Scheme and Clojure are in the lisp-1 family, which use the same namespace for functions and variables.
Common Lisp is in the lisp-2 family, which uses separate namespaces.

With the leiningen build automation tool.

```
lein new MyProj
```

```
(funcname arg1 arg2 ...)
```

- It uses the normal function prefix notation.
- `/` results in a `Ratio`.

Using `mod`.

- It's easier to support higher-arity versions of the function.
- There's no ambiguity and no need for consideration of operator precedence.

Using the equivalent of Java's `toString`.

```
(str obj1 obj2 ... objn)
```

Use `\`s:

```
user=> (str \h \i)
"hi"
```

Back slashes actually create characters, which are not strings in Clojure.

```
(class expr)
```

Using an `if` function.
Its first argument is a Boolean expression, the second argument is the code to run if the Boolean expression is `true`. The third argument is optional and is the `else` code.

Everything except `nil` and `false`.

`0` and `""` **do** evaluate to "true" in conditionals.

Lists are for code, while vectors are used to store data.

Vectors support fast random access.

With the `list` function or by quoting:

```
'("my" "favorite" "list")
```

Or with the `cons` function.

`first`, `rest`, `last`

```
user=> (nth ["a", "b", "c"] 2)
"c"
```

Use `concat`.

Vectors use `[]`, sets use `#{}`, maps use `{}`.

```
(def VarName expr)
```

Yes, reassignments are allowed.

Use `count`.

```
user=> (["a" "b" "c"] 1)
"b"
user=> (#{"a" "b" "c"} "a")
"a"
```

Commas as whitespace, to prevent mixing up keys/values, or putting an odd number of elements.

Keywords (beginning with `:`) stand for themselves, like atoms in other languages. Symbols refer to something else.

```
(:key map)
```

or

```
(map :key)
```

Both the map and the keyword are functions.

`merge` and `merge-with` combine maps.

`assoc map newElement` adds to a map.

```
(defn funcName [params] body)
```

Add a docstring after the function name and before the parameter vector. To retrieve use:

```
(doc funcName)
```

It can be used in an argument list or in a `let` statement.

Unlike pattern matching, destructuring does not require you match the entire data structure being destructured.

```
(let [bindings* ] exprs*)
```

The expressions are evaluated in the lexical context of the bindings, sequentially applied.
The even numbered elements in `bindings` are bound to the odd symbols.
Unlike in a `def`, those bindings are not active outside the `let`.

Because functions are really just lists.

Use `fn` instead of `defn` and don't put a name.

Or, put a `#` before the list containing the body of the function with `%` being bound to each argument.

```
(apply func arglist)
```

Using `loop`/`recur`.

The `loop` function takes a vector whose odd number elements are variables and even number elements are initial variables to those arguments. Its second argument is the body of the function which can use `recur` to go back to `loop`.

```
(every? func list)
(some func list)
```

`not-every` and `not-any` are the opposite.

```
(nil? list)
```

```
(for [el1 list1 ... el2 list2 :when filter1] body)
```

`x?`

`word1-word2` type function names.

- Erlang - `foldl`
- Ruby - `inject`
- Clojure - `reduce`
- Haskell - `foldl`

```
user=> (range -5 5 2)
(-5 -3 -1 1 3)
```

```
(repeat "to infinity and beyond!")

(repeat (cycle ["one" "more" "time"]))

(iterate inc 3)
```

Calls to `take` or `drop`, to extract sub-sequences.

Using the left-to-right operator $->>$. It applies each function to the result of the *previos* one.