

# Tutorial 1.2: The REPL

## 1. Interacting with the REPL

What is the repl? REPL is the acronym for read-eval-print-loop, this is also called the reader. Basically it does the following:

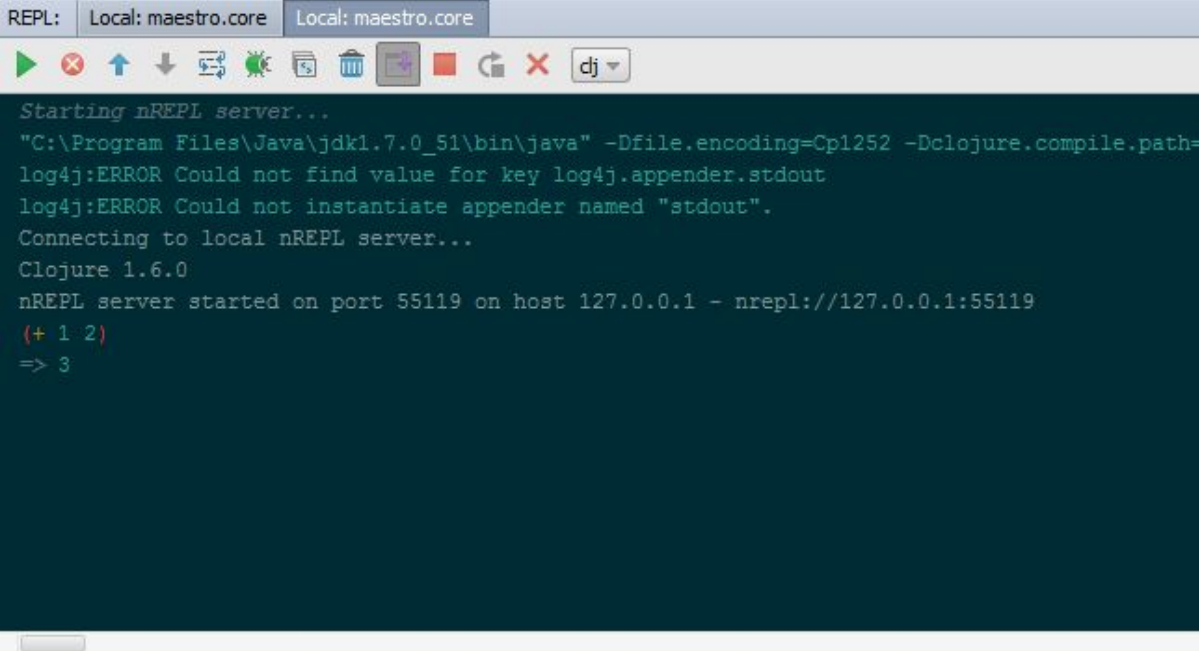
- **Read** code as text from a source.
- **Evaluate** the code to a value.
- **Print** the value from the evaluation to an output device.
- **Loop** start all over again.

You will be using this a lot as it's really useful to test your functions and test your ideas, you just enter statements and they are evaluated right a away. The performance and results is the same as it will result in your application. All the code submitted to the REPL is compiled to java bytecode right away!

This tutorial includes several examples for you to try them 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.

To begin using the repl just right click on your project and choose Run 'REPL...' . We will use the repl supplied by Leiningen. For starters type :

( + 1 2 )



```
REPL: Local: maestro.core Local: maestro.core
Starting nREPL server...
"C:\Program Files\Java\jdk1.7.0_51\bin\java" -Dfile.encoding=Cp1252 -Dclojure.compile.path=
log4j:ERROR Could not find value for key log4j.appender.stdout
log4j:ERROR Could not instantiate appender named "stdout".
Connecting to local nREPL server...
Clojure 1.6.0
nREPL server started on port 55119 on host 127.0.0.1 - nrepl://127.0.0.1:55119
(+ 1 2)
=> 3
```

As clojure is a lisp dialect it has prefix notation ( [https://en.wikipedia.org/wiki/Polish\\_notation](https://en.wikipedia.org/wiki/Polish_notation) ), translating this line would read to something like this: *“evaluate the call to function add with parameters one and two.”*

This is called a s-expression (symbolic expression) which is a notation for nested list data (<https://en.wikipedia.org/wiki/S-expression> ).

All expressions are evaluated to a single value following these rules :

- Lists (denoted by parentheses for ex : (+ 1 2 )) are calls, the first value in the list is the operator and the rest of the values are the parameters.
- Symbols evaluate to the named value in the current scope (will talk about namespaces later). What are symbols? In clojure a symbol is just a name, the clojure compiler will look for the name and will try to resolve it as a java class name, a local, a variable or a variable from another namespace.  
What is the difference between a symbol and a variable? A symbol is a name which associates an entity with a value. A variable does not have a name a symbol is used to associate a name it. To declare a variable in clojure we use the following s-expression :

```
(def a 1)
```

We create a variable that has the value 1 and associated it with the symbol a - when a symbol is said to be bound it means it is associated with a value. Why is this separation made? It's part of the code-as-data mantra of a lisp language and it is essential for macros that process code as data.

- All other expressions evaluate to their literal values for example: entering 1 is evaluated to 1.

## 2. Avoiding evaluation:

There are times when we need to avoid evaluation on the REPL. This is simply done by quoting our s-expr. Quoting can be done using ' or using the quote symbol, for example:

```
(def a '(+ 1 2))
```

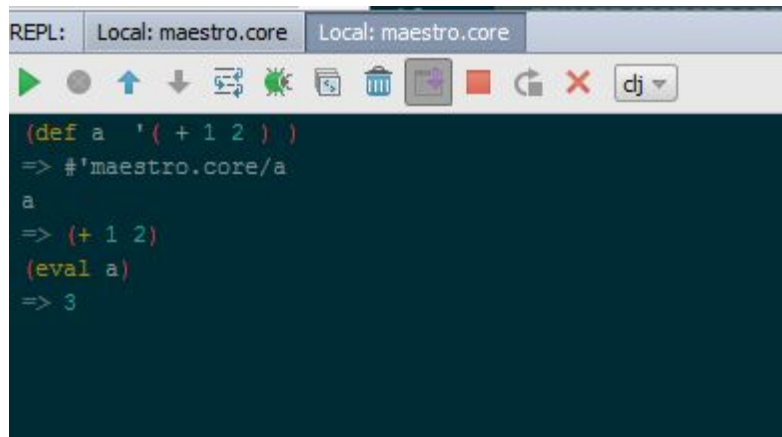
```
(def a (quote (+ 1 2)))
```

Evaluating symbol a will result in:

```
(+ 1 2)
```

This is data and also code, but when we quote it we are saying that this is data, but as this is also code, Could we evaluate it?. Of course typing eval will evaluate this data:

```
( eval a )  
=> 3
```

A screenshot of a REPL window with two tabs labeled 'Local: maestro.core'. The window has a toolbar with various icons. The code entered is: 

```
(def a '( + 1 2 ) )  
=> #'maestro.core/a  
a  
=> (+ 1 2)  
(eval a)  
=> 3
```

### 3. Handy REPL bound variables :

The REPL has a couple of variables that are only available during the repl session. These come in handy when evaluating your code in the REPL.

- \*1, \*2, and \*3 hold the values of the most recently evaluated expression.

For example in our recent test it should hold 3, to check this just type in the repl :

```
*1  
= 3
```

- \*e has the last caught exception (java exception) in the repl session, in our case nil.

```
*e  
=> nil
```

```
REPL: Local: maestro.core Local: maestro.core
=> #'maestro.core/a
a
=> (+ 1 2)
(eval a)
=> 3
*e
=> 3
*e
=> #<CompilerException java.lang.RuntimeException: Too many arguments to def,
```

Here my last exception caused by I typed the following invalid s-expression:  
(def a b d )

```
(def a b d )
CompilerException java.lang.RuntimeException: Too many arguments to def,
```

So in my RELP \*e symbol contains this exception.  
If we want to check the stack trace for this exception we use the pst function:

(pst \*e)

```
(pst *e)
CompilerException java.lang.RuntimeException: Unable to resolve symbol: date in this context, co
Caused by:
RuntimeException Unable to resolve symbol: date in this context <6 internal calls>
=> nil
```

Now just click in <6 internal calls> and you will see them:

```
(pst *e)
CompilerException java.lang.RuntimeException: Unable to resolve symbol
Caused by:
RuntimeException Unable to resolve symbol: date in this context
  clojure.lang.Util.runtimeException (Util.java:221)
  clojure.lang.Compiler.resolveIn (Compiler.java:6940)
  clojure.lang.Compiler.resolve (Compiler.java:6884)
  clojure.lang.Compiler.analyzeSymbol (Compiler.java:6845)
  clojure.lang.Compiler.analyze (Compiler.java:6427)
  clojure.lang.Compiler.analyze (Compiler.java:6406)
=> nil
```

Alternatively, to save time you could click on this symbol from the REPL toolbar



that will print the last exception into the REPL.

#### 4. Getting help

Clojure has a very useful function called `doc`. For example, if we want to know why our previous evaluation of `(def a b d)` failed, we have access to the documentation here in the REPL, the syntax is the following:

```
(doc <symbol > )
```

```
(doc doc)
-----
clojure.repl/doc
([name])
Macro
  Prints documentation for a var or special form given its name
=> nil
```

Let's check what `def` does:

```
(doc def )
```

```
(doc def )
-----
def
  (def symbol doc-string? init?)
Special Form
  Creates and interns a global var with the name
  of symbol in the current namespace (*ns*) or locates such a var if
  it already exists. If init is supplied, it is evaluated, and the
  root binding of the var is set to the resulting value. If init is
  not supplied, the root binding of the var is unaffected.

  Please see http://clojure.org/special\_forms#def
=> nil
```

Here it says that it has a `doc-string` (it's a documentation string by the programmer) parameter, this is what the `doc` function reads from the symbols, let's create a variable and associate a symbol to it, and also add a comment to it and also other symbol without a `doc-string`. As we will see evaluating `doc` returns `nil` when a `doc-string` is missing.

```
(def good "this symbol holds nothing important" "nothing important" )
```

```
(doc good )
```

```
(def bad "no doc for you)
```

```
(doc bad)
```

```
(def good "this symbol holds nothing important" "nothing important")
=> #'maestro.core/good
a
=> (+ 1 2)
good
=> "nothing important"
(doc good)
-----
maestro.core/good
  this symbol holds nothing important
=> nil
(def bad "no doc for you")
=> #'maestro.core/bad
(doc bad)
-----
maestro.core/bad
  nil
=> nil
```

If we do not know the full name of the symbol we want to check for a doc-string we could look for it using using the find-doc function this will search for all references of the symbol in the documentation.

For example

```
(find-doc "range" )
```

```
(find-doc "range" )
-----
clojure.pprint/get-pretty-writer
([writer])
  Returns the java.io.Writer passed in wrapped in a pretty writer proxy, unless it's
  already a pretty writer. Generally, it is unnecessary to call this function, since pprint,
  write, and cl-format all call it if they need to. However if you want the state to be
  preserved across calls, you will want to wrap them with this.

For example, when you want to generate column-aware output with multiple calls to cl-format,
do it like in this example:

(defn print-table [aseq column-width]
  (binding [*out* (get-pretty-writer *out*)]
    (doseq [row aseq]
      (doseq [col row]
        (cl-format true "~4D~7,vT" col column-width))
      (format "~n"))))
```

Now if you want to check the source code for a function, just use the source function, for example type:

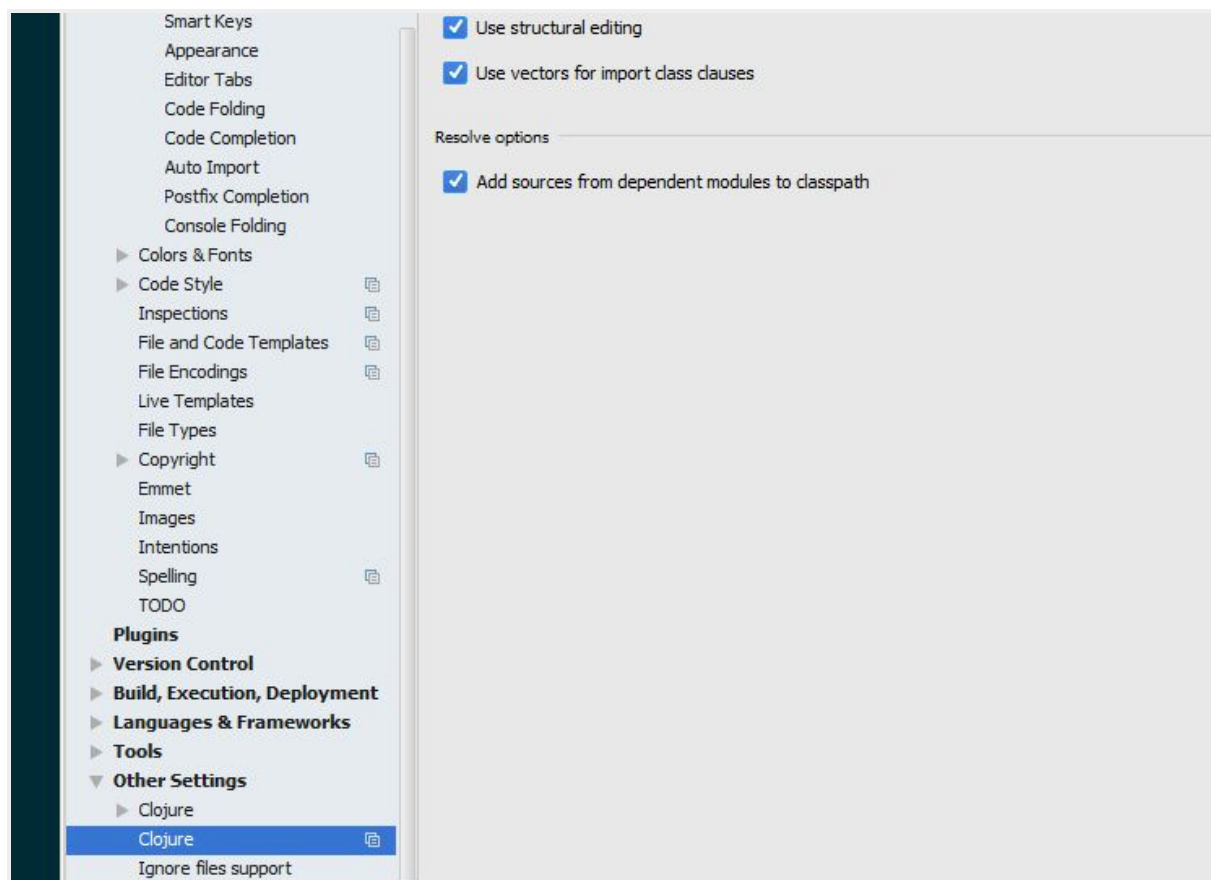
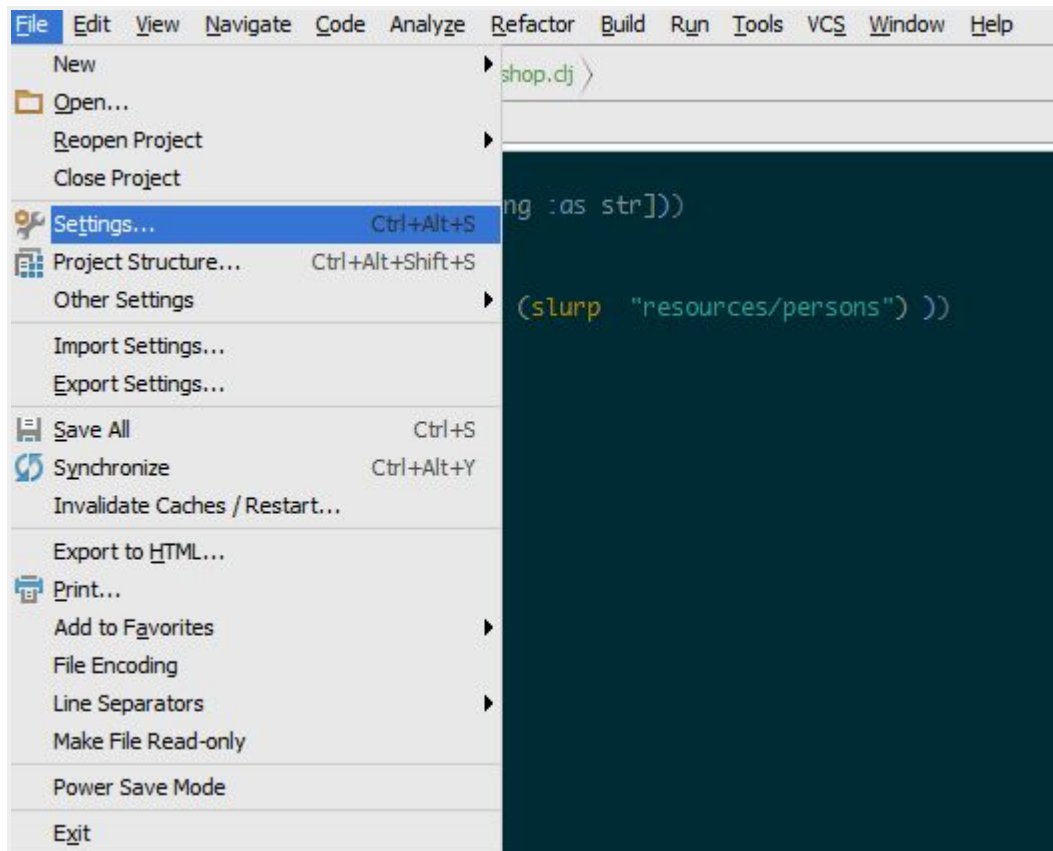
```
(source range)
```

```
(source range)
(defn range
  "Returns a lazy seq of nums from start (inclusive) to end
  (exclusive), by step, where start defaults to 0, step to 1, and end to
  infinity. When step is equal to 0, returns an infinite sequence of
  start. When start is equal to end, returns empty list."
  {:added "1.0"
   :static true}
  ([] (range 0 Double/POSITIVE_INFINITY 1))
  ([end] (range 0 end 1))
  ([start end] (range start end 1))
  ([start end step]
   (lazy-seq
    (let [b (chunk-buffer 32)
          comp (cond (or (zero? step) (= start end)) not=
                     (pos? step) <
                     (neg? step) >)]
      (loop [i start]
        (if (and (< (count b) 32)
```

## 5. Using cursive to edit your code

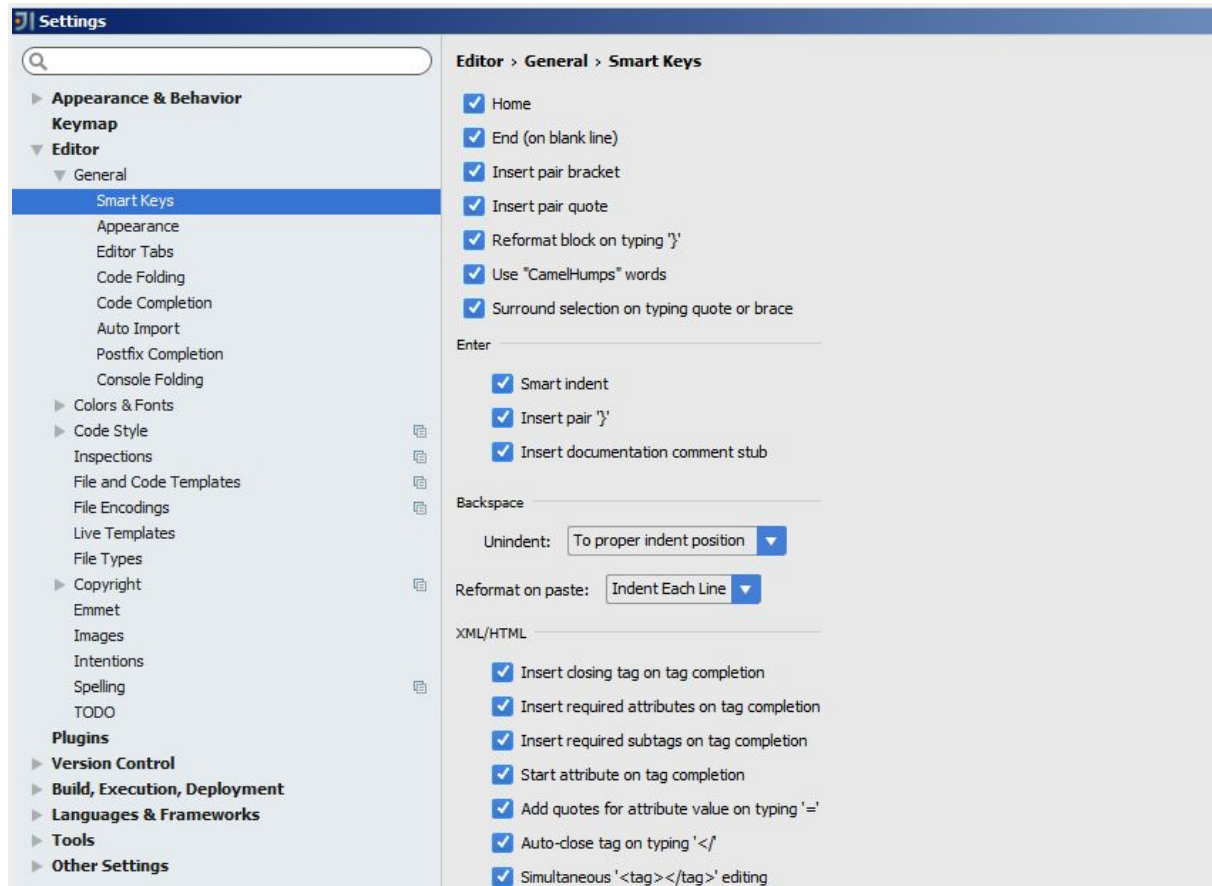
Cursive as well as emacs has structural editing also called paredit this will help us edit of S-expression data easily, for example cursive will keep your parenthesis, strings, quotes always balanced, select full s-expression to copy or delete among other functionalities, first turn on structural edition if it is not set. Go to File->Settings->Other Settings->Clojure and check the following boxes.



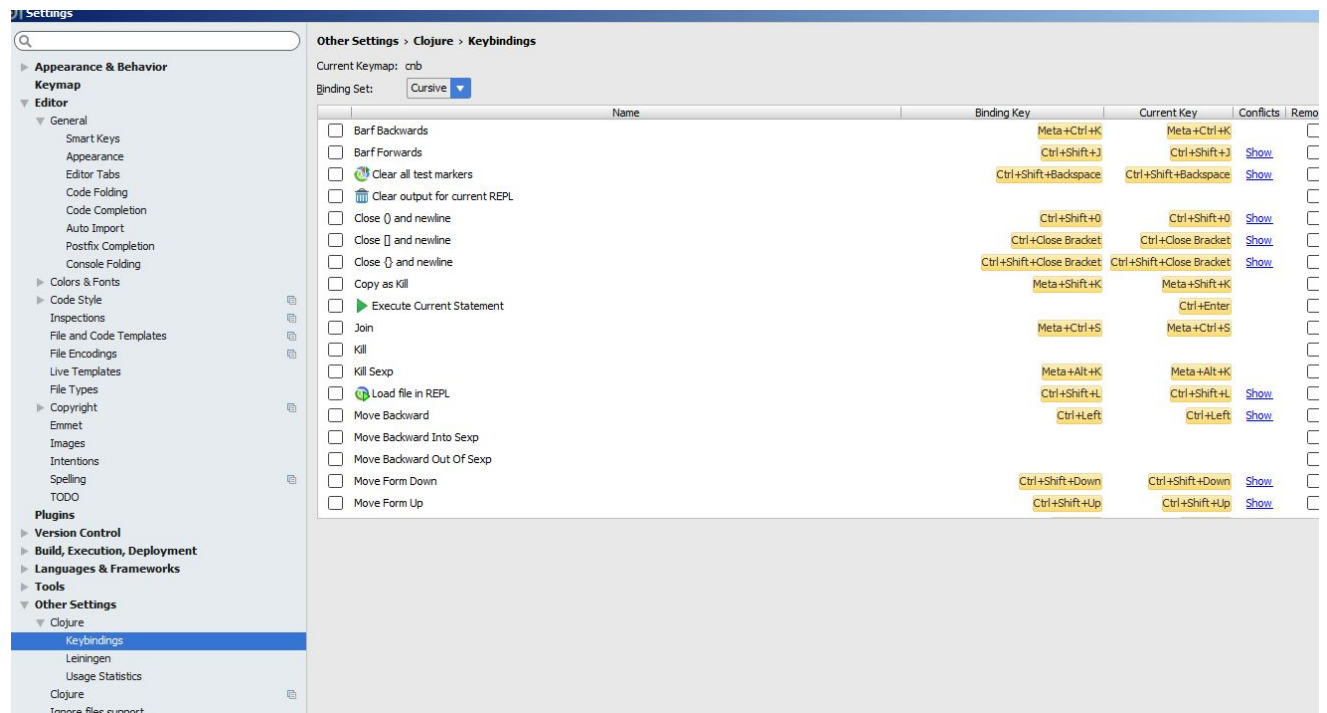




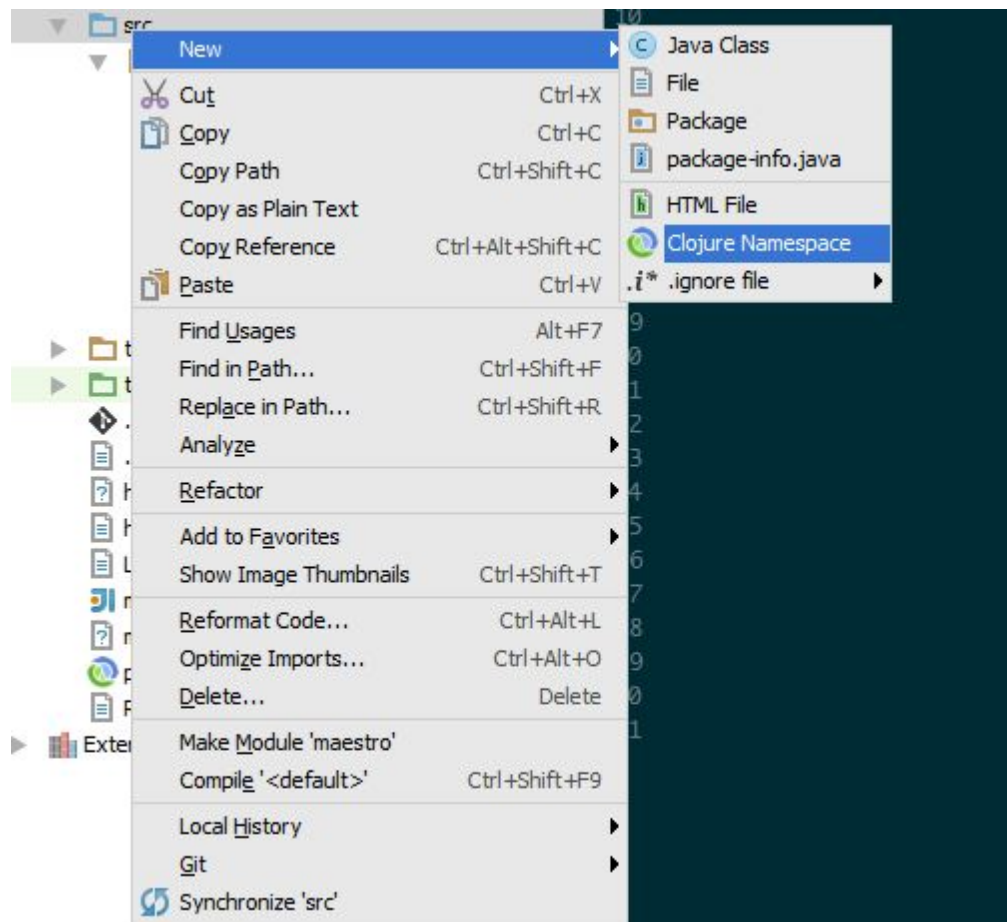
Then setup the following options for smartkeys, go to File->Settings->editor->general->smartkeys these options will save you time while typing.



Finally check if the default keybindings are comfortable for you. Go to File->Settings->Other Settings->Clojure->Keybindings.

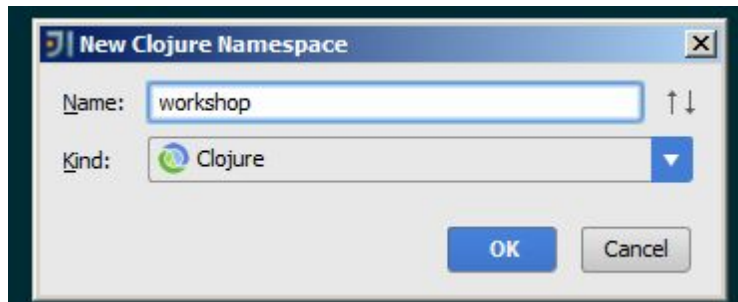


Now create a new file in the src folder of your project, this is done right clicking on the src folder and choosing New->Clojure Namespace.



Then just assign a name for this new namespace the name workshop will do.

What is a namespace ? At the moment just think of this as a group which elements are the symbols you create in this namespace and belong along with other namespaces to the clojure universe of symbol references, it is sort of like the java package concept.



*Now that the file is created, try typing the following :*

`"this is a string"`

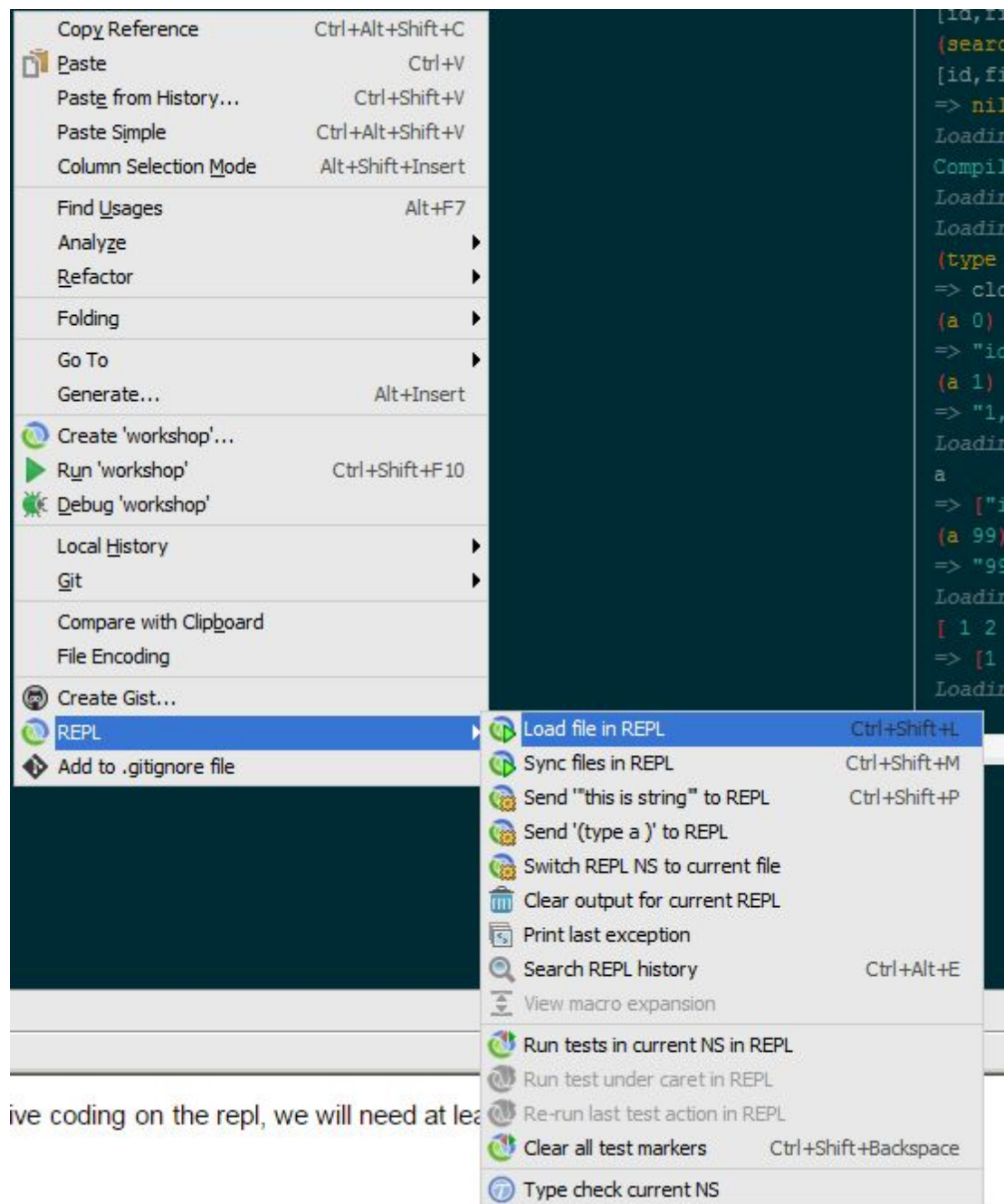
Select `"this is a string"` just position the cursor at the beginning of the string and type (it will surround the whole expression).

`( "this is a string")`

If you are the end of a s-expression you could move back and forward using CTRL+ ARROW KEYS.

## *6. The editor and the REPL:*

While you are coding you can load all your functions into the repl just clicking the option



ive coding on the repl, we will need at least

If you already had loaded this file, just choose Sync files in REPL. now all your symbol definitions should be available in the REPL to use them.

Now let's play with the REPL, type the following code in the new namespace you have created previously, this is the url for the project : <https://github.com/cneira/clojure-workshop1>.

```
(ns maestro.workshop
  (:require [clojure.string :as str]) ) 1
; split the input from slurp as newline as the token separator and store in symbol a all lines
as a vector
(def a (str/split-lines (slurp "resources/people.txt") )) 2
```

Let's explain what we just typed:

1. The require keyword (we will talk about keywords later) imports the clojure.string namespace into ours and give it the alias str to reference it. ns function assigns a namespace name to our current namespace.
2. Let's decompose this s-expression :  
(slurp "resources/persons") will return the contents of the file or stream as a string.

*clojure.core/slurp*  
*([f & opts])*  
*Opens a reader on f and reads all its contents, returning a string.*  
*See clojure.java.io/reader for a complete list of supported arguments.*

you could also slurp a web page :

```
(println (slurp "http://hmpg.net/")) )
```

println just prints the result of the slurp call. So after evaluating slurp the s-expression should be something like this.

```
(def a (str/split-lines <contents from the people.txt file as string> )
)
```

```
(str/split-lines <contents from the people.txt file as string> )
```

Will split all the strings using the newline as a token for separation and return the result as a vector, each line of the people.txt file will be an element of the vector.

finally the vector will be referenced by the symbol a as intended by the

```
(def a <vector of lines from the people.txt file> )
```

*What is a clojure vector ?*

It is a sequential data structure which gives practically  $O(1)$  runtime for appends, updates, lookups and subvec. As they are persistent, every modification creates a new vector instead of changing the old one.

As vector is a function you could evaluate a vector to its index, evaluate the following s-expression in your REPL

```
( a 1 )
```

This should result in the output below.

```
=> "1,Norma,Cooper,ncooper0@is.gd,Philippines,30.40.133.122"
```

If you just evaluate a without parameters you will see the full contents. you could also code the same line using the `->` macro, if that makes it clearer.

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

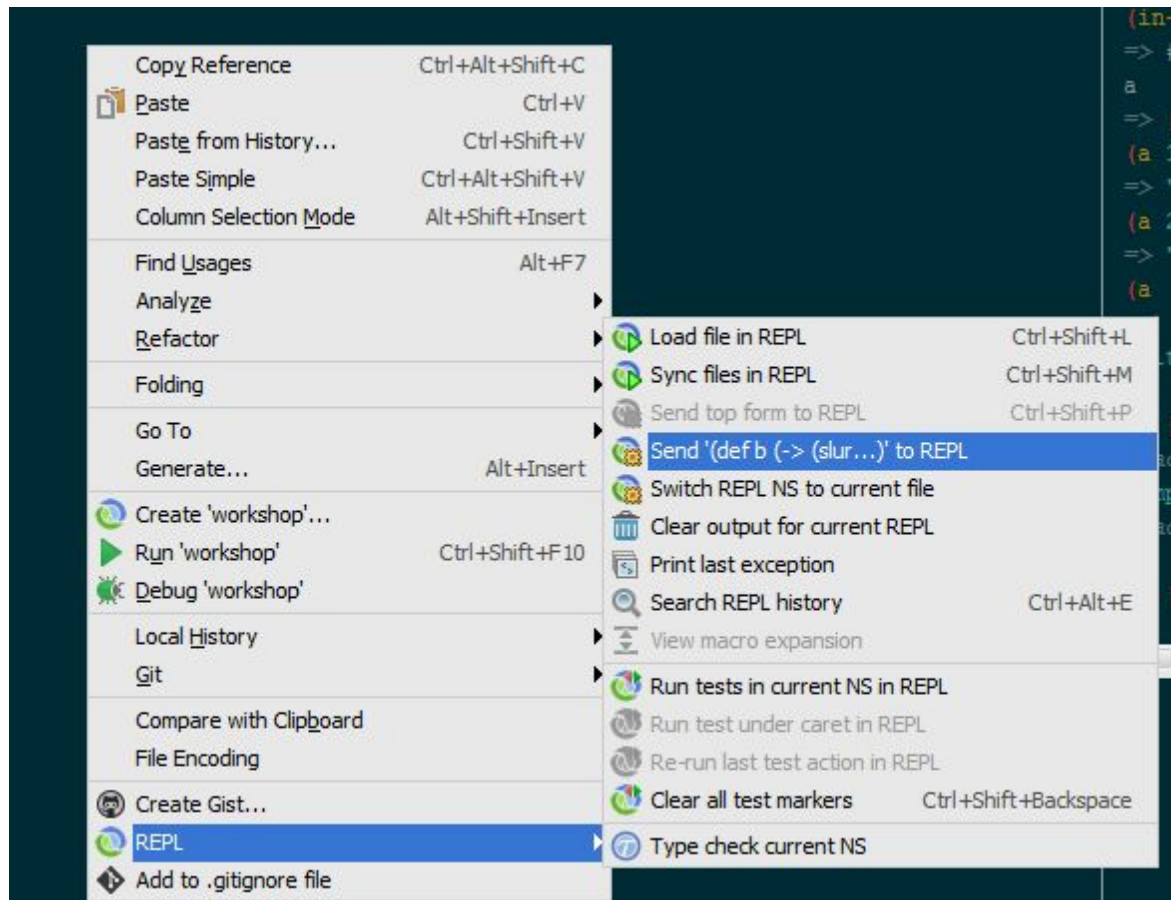
```
(-> (slurp "resources/people.txt") str/split-lines )
```

This means the result of slurp will be passed as parameter to `str/split-lines`, as this is a small s-expression using the `->` macro will be helpful when you need to code more complex ones.

Now let's create a function (for the moment we will see variable argument functions ) add this to your namespace (edit your workshop.clj file).

```
(defn add [a b ]  
  ( + a b ))
```

Sync your file with the repl (right click on your workshop.clj file) or you just can send this s-expression to be evaluated in the REPL clicking the Send '<s-expression>..' to REPL.



Now evaluate it in your REPL

```
(add 4 5 )
```

As you could see clojure has no types on the signature of the function, we could just do the following.

```
(add "potato" 3 )
```

That will result in an error, we could avoid that using preconditions when we define a function but we will talk about later on the workshop.

Let's create a more interesting function that operates on our persons vector, type in our workshop.clj file add the following function :

```
(defn return-names [file]      --> 1
  (for [x (rest file) ]        --> 2
    (let [ fields  (str/split x #",")] --> 3
      (fields 1))))            --> 4
```



Let's see what this code does:

1. This line just define a function which symbol is return-names and it takes only one parameter
2. This is the classic for loop, the syntax is the following:

According to the documentation :

```
clojure.core/for
([seq-exprs body-expr])
```

### Macro

List comprehension. Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a lazy sequence of evaluations of expr. Collections are iterated in a nested fashion, rightmost fastest, and nested coll-exprs can refer to bindings created in prior binding-forms. Supported modifiers are: `:let [binding-form expr ...]`, `:while test`, `:when test`.

For the moment I'll only explain this: "Takes a vector of one or more binding-form/collection-expr-pairs ", this is what we have done in this line :

```
(for [x (rest file)]
```

Binding form is this expression `[ x (rest file) ]` here you are associating `x` with the result of `(rest file)`, you are binding the symbol to a value.

```
3. (let [ fields (str/split x #",")]
```

Here we are creating a local symbol called `field` that is associated with the value of the `(str/split x #",")` function call. This symbol is only available inside the `let` scope, you could this a local variable.

```
(str/split x #",")
```

The `split` function separates a string using a regular expression as string separator, the clojure reader knows (remember the `read eval loop?` ) that a string prefixed with `#` is a regular expression, type this and check for yourself.

```
(type #",")
```

```
=> java.util.regex.Pattern
```

So you could use this as references when creating your regexp

<http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

The only difference would be that clojure does not need the escaping backslashes as in java.

Here as we have binded the symbol fields to the result of split call, the field symbol should have a vector composed of the fields from the first element of the vector composed by people and their personal information.

To make it clearer type:

```
(a 1)
=> "1,Teresa,Powell,tpowell0@dagondesign.com,China,168.220.52.12"
```

This is a string of the second element of vector a ( first element contain only headers).

```
(str/split (a 1) #",")
```

Now we split the previous string using a regular expression and this will return a vector with the following elements:

```
=> ["1" "Teresa" "Powell" "tpowell0@dagondesign.com" "China"
"168.220.52.12"]
```

4. Finally we evaluate this vector to its index and as we know the first name is located in the second element of the vector we return it.

Call this function in your repl

```
(return-names a )
```

## Assignments :

1. Modify the file and sync it with the repl, Why is helpful to use the sync option?
2. Create a function that only returns the person information which first name is "Teresa" (hint : check for when keyword and let optional keywords in for loop)

This material is property of [codemissions.com](https://codemissions.com)

Author: Carlos Antonio Neira Bustos

Email : [cneirabustos@gmail.com](mailto:cneirabustos@gmail.com)