- `infix` is the name of the macro we are defining.
- `[first-operand operator second-operand]` are the parameters this macro expects to receive when it is called.
- `(list operator first-operand second-operand)` is the body of our macro. It simply creates a

`list` with the values of the parameters provided to the `infix` macro and returns that.

`defmacro` is a *special form* because it behaves a little differently compared to other Clojure constructs: Its parameters are not immediately evaluated (when we call the macro). This is what allows us to write something like:

```
(infix 1 + 2)
;; => 3
```

The `infix` macro will expand the `1 + 2` arguments into `(+ 1 2)`, which is a valid Clojure form that can be evaluated.

If you want to see what the `infix` macro generates, you can use the `macroexpand` operator:

```
(macroexpand '(infix 1 + 2))
;; => (+ 1 2)
```

`macroexpand`, as implied by its name, will expand the macro (in this case, it will use the `infix` macro to transform `1 + 2` into `(+ 1 2)`) but won't allow the result of the macro expansion to be evaluated by Clojure's interpreter.

Syntax quoting and unquoting

Example from the standard library (core.clj:807):

```
(defmacro and
  "Evaluates exprs one at a time, from left to right. If a form
  returns logical false (nil or false), and returns that value and
  doesn't evaluate any of the other expressions, otherwise it returns
  the value of the last expr. (and) returns true."
  {:added "1.0"}
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
       (if and# (and ~@next) and#))))
```

- ``` called syntax-quote is like `(quote)`, but recursive: it causes `(let ...)`, `(if ...)`, etc to not evaluate during macro expansion but to output as is
- `~` aka unquote cancels syntax-quote for single form inside syntax-quoted form. So `x`'s value is outputted when expanding macro (instead of outputting `x` symbol)
- `~@` aka unquote-splicing is like unquote but takes list argument and expands it, each list item to separate form
- `#` appends unique id to symbols to prevent name conflicts. It appends the same id for the same symbol inside syntax-quoted expression, so `and#` inside `let` and `and#` inside `if` will get the same name

Read Macros online: <https://riptutorial.com/clojure/topic/2322/macros>

Chapter 16: Parsing logs with clojure

Examples

Parse a line of log with record & regex

```
(defrecord Logline [datetime action user id])
(def pattern #"(\d{8}-\d{2}:\d{2}:\d{2}.\d{3})\|.*\|(\w*),(\w*),(\d*)")
(defn parser [line]
  (if-let [[_ dt a u i] (re-find pattern line)]
    (->Logline dt a u i)))
```

Define a sample line :

```
(def sample "20170426-17:20:04.005|bip.com|1.0.0|alert|Update, john, 12")
```

Test it :

```
(parser sample)
```

Result :

```
#user.Logline{:datetime "20170426-17:20:04.005", :action "Update", :user "john", :id "12"}
```

Read Parsing logs with clojure online: <https://riptutorial.com/clojure/topic/9822/parsing-logs-with-clojure>

Chapter 17: Pattern Matching with core.match

Remarks

The `core.match` library implements a pattern match compilation algorithm that uses the notion of "necessity" from lazy pattern matching.

Examples

Matching Literals

```
(let [x true
      y true
      z true]
  (match [x y z]
    [_ false true] 1
    [false true _] 2
    [_ _ false] 3
    [_ _ true] 4))

;=> 4
```

Matching a Vector

```
(let [v [1 2 3]]
  (match [v]
    [[1 1 1]] :a0
    [[1 _ 1]] :a1
    [[1 2 _]] :a2)) ;; _ is used for wildcard matching

;=> :a2
```

Matching a Map

```
(let [x {:a 1 :b 1}]
  (match [x]
    [{:a _ :b 2}] :a0
    [{:a 1 :b _}] :a1
    [{:x 3 :y _ :z 4}] :a2))

;=> :a1
```

Matching a literal symbol

```
(match [['asymbol]]
  [['asymbol]] :success)

;=> :success
```

Read Pattern Matching with `core.match` online: <https://riptutorial.com/clojure/topic/2569/pattern-matching-with-core-match>

Chapter 18: Performing Simple Mathematical Operations

Introduction

This is how you would add some numbers in Clojure syntax. Since the method occurs as the first argument in the list, we are evaluating the + (or addition) method on the rest of the arguments in the list.

Remarks

Performing mathematical operations is the basis of manipulating data and working with lists. Therefore, understand how it works is key to progressing in one's understanding of Clojure.

Examples

Math Examples

```
;; returns 3
(+ 1 2)

;; returns 300
(+ 50 210 40)

;; returns 2
(/ 8 4)
```

Read Performing Simple Mathematical Operations online:

<https://riptutorial.com/clojure/topic/8901/performing-simple-mathematical-operations>

Chapter 19: Setting up your development environment

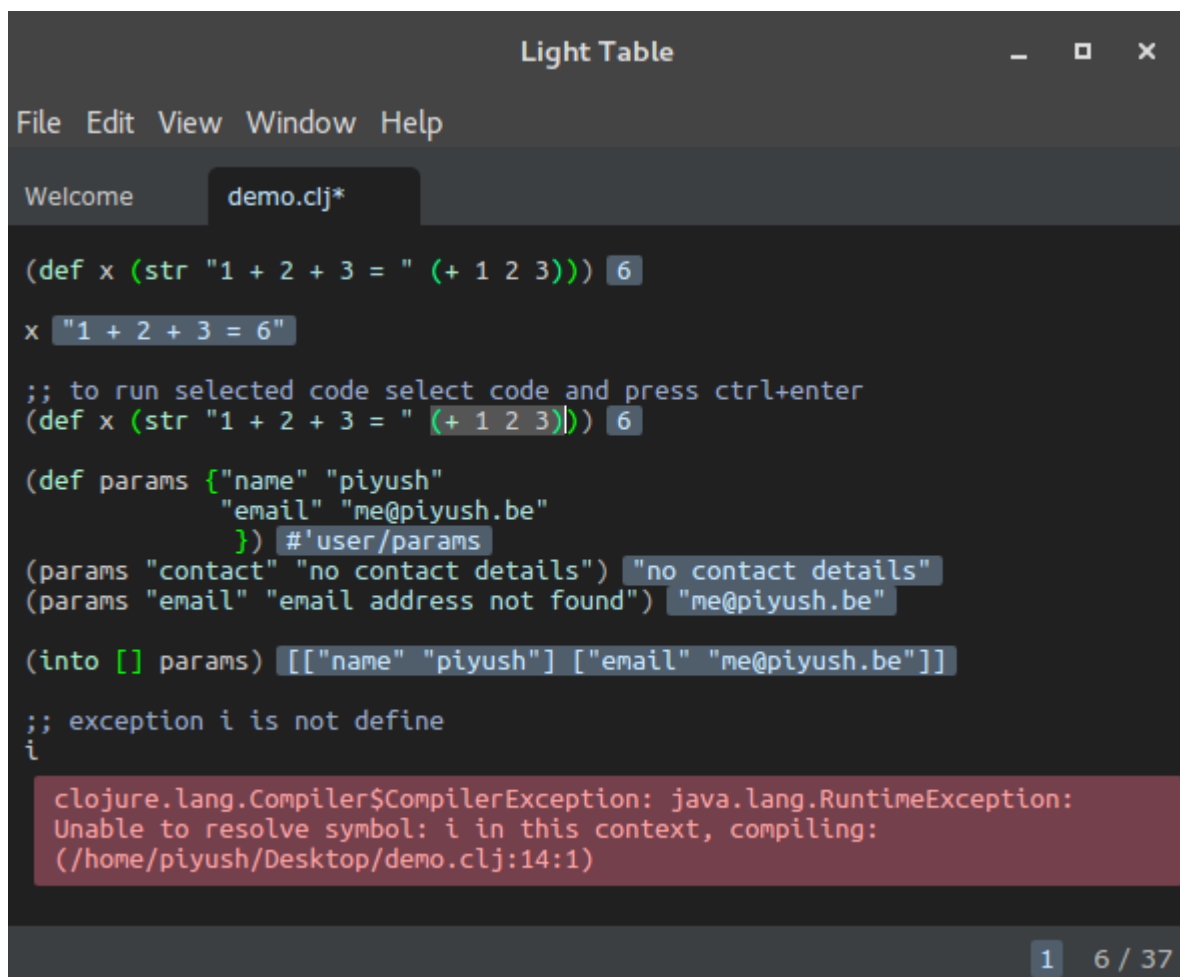
Examples

Light Table

Light Table is a good editor to learn, experiment and run Clojure projects.

You can also run lein/boot projects by opening `project.clj` file. It will load all project dependencies.

It supports inline evaluation, plugins and much more, so there's no need to add print statements and check the output in the console. You can run individual lines or code blocks by pressing `ctrl + enter`. To run partial code, select the code and press `ctrl + enter`. check the following screenshot for how you can use Light Table to learn and experiment with Clojure code.

A screenshot of the Light Table application window. The title bar says "Light Table". The menu bar includes "File", "Edit", "View", "Window", and "Help". Below the menu bar, there are two tabs: "Welcome" and "demo.clj*". The main editor area contains Clojure code. The first line is `(def x (str "1 + 2 + 3 = " (+ 1 2 3)))` followed by a small box containing the number 6. Below this, the text `x "1 + 2 + 3 = 6"` is shown. Then, a comment `;; to run selected code select code and press ctrl+enter` is present. This is followed by another line of code: `(def x (str "1 + 2 + 3 = " (+ 1 2 3)))` with a box containing 6. Next, a `def` block defines `params` with a map containing "name" and "email". Below this, two `params` function calls are shown with their respective outputs in boxes. Then, an `into` function call is shown with its output in a box. Finally, there is a comment `;; exception i is not define` followed by the letter `i`. At the bottom of the editor, a red box displays an exception message: `clojure.lang.Compiler$CompilerException: java.lang.RuntimeException: Unable to resolve symbol: i in this context, compiling: (/home/piyush/Desktop/demo.clj:14:1)`. The bottom right corner of the window shows a status bar with "1" and "6 / 37".

Pre-built binaries of Light Table can be found [here](#). No further setup is required.

Light Table is able to automatically locate your Leiningen project and evaluate your code. If you don't have Leiningen installed, install it using the instructions [here](#).

Documentation: docs.lighttable.com

Emacs

To setup Emacs for working with Clojure, install `clojure-mode` and `cider` package from melpa:

```
M-x package-install [RET] clojure-mode [RET]
M-x package-install [RET] cider [RET]
```

Now when you open a `.clj` file, run `M-x cider-jack-in` to connect to a REPL. Alternatively, you can use `C-u M-x (cider-jack-in)` to specify the name of a `lein` or `boot` project, without having to visit any file in it. You should now be able to evaluate expressions in your file using `C-x C-e`.

Editing code in lisp-like languages is much more comfortable with a paren-aware editing plugin. Emacs has several good options.

- `paredit` A classic Lisp editing mode that has a steeper learning curve, but provides a lot of power once mastered.

```
M-x package-install [RET] paredit [RET]
```

- `smartparens` A newer project with similar goals and usage to `paredit`, but also provides reduced capabilities with non-Lisp languages.

```
M-x package-install [RET] smartparens [RET]
```

- `parinfer` A much simpler Lisp editing mode that operates mainly via inferring proper paren nesting from indentation.

Installation is more involved, see the Github page for `parinfer-mode` for [setup instructions](#).

To enable `paredit` in `clojure-mode`:

```
(add-hook 'clojure-mode-hook #'paredit-mode)
```

To enable `smartparens` in `clojure-mode`:

```
(add-hook 'clojure-mode-hook #'smartparens-strict-mode)
```

Atom

Install Atom for your distribution [here](#).

After that run the following commands from a terminal:

```
apm install parinfer
apm install language-clojure
apm install proto-repl
```

IntelliJ IDEA + Cursive

[Download](#) and install the latest IDEA version.

[Download](#) and install the latest version of the Cursive plugin.

After restarting IDEA, Cursive should be working out of the box. Follow the [user guide](#) to fine-tune appearance, keybindings, code style etc.

Note: Like [IntelliJ](#), [Cursive](#) is a commercial product, with a 30-day evaluation period. Unlike [IntelliJ](#), it doesn't have a community edition. Free non-commercial licences are available to individuals for non-commercial use, including personal hacking, open-source and student work. The licence is valid for 6 months and can be renewed.

Spacemacs + CIDER

[Spacemacs](#) is a distribution of emacs that comes with a lot of packages preconfigured and easily installed. Also, it is very friendly for those who are familiar with vim style of editing. Spacemacs provides a [CIDER-based Clojure layer](#).

To install and configure it for use with Clojure, first install emacs. Then make a backup of your previous configurations:

```
$ mv ~/.emacs.d ~/.emacs.d.backup
```

Then clone the spacemacs' repository:

```
$ git clone https://github.com/syl20bnr/spacemacs ~/.emacs.d
```

Now, open emacs. It will ask you some questions regarding your preferences. Then it downloads more packages and configures your emacs. After that the spacemacs is installed, you are ready to add Clojure support. Press `SPC f e d` to open your `.spacemacs` file for configuration. Find `dotspacemacs-configuration-layers` in the file, under it there is an open paren. Anywhere in between the parens in a new line type `clojure`.

```
(defun dotspacemacs/layers ()
  (setq-default
    ;; ...
    dotspacemacs-configuration-layers
    '(clojure
      ;; ...
    )
    ;; ...
  ))
```

Press `SPC f e R` to save and install the clojure layer. Now, in any `.clj` file if you press `, s i` spacemacs will try to spawn a new clojure REPL connection to your project, and if it succeeds it will show in the statusbar, which afterwards you can press `, s s` to open a new REPL buffer to evaluate your codes.

For more information about spacemacs and cider contact their documentations. [Spacemacs docs](#), [Cider docs](#)

Vim

Install the following plugins using your favourite plugin manager:

1. [fireplace.vim](#): Clojure REPL support
2. [vim-sexp](#): For taming [those hugs around your function calls](#)
3. [vim-sexp-mappings-for-regular-people](#): Modified sexp mappings that are a little easier to bear
4. [vim-surround](#): Easily delete, change, add "surroundings" in pair
5. [salve.vim](#): Static Vim support for Leiningen and Boot.
6. [rainbow_parentheses.vim](#): Better Rainbow Parentheses

and also relate to syntax highlighting, omni completion, advanced highlighting and so on:

1. [vim-clojure-static](#) (if you have a vim older than 7.3.803, newer versions ship with this)
2. [vim-clojure-highlight](#)

Other options in place of vim-sexp include [paredit.vim](#) and [vim-parinfer](#).

Read [Setting up your development environment online](#):

<https://riptutorial.com/clojure/topic/1387/setting-up-your-development-environment>

Chapter 20: Threading Macros

Introduction

Also known as arrow macros, threading macros convert nested function calls into a linear flow of function calls.

Examples

Thread Last (->>)

This macro gives the output of a given line as the last argument of the next line function call. For e.g.

```
(prn (str (+ 2 3)))
```

is same as

```
(->> 2  
  (+ 3)  
  (str)  
  (prn))
```

Thread First (->)

This macro gives the output of a given line as the first argument of the next line function call. For e.g.

```
(rename-keys (assoc {:a 1} :b 1) {:b :new-b}))
```

Can't understand anything, right? Lets try again, with ->

```
(-> {:a 1}  
  (assoc :b 1)                ;; (assoc map key val)  
  (rename-keys {:b :new-b}))  ;; (rename-keys map key-newkey-map)
```

Thread as (as->)

This is a more flexible alternative to thread first or thread last. It can be inserted anywhere in the list of parameters of the function.

```
(as-> [1 2] x  
  (map #(+ 1 %) x)  
  (if (> (count x) 2) "Large" "Small"))
```


Read Threading Macros online: <https://riptutorial.com/clojure/topic/9582/threading-macros>

Chapter 21: Transducers

Introduction

Transducers are composable components for processing data independently of the context. So they can be used to process collections, streams, channels, etc. without knowledge of their input sources or output sinks.

The Clojure core library was extended in 1.7 so that the sequence functions like `map`, `filter`, `take`, etc. return a transducer when called without a sequence. Because transducers are functions with specific contracts, they can be composed using the normal `comp` function.

Remarks

Transducers allow the laziness to be controlled as they are consumed. For example `into` is eager as would be expected, but `sequence` will lazily consume the sequence through the transducer. However, the laziness guarantee is different. Enough of the source will be consumed to produce an element initially:

```
(take 0 (sequence (map #(do (prn '-> %) %)) (range 5)))  
;; -> 0  
;; => ()
```

Or decide if the list is empty:

```
(take 0 (sequence (comp (map #(do (prn '-> %) %)) (remove number?)) (range 5)))  
;; -> 0  
;; -> 1  
;; -> 2  
;; -> 3  
;; -> 4  
;; => ()
```

Which differs from the usual lazy sequence behaviour:

```
(take 0 (map #(do (prn '-> %) %) (range 5)))  
;; => ()
```

Examples

Small transducer applied to a vector

```
(let [xf (comp  
          (map inc)  
          (filter even?))]  
  (transduce xf + [1 2 3 4 5 6 7 8 9 10]))  
;; => 30
```

This example creates a transducer assigned to the local `xf` and uses `transduce` to apply it to some data. The transducer adds one to each of its inputs and only returns the even numbers.

`transduce` is like `reduce`, and collapses the input collection to a single value using the provided `+` function.

This reads like the `thread-last` macro, but separates the input data from the computations.

```
(->> [1 2 3 4 5 6 7 8 9 10]
      (map inc)
      (filter even?)
      (reduce +))
;; => 30
```

Applying transducers

```
(def xf (filter keyword?))
```

Apply to a collection, returning a sequence:

```
(sequence xf [:a 1 2 :b :c]) ;; => (:a :b :c)
```

Apply to a collection, reducing the resulting collection with another function:

```
(transduce xf str [:a 1 2 :b :c]) ;; => ":a:b:c"
```

Apply to a collection, and `conj` the result into another collection:

```
(into [] xf [:a 1 2 :b :c]) ;; => [:a :b :c]
```

Create a core async channel that uses a transducer to filter messages:

```
(require '[clojure.core.async :refer [chan >!! <!! poll!]])
(doseq [e [:a 1 2 :b :c]] (>!! ch e))
(<!! ch) ;; => :a
(<!! ch) ;; => :b
(<!! ch) ;; => :c
(poll! ch) ;;=> nil
```

Creating/Using Transducers

So the most used functions on Clojure `map` and `filter` have been modified to return transducers (composable algorithmic transformations), if not called with a collection. That means:

`(map inc)` returns a transducer and so does `(filter odd?)`

The advantage: the functions can be composed into a single function by `comp`, which means traversing the collection just once. Saves run time by over 50% in some scenarios.

Definition:

```
(def composed-fn (comp (map inc) (filter odd?)))
```

Usage:

```
;; So instead of doing this:  
(->> [1 8 3 10 5]  
      (map inc)  
      (filter odd?))  
;; Output [9 11]  
  
;; We do this:  
(into [] composed-fn [1 8 3 10 5])  
;; Output: [9 11]
```

Read Transducers online: <https://riptutorial.com/clojure/topic/10814/transducers>

Chapter 22: Truthiness

Examples

Truthiness

In Clojure everything that is not `nil` or `false` is considered logical true.

Examples:

```
(boolean nil)           ;=> false
(boolean false)         ;=> false
(boolean true)          ;=> true
(boolean :a)            ;=> true
(boolean "false")       ;=> true
(boolean 0)             ;=> true
(boolean "")            ;=> true
(boolean [])            ;=> true
(boolean '())           ;=> true

(filter identity [:a false :b true]) ;=> (:a :b true)
(remove identity [:a false :b true]) ;=> (false)
```

Booleans

Any value in Clojure is considered truthy unless it is `false` or `nil`. You can find the truthiness of a value with `(boolean value)`. You can find the truthiness of a list of values using `(or)`, which returns `true` if any arguments are truthy, or `(and)` which returns `true` if all arguments are truthy.

```
=> (or false nil)
nil      ; none are truthy
=> (and '() [] {} #{} "" :x 0 1 true)
true     ; all are truthy
=> (boolean "false")
true     ; because naturally, all strings are truthy
```

Read Truthiness online: <https://riptutorial.com/clojure/topic/4116/truthiness>

Chapter 23: Vars

Syntax

- (def symbol value)
- (def symbol "docstring" value)
- (declare symbol_0 symbol_1 symbol_2 ...)

Remarks

This should not be confused with `(defn)`, which is used for defining functions.

Examples

Types of Variables

There are different types of variables in Clojure:

- numbers

Types of numbers:

- integers
- longs (numbers larger than $2^{31} - 1$)
- floats (decimals)

- strings

- collections

Types of collections:

- maps
- sequences
- vectors

- functions

Read Vars online: <https://riptutorial.com/clojure/topic/4449/vars>

Credits

S. No	Chapters	Contributors
1	Getting started with clojure	adairdavid , Adeel Ansari , alejosocorro , Alex Miller , Arclite , avichalp , Blake Miller , Community , CP9 , D-side , Geoff , Greg , KettuJKL , Kiran , Martin Janiczek , n2o , Nikita Prokopov , Sajjad , Sam Estep , Sean Allred , Zaz
2	Atom	Qwerp-Derp , systemfreund
3	clj-time	Rishu Saniya , Akanksha , Mrinal Saurabh , Vishakha Silky
4	Clojure destructuring	camdez , kaffein , kolen , leeor , Michał Marczyk , MuSaiXi , r00tt , RedBlueThing , ryo , tsleyson , Zaz
5	clojure.core	Akanksha , Mrinal Saurabh , Surbhi Garg
6	clojure.spec	Adam Lee , Alex Miller , kolen , leeor , nXqd
7	clojure.test	jisaw , kolen , leeor , porglezomp , Sam Estep
8	Collections and Sequences	Alex Miller , Kenogu Labz , nXqd , Sam Estep
9	core.async	Valentin Waeselynck
10	Emacs CIDER	avichalp
11	File Operations	Zaz
12	Functions	alejosocorro , fokz , Qwerp-Derp , tar , tsleyson
13	Getting started with web development	Emin Tham , kolen , r00tt
14	Java interop	leeor
15	Macros	Alex Miller , kolen , mathk , Sam Estep , snowcrshd
16	Parsing logs with clojure	user2611740
17	Pattern Matching with core.match	Kiran
18	Performing Simple	Jim

	Mathematical Operations	
19	Setting up your development environment	Adeel Ansari , agent_orange , amallo , g1eny0ung , Geoff , Kiran , kolen , MuSaiXi , Piyush , Qwerp-Derp , spinningarrow , stardiviner , superkondukt , swlkr
20	Threading Macros	Kusum Ijari , Mrinal Saurabh
21	Transducers	l0st3d , Mrinal Saurabh
22	Truthiness	Alan Thompson , Michiel Borkent , Zaz
23	Vars	Aryaman Arora , Qwerp-Derp , Stephen Leppik , Zaz