

Introduction to clojure

Evan Misshula

2017-05

Outline

- 1 Why learn to code?
 - Why you are here
- 2 Why clojure?
 - What is Clojure?
- 3 But first, we have to start small
 - Walking before we run
- 4 Data structures
 - our building blocks
- 5 Functions
 - change the world
- 6 Building a project
 - Clojure build tools
- 7 The turtles project
 - Imperative Programming through Turtles
 - Movement
 - Our first programs
 - variations on the theme

Why you are here

- You will understand more about how the world around you works
- You might be able to get a job coding
- You will understand how to break big problems into smaller ones
- You will eventually build websites and phone apps

What is Clojure?

- Clojure is a hosted language that can compile to Java or Javascript which allows us to build rich web or phone apps that run in many environments.
- Clojure is a functional programming language which allows us to build complex solutions from simple ones.
- Clojure has immutable data structures which makes it easier to reason about programs and helps with performance
- We can change the state of the program through a series of immutable values over time
- Clojure evaluates programs using its own types allowing us to write programs that write programs
- Clojure has an elegant way of doing asynchronous programming which is necessary both for large data and uncertain execution environments like the web

Walking before we run

- simple values
- booleans and nil
- keywords
- numbers and arithmetic

our building blocks

- lists
- vectors
- maps

- What are functions?
- Functions that take other functions
- Predicate functions
- Anonymous functions
- Pure functions

Clojure build tools

- Lennigen
- the REPL
- Namespaces

Imperative Programming through Turtles

- How to walk turtles
 - initial state
 - undo, clean and home
 - state
 - doc

- Basic Movement - forward, backward right and left
- Multiple turtles
 - tilt
- Add one more turtle and give them commands
- Move all five turtles - Introduction to functions
 - map higher order functions

Our first programs

- Write a function that adds turtles
- Functions with parameters
- control flow with `if` to check input

- Functions with five turtles moving in different directions
 - use `map`
 - use recursion

Data Structures revisited

- Immutable state
- Sequence and collection abstractions
- Lists, vectors, and hash-maps

- Sequence functions
- Key-value functions
- Lazy evaluation

Higher order functions

- map, reduce, filter, apply, etc.
- Solving problems functionally

Base data types

- Numbers

```
'(12  
+12  
-100)
```

```
(println (+ 9 1))
```

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


- **Keywords** are special words that name a part of a complicated data structure. They are clojures fast way to get to the data we need

```
:foobar
```

```
:2
```

```
:?
```

- **Symbols** evaluate to functions or variables

```
sample-symbol  
f1  
swap!
```

Strings

- **Strings** are ordered collections of characters. We denote a string by putting it in double quotes

```
(print "Hi Evan")
```

Characters

- **Characters** are also a type. A character is just a single letter
- it has a backslash in front of it.

Arithmetic in Clojure

- Computers were first used to do large arithmetic calculations.
- Clojure uses **prefix notation** which means the operation is at the beginning

`(+ 1 3)`

`;;NOT (1 + 3)`

Clojure has namespaces

- A namespace is a container that makes it easier to create unique names for functions and symbols

```
radio/change station  
clojure.core/-
```

- Strings put individual characters together but we will want to put numbers and letters together. Clojure has several compound data structures.
 - 1 a list `'(1 2 3 4)`
 - 2 a vector `[1 2 3 4]`
 - 3 a set which has only unique elements `#{1 2 3}`
 - 4 a map `{:evan 1, :ray 3, :jeff 2}` which maps names to values. In other languages this is called an associative array or a hash-map.

Clojure is functional

- Clojure is a functional language. In practical terms this means that it accomplishes its work by dividing its tasks onto functions which transform data from input to output. The next few slides will help us understand how to read and write our own functions.
- `defn` defines a named functions

```
(defn greet [name] (str "Welcome, " name))
```

- Our functions name is `greet`
- It takes a single parameter `'name'`
- It does not return anything, only `nil`
- It prints which is considered a side effect

Multi-arity

- A function can take different numbers of arguments
- Functions must be defined with `defn`
- Doing this twice replaces the function definition
 - Slow down! This is strange/weird
 - One arity c-a-n c-a-l-l a-n-o-t-h-e-r

```
(defn myGreeterWdefault
  ([]      (myGreeterWdefault "Where's lunch?"))
  [msg]    (println msg))
```

Variadic Functions

- *Variadic* functions have an undefined number of parameters which are collected into a sequence and used by the function. The ampersand & does not appear when you call the function.
- The collected arguments come after a '&'

```
(defn myGreetingWdefault [greeting & who]  
  (println greeting who))
```

```
(hello "Hello" "world" "class")
```

Anonymous functions

- *Anonymous* functions can be created and called for a single purpose
- can be defined with `fn` or `#()`
- So:

```
(=
  (fn [x y] (+ x y))
  #(+ %1 %2))
```

Anonymous and variadic

- These functions can be expressed as

```
(=
  (fn [x y & zs] (println x y zs))
  #(println %1 %2 %&))
```

Pure functions in Clojure

- *Side Effects* are ways in which the your program interacts with the outside world such as printing or writitng to and reading from disk. These are necessary but it is impossible to know if they will succeed.
- *Pure* functions are ones that just return a value and have no side effects.