Tutorial 1.3: Data Structures and Functions

1. Clojure data structures

First to start working with clojure, we need to know some basic data types and operations that could be perform on them.
These are some data types available in clojure, we will be adding new concepts and operations to our repertoire as we need but for the moment below is our list.
As you did with tutorial 1.2, please try all the examples while you are reading. The idea is you type them in your newly configured environment from tutorial 1.1. This will help you to do your assignments at the end of this document. Code with these examples to evaluate them right away in the REPL are at https://github.com/cneira/clojure-workshop1.

a. Strings

Strings are enclosed in double quotes, Clojure strings are java strings.

```
“I’m a string”
```

b. List

A list of elements is created like this.

```
‘(1 2 3 x t x s g )
```

Also you could type the following.

```
(list 1 2 3 'x 't 's 'g )
```

c. Vector

A vector is a sequential data structure with efficient random access lookup performance.
A vector has the following syntax, the keywords vector and vec are used to create them. You could store any kind of datatype inside a vector.

```
[1 2 3]
```

```
(vec [ 1 2 3 “green” ] )
```

```
(vector [ 1 2 3 “green” ] )
```

Vectors are functions, that means you could evaluate them to their indexes.

```
([1 2 3] 0 )
```

You could check if a symbol references a vector with the function.

```
(vector? [1 2 4 ] )
```

```
=> true
```

d. Hashmaps

http://clojure.org/data_structures#Data Structures-Maps (IPersistentMap)

A hashmap is a map data structure. A map is a set of distinct keys (or indexes) mapped (or associated) to values. The association is such that the map will return the value when given the key. Using a map called a hashmap allows near instant look up time. Clojure has also another similar structure called sorted-map as the name implies their keys are sorted.

For example, recording a character's status in a game a hashmap could come in handy as follows.

```
{:str 13 :char 10  :dex  20 :wis 8   }
```

You could also type the following.

```
(hash-map  :str 13 :char 10  :dex  20 :wis 8  )
```

```
=> {:str 13, :dex 20, :wis 8, :char 10}
```

```
(sorted-map  :str 13 :char 10  :dex  20 :wis 8  )
```

```
=> {:char 10, :dex 20, :str 13, :wis 8}
```

Maps are also functions so evaluating this map to one of the keys we get the value.

```
( {:str 13 :char 10  :dex  20 :wis 8   }  : str )
```

```
=> 13
```

Maybe you want the keys from a map.

```
(keys  {:str 13 :char 10  :dex  20 :wis 8   }  )
```

```
(:str :dex :wis :char)
```

Or just the values.

```
(vals    {:str 13 :char 10  :dex  20 :wis 8   }  )

=> (13 20 8 10)
```

e. Sets

A set is a data structure that does not have repeated elements and it their elements have no particular order.

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

As with maps there is a hash and sorted version of this data structure.

```
(hash-set  :a 1 :b 2 :c 1  4  )

=> #{1 4 :c 2 :b :a}

(sorted-set   999 99 99 11 223 444 55 6677 88 99  )

=> #{11 55 88 99 223 444 999 6677}
```

To check if an element exists in a set, just evaluated to the element.

```
((hash-set  :a 1 :b 2 :c 1  4  )  1 )
=> 1

((hash-set  :a 1 :b 2 :c 1  4  )  "monday" )
=> nil
```

You could create a set ordered right away, the > is the comparator function that is applied to every element, this set is ordered from greatest to least.
A comparator is a function that takes two arguments x and y, and returns a value indicating the relative order in which the elements x and y should be sorted.

```
(sorted-set-by  >   2 3 4 5 6 9 10 )

=> #{10 9 6 5 4 3 2}
```

Let's create a simple comparator and use it. This simple function will return true or false if the element being evaluated is a pair.

```
(defn test_comparator [a b ]
    (if (and (even? b ) ( > b a ) false true )))
```

```
( sorted-set-by  test_comparator   2 3 4 5 6 9 1)

=> #{9 6 1 5 4 3 2}
```

f. Keywords

Are used to access the values they make reference in collections.

(:str {:str 10 :dex 9 :wis 18 })
=> 10

g. Collections

Vectors, maps, sets are collections. All Clojure collections are immutable, they are persistent. All collections support count, conj, and seq.

• count – returns the number of items in a collection.
• conj – add items to the collection; where the item is added depends on the collection type.
• seq – returns a seq on the collection; if the collection is empty returns nil.

As all collections support seq, we can use the seq group of functions to go through the entire collection.

Try the following in your REPL.

```
(def  a  (str/split-lines (slurp  "resources/people.txt") ))

(count a )

=> 101

( def b  (conj a  "102,john,doe,doe@none.org,127.0.01") )

(count b)
=> 102
```

What is a sequence ?

It is a way to sequentially consume a succession of data, is a sequential view of a collection or collection-like entity.

The sequence abstraction supports the following operations:

- first
- rest
- next

You can create sequences using the following functions.

- seq produces a sequence using a collection as parameter
- lazy-seq creates a lazy sequence

What is a lazy sequence?

It means that a value is not computed until it is needed, once the value is computed is it cached so future calls don't need to recalculate again. This is also called call-by-need.

What are the benefits of lazy evaluation?

- Performance increases by avoiding needless calculations and error conditions in evaluation compound expressions.
- The ability to construct potentially infinite data structures

2. Defining functions

a. Functions

Functions in clojure are first-class values this means that the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables. To define a variable we will use a special form called fn.

```
(fn [x] ( + 20  x ))
```

To define a one parameter function, we could associate this to a symbol.

```
(def f  (fn [x] (+ 20))
```

So we just could call it like the following.

```
(f 20)
=> 40
```

The macro *defn* combines the fn and def calls to make it easier to define functions and register them in the current namespace.

```
(defn f [x]  (+ 20 x ))
```

We could also define multi arity functions.

```
(defn f
 ([x] ( + 30 x ))
 ([ x y ]  ( * x y )))

=> #'repl-tutorial.core/f

(f 3 )
=> 33
(f 3 2)
=> 6
```

There is also a construct to define mutually recursive local functions.

```
(letfn [ (a [x]  (b x ) )
         (b [x]  (c 4 x))
         (c [x y]  (+ x y ))
        ]
  (a 3.3)
  )
=> 7.3
```

Now we need to define what a special form is.
"Special forms are those expressions in the Lisp language which do not follow normal rules for evaluation. Some such forms are necessary as primitives of the language, while others may be desirable in order to improve readability, control the evaluation environment, implement abstraction and modularity, affect the flow of control, allow extended scoping mechanisms, define functions which accept a variable number of arguments, or achieve greater efficiency. " (http://www.nhplace.com/kent/Papers/Special-Forms.html)
We are going to take a look at a few special forms, that will be helpful to create our app,
to check the rest of the special forms refer to the clojure documentation : http://clojure.org/special_forms. Let's see some examples of special forms:

b. (if test then else?)

Evaluates test. if test is not nil nor false evaluates then otherwise evaluates else?

(if (true? nil) --> **test** (*true? returns true if argument is true, false otherwise.)*
"yes" --> then
"no") --> else ?

=> "no"

*“In Clojure **nil** means 'nothing'. It signifies the absence of a value, of any type, and is not specific to lists or sequences.” (http://clojure.org/lisps)*

c. (do exprs)*

Evaluates the expressions in order and returns the value of the last. If no expressions are supplied, returns nil.

```
(do "cat" "mouse" "trap" "poison")

=>  “poison”
```

d. (let [bindings] exprs*)*

Sequentially associates symbols to expressions, an inner let binding knows about the symbols bounded in the outer let form.

`(let [ cat "tom" mouse "jerry" ]` -> associates symbol cat to “tom”, symbol mouse to “jerry”

```
  (let [ watcher "bill" ]
       (println  watcher "knows about " cat  “and” mouse)))

=> bill knows about  tom and jerry
```

Assignments:

1. Using the people.txt file provided in the previous workshop, create a function that searches by name, ip, last name or email and return the result in a hashmap.

2. Represent the people.txt file records as a list of hashmaps, sort them by first_name.(hint : the map function will come in handy for this).

3. Using the people.txt file, represent the records of this file as hashmaps and add the age property (keyword :age) on the records, it's value should be random.

4. Using the hashmap created in 3, create a function that returns the persons that should be deceased (age >= 100) as a list of hashmaps.

5. Create a function that takes as parameter two hashmaps each one is a representation of a record from the persons.txt return the person with the longest name.

Author: Carlos Antonio Neira Bustos
Email : cneirabustos@gmail.com