FREE eBook

LEARNING Elixir Language

Free unaffiliated eBook created from **Stack Overflow contributors.**

#elixir

Table of Contents

About ... 1

Chapter 1: Getting started with Elixir Language ... 2

Remarks ... 2

Versions ... 2

Examples ... 2

Hello World ... 2

Hello World from IEx ... 3

Chapter 2: Basic .gitignore for elixir program ... 5

Chapter 3: Basic .gitignore for elixir program ... 6

Remarks ... 6

Examples ... 6

A basic .gitignore for Elixir ... 6

Example ... 6

Standalone elixir application ... 6

Phoenix application ... 7

Auto-generated .gitignore ... 7

Chapter 4: basic use of guard clauses ... 8

Examples ... 8

basic uses of guard clauses ... 8

Chapter 5: BEAM ... 10

Examples ... 10

Introduction ... 10

Chapter 6: Behaviours ... 11

Examples ... 11

Introduction ... 11

Chapter 7: Better debugging with IO.inspect and labels ... 12

Introduction ... 12

Remarks ... 12

Examples ... 12

Without labels ... 12

With labels 13
Chapter 8: Built-in types 14
Examples 14
Numbers 14
Atoms 15
Binaries and Bitstrings 15
Chapter 9: Conditionals 17
Remarks 17
Examples 17
case 17
if and unless 17
cond 18
with clause 18
Chapter 10: Constants 20
Remarks 20
Examples 20
Module-scoped constants 20
Constants as functions 20
Constants via macros 21
Chapter 11: Data Structures 23
Syntax 23
Remarks 23
Examples 23
Lists 23
Tuples 23
Chapter 12: Debugging Tips 24
Examples 24
Debugging with IEX.pry/0 24
Debugging with IO.inspect/1 24
Debug in pipe 25
Pry in pipe 25
Chapter 13: Doctests 27

Examples ... 27
Introduction ... 27
Generating HTML documentation based on doctest ... 27
Multiline doctests ... 27
Chapter 14: Ecto ... 29
Examples ... 29
Adding a Ecto.Repo in an elixir program ... 29
"and" clause in a Repo.get_by/3 ... 29
Querying with dynamic fields ... 30
Add custom data types to migration and to schema ... 30
Chapter 15: Erlang ... 31
Examples ... 31
Using Erlang ... 31
Inspect an Erlang module ... 31
Chapter 16: ExDoc ... 32
Examples ... 32
Introduction ... 32
Chapter 17: ExUnit ... 33
Examples ... 33
Asserting Exceptions ... 33
Chapter 18: Functional programming in Elixir ... 34
Introduction ... 34
Examples ... 34
Map ... 34
Reduce ... 34
Chapter 19: Functions ... 36
Examples ... 36
Anonymous Functions ... 36
Using the capture operator ... 36
Multiple bodies ... 37
Keyword lists as function parameters ... 37

Named Functions & Private Functions ... 37
Pattern Matching ... 38
Guard clauses ... 38
Default Parameters ... 39
Capture functions ... 39
Chapter 20: Getting help in IEx console ... 41
Introduction ... 41
Examples ... 41
Listing Elixir modules and functions ... 41
Chapter 21: IEx Console Tips & Tricks ... 42
Examples ... 42
Recompile project with `recompile` ... 42
See documentation with `h` ... 42
Get value from last command with `v` ... 42
Get the value of a previous command with `v` ... 42
Exit IEx console ... 43
See information with `i` ... 43
Creating PID ... 44
Have your aliases ready when you start IEx ... 44
Persistent history ... 44
When Elixir console is stuck... ... 44
break out of incomplete expression ... 45
Load a module or script into the IEx session ... 46
Chapter 22: Installation ... 47
Examples ... 47
Fedora Installation ... 47
OSX Installation ... 47
Homebrew ... 47
Macports ... 47
Debian/Ubuntu Installation ... 47
Gentoo/Funtoo Installation ... 47
Chapter 23: Join Strings ... 49

Examples ... 49
Using String Interpolation ... 49
Using IO List ... 49
Using Enum.join ... 49
Chapter 24: Lists ... 50
Syntax ... 50
Examples ... 50
Keyword Lists ... 50
Char Lists ... 51
Cons Cells ... 52
Mapping Lists ... 52
List Comprehensions ... 53
Combined example ... 53
Summary ... 53
List difference ... 54
List Membership ... 54
Converting Lists to a Map ... 54
Chapter 25: Maps and Keyword Lists ... 55
Syntax ... 55
Remarks ... 55
Examples ... 55
Creating a Map ... 55
Creating a Keyword List ... 55
Difference between Maps and Keyword Lists ... 56
Chapter 26: Metaprogramming ... 57
Examples ... 57
Generate tests at compile time ... 57
Chapter 27: Mix ... 58
Examples ... 58
Create a Custom Mix Task ... 58
Custom mix task with command line arguments ... 58
Aliases ... 58

Get help on available mix tasks ... 59
Chapter 28: Modules ... 60
Remarks ... 60
Module Names ... 60
Examples ... 60
List a module's functions or macros ... 60
Using modules ... 60
Delegating functions to another module ... 61
Chapter 29: Nodes ... 62
Examples ... 62
List all visible nodes in the system ... 62
Connecting nodes on the same machine ... 62
Connecting nodes on different machines ... 62
Chapter 30: Operators ... 64
Examples ... 64
The Pipe Operator ... 64
Pipe operator and parentheses ... 64
Boolean operators ... 65
Comparison operators ... 66
Join operators ... 66
'In' operator ... 67
Chapter 31: Optimization ... 68
Examples ... 68
Always measure first! ... 68
Chapter 32: Pattern matching ... 69
Examples ... 69
Pattern matching functions ... 69
Pattern matching on a map ... 69
Pattern matching on a list ... 69
Get the sum of a list using pattern matching ... 70
Anonymous functions ... 70
Tuples ... 71

Reading a File ... 71
Pattern matching anonymous functions ... 71
Chapter 33: Polymorphism in Elixir ... 73
Introduction ... 73
Remarks ... 73
Examples ... 73
Polymorphism with Protocols ... 73
Chapter 34: Processes ... 75
Examples ... 75
Spawning a Simple Process ... 75
Sending and Receiving Messages ... 75
Recursion and Receive ... 75
Chapter 35: Protocols ... 77
Remarks ... 77
Examples ... 77
Introduction ... 77
Chapter 36: Sigils ... 78
Examples ... 78
Build a list of strings ... 78
Build a list of atoms ... 78
Custom sigils ... 78
Chapter 37: State Handling in Elixir ... 79
Examples ... 79
Managing a piece of state with an Agent ... 79
Chapter 38: Stream ... 80
Remarks ... 80
Examples ... 80
Chaining multiple operations ... 80
Chapter 39: Strings ... 81
Remarks ... 81
Examples ... 81

Convert to string ... 81
Get a substring ... 81
Split a string ... 81
String Interpolation ... 81
Check if String contains Substring ... 81
Join Strings ... 82
Chapter 40: Task ... 83
Syntax ... 83
Parameters ... 83
Examples ... 83
Doing work in the background ... 83
Parallel processing ... 83
Chapter 41: Tips and Tricks ... 84
Introduction ... 84
Examples ... 84
Creating Custom Sigils and Documenting ... 84
Multiple [OR] ... 84
iex Custom Configuration - iex Decoration ... 84
Credits ... 86

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: elixir-language

It is an unofficial and free Elixir Language 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 Elixir Language.

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 Elixir Language

Remarks

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

Versions

Version	Release Date
0.9	2013-05-23
1.0	2014-09-18
1.1	2015-09-28
1.2	2016-01-03
1.3	2016-06-21
1.4	2017-01-05

Examples

Hello World

For installation instructions on elixir check here, it describes instructions related to different platforms.

Elixir is a programming language that is created using `erlang`, and uses erlang's `BEAM` runtime (like `JVM` for java).

We can use elixir in two modes: interactive shell `iex` or directly running using `elixir` command.

Place the following in a file named `hello.exs`:

```
IO.puts "Hello world!"
```

From the command line, type the following command to execute the Elixir source file:

```
$ elixir hello.exs
```

This should output:

    Hello world!

This is known as the *scripted mode* of `Elixir`. In fact, Elixir programs can also be compiled (and generally, they are) into bytecode for the BEAM virtual machine.

You can also use `iex` for interactive elixir shell (recommended), run the command you will get a prompt like this:

```
Interactive Elixir (1.3.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Here you can try your elixir `hello world` examples:

```
iex(1)> IO.puts "hello, world"
hello, world
:ok
iex(2)>
```

You can also compile and run your modules through `iex`. For example, if you have a `helloworld.ex` that contains:

```
defmodule Hello do
   def sample do
       IO.puts "Hello World!"
   end
end
```

Through `iex`, do:

```
iex(1)> c("helloworld.ex")
[Hello]
iex(2)> Hello.sample
Hello World!
```

Hello World from IEx

You can also use the `IEx` (Interactive Elixir) shell to evaluate expressions and execute code.

If you are on Linux or Mac, just type iex on your bash and press enter:

```
$ iex
```

If you are on a Windows machine, type:

```
C:\ iex.bat
```

Then you will enter into the IEx REPL (Read, Evaluate, Print, Loop), and you can just type something like:

```
iex(1)> "Hello World"
"Hello World"
```

If you want to load a script while opening an IEx REPL, you can do this:

```
$ iex script.exs
```

Given `script.exs` is your script. You can now call functions from the script in the console.

Read Getting started with Elixir Language online: https://riptutorial.com/elixir/topic/954/getting-started-with-elixir-language

Chapter 2: Basic .gitignore for elixir program

Read Basic .gitignore for elixir program online: https://riptutorial.com/elixir/topic/6493/basic--gitignore-for-elixir-program

Chapter 3: Basic .gitignore for elixir program

Remarks

Note that the `/rel` folder may not be needed in your .gitignore file. This is generated if you are using a release management tool such as `exrm`

Examples

A basic .gitignore for Elixir

```
/_build
/cover
/deps
erl_crash.dump
*.ez

# Common additions for various operating systems:
# MacOS
.DS_Store

# Common additions for various editors:
# JetBrains IDEA, IntelliJ, PyCharm, RubyMine etc.
.idea
```

Example

```
### Elixir ###
/_build
/cover
/deps
erl_crash.dump
*.ez

### Erlang ###
.eunit
deps
*.beam
*.plt
ebin
rel/example_project
.concrete/DEV_MODE
.rebar
```

Standalone elixir application

```
/_build
/cover
/deps
erl_crash.dump
*.ez
```

```
/rel
```

Phoenix application

```
/_build
/db
/deps
/*.ez
erl_crash.dump
/node_modules
/priv/static/
/config/prod.secret.exs
/rel
```

Auto-generated .gitignore

By default, `mix new <projectname>` will generate a `.gitignore` file in the project root that is suitable for Elixir.

```
# The directory Mix will write compiled artifacts to.
/_build

# If you run "mix test --cover", coverage assets end up here.
/cover

# The directory Mix downloads your dependencies sources to.
/deps

# Where 3rd-party dependencies like ExDoc output generated docs.
/doc

# If the VM crashes, it generates a dump, let's ignore it too.
erl_crash.dump

# Also ignore archive artifacts (built via "mix archive.build").
*.ez
```

Read Basic .gitignore for elixir program online: https://riptutorial.com/elixir/topic/6526/basic--gitignore-for-elixir-program

Chapter 4: basic use of guard clauses

Examples

basic uses of guard clauses

In Elixir, one can create multiple implementations of a function with the same name, and specify rules which will be applied to the parameters of the function *before calling the function* in order to determine which implementation to run.

These rules are marked by the keyword `when`, and they go between the `def function_name(params)` and the `do` in the function definition. A trivial example:

```
defmodule Math do

  def is_even(num) when num === 1 do
    false
  end
  def is_even(num) when num === 2 do
    true
  end

  def is_odd(num) when num === 1 do
    true
  end
  def is_odd(num) when num === 2 do
    false
  end

end
```

Say I run `Math.is_even(2)` with this example. There are two implementations of `is_even`, with differing guard clauses. The system will look at them in order, and run the first implementation where the parameters satisfy the guard clause. The first one specifies that `num === 1` which is not true, so it moves on to the next one. The second one specifies that `num === 2`, which is true, so this is the implementation that is used, and the return value will be `true`.

What if I run `Math.is_odd(1)`? The system looks at the first implementation, and sees that since `num` is `1` the guard clause of the first implementation is satisfied. It will then use that implementation and return `true`, and not bother looking at any other implementations.

Guards are limited in the types of operations they can run. The Elixir documentation lists every allowed operation; in a nutshell they allow comparisons, math, binary operations, type-checking (e.g. `is_atom`), and a handful of small convenience functions (e.g. `length`). It is possible to define custom guard clauses, but it requires creating macros and is best left for a more advanced guide.

Note that guards do not throw errors; they are treated as normal failures of the guard clause, and the system moves on to look at the next implementation. If you find that you're getting `(FunctionClauseError) no function clause matching` when calling a guarded function with params

you expect to work, it may be that a guard clause which you expect to work is throwing an error which is being swallowed up.

To see this for yourself, create and then call a function with a guard which makes no sense, such as this which tries to divide by zero:

```
defmodule BadMath do
  def divide(a) when a / 0 === :foo do
    :bar
  end
end
```

Calling `BadMath.divide("anything")` will provide the somewhat-unhelpful error `(FunctionClauseError) no function clause matching in BadMath.divide/1` — whereas if you had tried to run `"anything" / 0` directly, you would get a more helpful error: `(ArithmeticError) bad argument in arithmetic expression`.

Read basic use of guard clauses online: https://riptutorial.com/elixir/topic/6121/basic-use-of-guard-clauses

Chapter 5: BEAM

Examples

Introduction

```
iex> :observer.start
:ok
```

`:observer.start` opens the GUI observer interface, showing you CPU breakdown, memory usage, and other information critical to understanding the usage patterns of your applications.

Read BEAM online: https://riptutorial.com/elixir/topic/3587/beam

Chapter 6: Behaviours

Examples

Introduction

Behaviours are a list of functions specifications that another module can implement. They are similar to interfaces in other languages.

Here's an example behaviour:

```
defmodule Parser do
  @callback parse(String.t) :: any
  @callback extensions() :: [String.t]
end
```

And a module that implements it:

```
defmodule JSONParser do
  @behaviour Parser

  def parse(str), do: # ... parse JSON
  def extensions, do: ["json"]
end
```

The `@behaviour` module attribute above indicates that this module is expected to define every function defined in the Parser module. Missing functions will result in undefined behaviour function compilation errors.

Modules can have multiple `@behaviour` attributes.

Read Behaviours online: https://riptutorial.com/elixir/topic/3558/behaviours

Chapter 7: Better debugging with IO.inspect and labels

Introduction

`IO.inspect` is very useful when you try to debug your chains of method calling. It can get messy though if you use it too often.

Since Elixir 1.4.0 the `label` option of `IO.inspect` can help

Remarks

Only works with Elixir 1.4+, but I can't tag that yet.

Examples

Without labels

```
url
  |> IO.inspect
  |> HTTPoison.get!
  |> IO.inspect
  |> Map.get(:body)
  |> IO.inspect
  |> Poison.decode!
  |> IO.inspect
```

This will result in a lot of output with no context:

```
"https://jsonplaceholder.typicode.com/posts/1"
%HTTPoison.Response{body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere
repellat provident occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et
suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut
quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}",
 headers: [{"Date", "Thu, 05 Jan 2017 14:29:59 GMT"},
  {"Content-Type", "application/json; charset=utf-8"},
  {"Content-Length", "292"}, {"Connection", "keep-alive"},
  {"Set-Cookie",
   "__cfduid=d56d1be0a544fcbdbb262fee9477600c51483626599; expires=Fri, 05-Jan-18 14:29:59 GMT;
path=/; domain=.typicode.com; HttpOnly"},
  {"X-Powered-By", "Express"}, {"Vary", "Origin, Accept-Encoding"},
  {"Access-Control-Allow-Credentials", "true"},
  {"Cache-Control", "public, max-age=14400"}, {"Pragma", "no-cache"},
  {"Expires", "Thu, 05 Jan 2017 18:29:59 GMT"},
  {"X-Content-Type-Options", "nosniff"},
  {"Etag", "W/\"124-yv65LoT2uMHrpn06wNpAcQ\""}, {"Via", "1.1 vegur"},
  {"CF-Cache-Status", "HIT"}, {"Server", "cloudflare-nginx"},
  {"CF-RAY", "31c7a025e94e2d41-TXL"}], status_code: 200}
"{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere repellat provident
```

```
occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et suscipit\\nsuscipit
recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum
rerum est autem sunt rem eveniet architecto\"\n}"
%{"body" => "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit
molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto",
  "id" => 1,
  "title" => "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "userId" => 1}
```

With labels

using the `label` option to add context can help a lot:

```
url
  |> IO.inspect(label: "url")
  |> HTTPoison.get!
  |> IO.inspect(label: "raw http resonse")
  |> Map.get(:body)
  |> IO.inspect(label: "raw body")
  |> Poison.decode!
  |> IO.inspect(label: "parsed body")

url: "https://jsonplaceholder.typicode.com/posts/1"
raw http resonse: %HTTPoison.Response{body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\":
\"sunt aut facere repellat provident occaecati excepturi optio reprehenderit\",\n  \"body\":
\"quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit
molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}",
 headers: [{"Date", "Thu, 05 Jan 2017 14:33:06 GMT"},
  {"Content-Type", "application/json; charset=utf-8"},
  {"Content-Length", "292"}, {"Connection", "keep-alive"},
  {"Set-Cookie",
   "__cfduid=d22d817e48828169296605d27270af7e81483626786; expires=Fri, 05-Jan-18 14:33:06 GMT;
path=/; domain=.typicode.com; HttpOnly"},
  {"X-Powered-By", "Express"}, {"Vary", "Origin, Accept-Encoding"},
  {"Access-Control-Allow-Credentials", "true"},
  {"Cache-Control", "public, max-age=14400"}, {"Pragma", "no-cache"},
  {"Expires", "Thu, 05 Jan 2017 18:33:06 GMT"},
  {"X-Content-Type-Options", "nosniff"},
  {"Etag", "W/\"124-yv65LoT2uMHrpn06wNpAcQ\""}, {"Via", "1.1 vegur"},
  {"CF-Cache-Status", "HIT"}, {"Server", "cloudflare-nginx"},
  {"CF-RAY", "31c7a4b8ae042d77-TXL"}], status_code: 200}
raw body: "{\n  \"userId\": 1,\n  \"id\": 1,\n  \"title\": \"sunt aut facere repellat
provident occaecati excepturi optio reprehenderit\",\n  \"body\": \"quia et
suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut
quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n}"
parsed body: %{"body" => "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet
architecto",
  "id" => 1,
  "title" => "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "userId" => 1}
```

Read Better debugging with IO.inspect and labels online:
https://riptutorial.com/elixir/topic/8725/better-debugging-with-io-inspect-and-labels

Chapter 8: Built-in types

Examples

Numbers

Elixir comes with **integers** and **floating point numbers**. An **integer literal** can be written in decimal, binary, octal and hexadecimal formats.

```
iex> x = 291
291

iex> x = 0b100100011
291

iex> x = 0o443
291

iex> x = 0x123
291
```

As Elixir uses bignum arithmetic, **the range of integer is only limited by the available memory on the system**.

Floating point numbers are double precision and follows IEEE-754 specification.

```
iex> x = 6.8
6.8

iex> x = 1.23e-11
1.23e-11
```

Note that Elixir also supports exponent form for floats.

```
iex> 1 + 1
2

iex> 1.0 + 1.0
2.0
```

First we added two integers numbers, and the result is an integer. Later we added two floating point numbers, and the result is a floating point number.

Dividing in Elixir always returns a floating point number:

```
iex> 10 / 2
5.0
```

In the same way, if you add, subtract or multiply an integer by a floating point number the result will be floating point:

```
iex> 40.0 + 2
42.0

iex> 10 - 5.0
5.0

iex> 3 * 3.0
9.0
```

For integer division, one can use the `div/2` function:

```
iex> div(10, 2)
5
```

Atoms

Atoms are constants that represent a name of some thing. The value of an atom is it's name. An atom name starts with a colon.

```
:atom   # that's how we define an atom
```

An atom's name is unique. Two atoms with the same names always are equal.

```
iex(1)> a = :atom
:atom

iex(2)> b = :atom
:atom

iex(3)> a == b
true

iex(4)> a === b
true
```

Booleans `true` and `false`, actually are atoms.

```
iex(1)> true == :true
true

iex(2)> true === :true
true
```

Atoms are stored in special atoms table. It's very important to know that this table is not garbage-collected. So, if you want (or accidentally it is a fact) constantly create atoms - it is a bad idea.

Binaries and Bitstrings

Binaries in elixir are created using the Kernel.SpecialForms construct <<>>.

They are a powerful tool which makes Elixir very useful for working with binary protocols and encodings.

Binaries and bitstrings are specified using a comma delimited list of integers or variable values, bookended by "<<" and ">>". They are composed of 'units', either a grouping of bits or a grouping of bytes. The default grouping is a single byte (8 bits), specified using an integer:

```
<<222,173,190, 239>> # 0xDEADBEEF
```

Elixir strings also convert directly to binaries:

```
iex> <<0, "foo">>
<<0, 102, 111, 111>>
```

You can add "specifiers" to each "segment" of a binary, allowing you to encode:

- Data Type
- Size
- Endianness

These specifiers are encoded by following each value or variable with the "::" operator:

```
<<102::integer-native>>
<<102::native-integer>> # Same as above
<<102::unsigned-big-integer>>
<<102::unsigned-big-integer-size(8)>>
<<102::unsigned-big-integer-8>> # Same as above
<<102::8-integer-big-unsigned>>
<<-102::signed-little-float-64>> # -102 as a little-endian Float64
<<-102::native-little-float-64>> # -102 as a Float64 for the current machine
```

The available data types you can use are:

- integer
- float
- bits (alias for bitstring)
- bitstring
- binary
- bytes (alias for binary)
- utf8
- utf16
- utf32

Be aware that when specifying the 'size' of the binary segment, it varies according to the 'type' chosen in the segment specifier:

- integer (default) 1 bit
- float 1 bit
- binary 8 bits

Read Built-in types online: https://riptutorial.com/elixir/topic/1774/built-in-types

Chapter 9: Conditionals

Remarks

Note that the `do...end` syntax is syntactic sugar for regular keyword lists, so you can actually do this:

```
unless false, do: IO.puts("Condition is false")
# Outputs "Condition is false"

# With an `else`:
if false, do: IO.puts("Condition is true"), else: IO.puts("Condition is false")
# Outputs "Condition is false"
```

Examples

case

```
 case {1, 2} do
  {3, 4} ->
    "This clause won't match."
  {1, x} ->
    "This clause will match and bind x to 2 in this clause."
  _ ->
    "This clause would match any value."
end
```

`case` is only used to match the given pattern of the particular data. Here , `{1,2}` is matching with different case pattern that is given in the code example.

if and unless

```
if true do
    "Will be seen since condition is true."
end

if false do
    "Won't be seen since condition is false."
else
    "Will be seen.
end

unless false do
    "Will be seen."
end

unless true do
    "Won't be seen."
else
    "Will be seen."
end
```

cond

```
cond do
   0 == 1 -> IO.puts "0 = 1"
   2 == 1 + 1 -> IO.puts "1 + 1 = 2"
   3 == 1 + 2 -> IO.puts "1 + 2 = 3"
end

# Outputs "1 + 1 = 2" (first condition evaluating to true)
```

`cond` will raise a `CondClauseError` if no conditions are true.

```
cond do
  1 == 2 -> "Hmmm"
  "foo" == "bar" -> "What?"
end
# Error
```

This can be avoided by adding a condition that will always be true.

```
cond do
  ... other conditions
  true -> "Default value"
end
```

Unless it is never expected to reach the default case, and the program should in fact crash at that point.

with clause

`with` clause is used to combine matching clauses. It looks like we combine anonymous functions or handle function with multiple bodies (matching clauses). Consider the case: we create a user, insert it into DB, then create greet email and then send it to the user.

Without the `with` clause we might write something like this (I omitted functions implementations):

```
case create_user(user_params) do
  {:ok, user} ->
    case Mailer.compose_email(user) do
      {:ok, email} ->
        Mailer.send_email(email)
      {:error, reason} ->
        handle_error
    end
  {:error, changeset} ->
    handle_error
end
```

Here we handle our business process's flow with `case` (it could be `cond` or `if`). That leads us to so-called 'pyramid of doom', because we have to deal with possible conditions and decide: whether move further or not. It would be much nicer to rewrite this code with `with` statement:

```
with {:ok, user} <- create_user(user_params),
     {:ok, email} <- Mailer.compose_email(user) do
  {:ok, Mailer.send_email}
else
  {:error, _reason} ->
    handle_error
end
```

In the code snippet above we've rewrite nested `case` clauses with `with`. Within `with` we invoke some functions (either anonymous or named) and pattern match on their outputs. If all matched, `with` return `do` block result, or `else` block result otherwise.

We can omit `else` so `with` will return either `do` block result or the first fail result.

So, the value of `with` statement is its `do` block result.

Read Conditionals online: https://riptutorial.com/elixir/topic/2118/conditionals

Chapter 10: Constants

Remarks

So this is a summary analysis I've done based on the methods listed at How do you define constants in Elixir modules?. I'm posting it for a couple reasons:

- Most Elixir documentation is quite thorough, but I found this key architectural decision lacking guidance - so I would have requested it as a topic.
- I wanted to get a little visibility and comments from others about the topic.
- I also wanted to test out the new SO Documentation workflow. ;)

I've also uploaded the entire code to the GitHub repo elixir-constants-concept.

Examples

Module-scoped constants

```
defmodule MyModule do
  @my_favorite_number 13
  @use_snake_case "This is a string (use double-quotes)"
end
```

These are only accessible from within this module.

Constants as functions

Declare:

```
defmodule MyApp.ViaFunctions.Constants do
  def app_version, do: "0.0.1"
  def app_author, do: "Felix Orr"
  def app_info, do: [app_version, app_author]
  def bar, do: "barrific constant in function"
end
```

Consume with require:

```
defmodule MyApp.ViaFunctions.ConsumeWithRequire do
  require MyApp.ViaFunctions.Constants

  def foo() do
    IO.puts MyApp.ViaFunctions.Constants.app_version
    IO.puts MyApp.ViaFunctions.Constants.app_author
    IO.puts inspect MyApp.ViaFunctions.Constants.app_info
  end

  # This generates a compiler error, cannot invoke `bar/0` inside a guard.
  # def foo(_bar) when is_bitstring(bar) do
```

```
  #   IO.puts "We just used bar in a guard: #{bar}"
  # end
end
```

Consume with import:

```
defmodule MyApp.ViaFunctions.ConsumeWithImport do
  import MyApp.ViaFunctions.Constants

  def foo() do
    IO.puts app_version
    IO.puts app_author
    IO.puts inspect app_info
  end
end
```

This method allows for reuse of constants across projects, but they will not be usable within guard functions that require compile-time constants.

Constants via macros

Declare:

```
defmodule MyApp.ViaMacros.Constants do
  @moduledoc """
  Apply with `use MyApp.ViaMacros.Constants, :app` or `import MyApp.ViaMacros.Constants,
:app`.

  Each constant is private to avoid ambiguity when importing multiple modules
  that each have their own copies of these constants.
  """

  def app do
    quote do
      # This method allows sharing module constants which can be used in guards.
      @bar "barrific module constant"
      defp app_version, do: "0.0.1"
      defp app_author, do: "Felix Orr"
      defp app_info, do: [app_version, app_author]
    end
  end

  defmacro __using__(which) when is_atom(which) do
    apply(__MODULE__, which, [])
  end
end
```

Consume with `use`:

```
defmodule MyApp.ViaMacros.ConsumeWithUse do
  use MyApp.ViaMacros.Constants, :app

  def foo() do
    IO.puts app_version
    IO.puts app_author
    IO.puts inspect app_info
```

```
  end

  def foo(_bar) when is_bitstring(@bar) do
    IO.puts "We just used bar in a guard: #{@bar}"
  end
end
```

This method allows you to use the `@some_constant` inside guards. I'm not even sure that the functions would be strictly necessary.

Read Constants online: https://riptutorial.com/elixir/topic/6614/constants

Chapter 11: Data Structures

Syntax

- [head | tail] = [1, 2, 3, true] # one can use pattern matching to break up cons cells. This assigns head to 1 and tail to [2, 3, true]
- %{d: val} = %{d: 1, e: true} # this assigns val to 1; no variable d is created because the d on the lhs is really just a symbol that is used to create the pattern %{:d => _} (note that hash rocket notation allows one to have non-symbols as keys for maps just like in ruby)

Remarks

As for which data structure to us here are some brief remarks.

If you need an array data structure if you're going to be doing a lot of writing use lists. If instead you are going to be doing a lot of read you should use tuples.

As for maps they are just simply how you do key value stores.

Examples

Lists

```
a = [1, 2, 3, true]
```

Note that these are stored in memory as linked lists. Id est this is a series of cons cells where the head (List.hd/1) is the value of first item of the list and the tail (List.tail/1) is the value of the rest of the list.

```
List.hd(a) = 1
List.tl(a) = [2, 3, true]
```

Tuples

```
b = {:ok, 1, 2}
```

Tuples are the equivalent of arrays in other languages. They are stored contiguously in memory.

Read Data Structures online: https://riptutorial.com/elixir/topic/1607/data-structures

Chapter 12: Debugging Tips

Examples

Debugging with IEX.pry/0

Debugging with `IEx.pry/0` is quite simple.

1. `require IEx` in your module
2. Find the line of code you want to inspect
3. Add `IEx.pry` after the line

Now start your project (e.g. `iex -S mix`).

When the line with `IEx.pry/0` is reached the program will stop and you have the chance to inspect. It is like a breakpoint in a traditional debugger.

When you are finished just type `respawn` into the console.

```
require IEx;

defmodule Example do
  def double_sum(x, y) do
    IEx.pry
    hard_work(x, y)
  end

  defp hard_work(x, y) do
    2 * (x + y)
  end
end
```

Debugging with IO.inspect/1

It is possible to use IO.inspect/1 as a tool to debug an elixir program.

```
defmodule MyModule do
  def myfunction(argument_1, argument_2) do
    IO.inspect(argument_1)
    IO.inspect(argument_2)
  end
end
```

It will print out argument_1 and argument_2 to the console. Since `IO.inspect/1` returns its argument it is very easy to include it in function calls or pipelines without breaking the flow:

```
do_something(a, b)
|> do_something_else(c)

# can be adorned with IO.inspect, with no change in functionality:
```

```
do_something(IO.inspect(a), IO.inspect(b))
|> IO.inspect
do_something(IO.inspect(c))
```

Debug in pipe

```
defmodule Demo do
  def foo do
    1..10
    |> Enum.map(&(&1 * &1))           |> p
    |> Enum.filter(&rem(&1, 2) == 0) |> p
    |> Enum.take(3)                   |> p
  end

  defp p(e) do
    require Logger
    Logger.debug inspect e, limit: :infinity
    e
  end
end
```

```
iex(1)> Demo.foo

23:23:55.171 [debug] [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

23:23:55.171 [debug] [4, 16, 36, 64, 100]

23:23:55.171 [debug] [4, 16, 36]

[4, 16, 36]
```

Pry in pipe

```
defmodule Demo do
  def foo do
    1..10
    |> Enum.map(&(&1 * &1))
    |> Enum.filter(&rem(&1, 2) == 0) |> pry
    |> Enum.take(3)
  end

  defp pry(e) do
    require IEx
    IEx.pry
    e
  end
end
```

```
iex(1)> Demo.foo
Request to pry #PID<0.117.0> at lib/demo.ex:11

      def pry(e) do
        require IEx
```

```
        IEx.pry
        e
      end

Allow? [Yn] Y

Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
pry(1)> e
[4, 16, 36, 64, 100]
pry(2)> respawn

Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
[4, 16, 36]
iex(1)>
```

Read Debugging Tips online: https://riptutorial.com/elixir/topic/2719/debugging-tips

Chapter 13: Doctests

Examples

Introduction

When you document your code with `@doc`, you can supply code examples like so:

```
# myproject/lib/my_module.exs

defmodule MyModule do
  @doc """
  Given a number, returns `true` if the number is even, otherwise `false`.

  ## Example
    iex> MyModule.even?(2)
    true
    iex> MyModule.even?(3)
    false
  """
  def even?(number) do
    rem(number, 2) == 0
  end
end
```

You can add the code examples as test cases into one of your test suites:

```
# myproject/test/doc_test.exs

defmodule DocTest do
  use ExUnit.Case
  doctest MyModule
end
```

Then, you can then run your tests with `mix test`.

Generating HTML documentation based on doctest

Because generating documentation is based on markdown, you have to do 2 things :

1/ Write your doctest and make your doctest examples clear to improve readability (It is better to give a headline, like "examples" or "tests"). When you write your tests, do not forget to give 4 spaces to your tests code so that it will be formatting as code in the HTML documentation.

2/ Then, enter "mix docs" in console at the root of your elixir project to generate the HTML documentation in the doc directory located in the root of your elixir project.

```
$> mix docs
```

Multiline doctests

You can do a multiline doctest by using '...>' for the lines following the first

```
iex> Foo.Bar.somethingConditional("baz")
...>   |> case do
...>       {:ok, _} -> true
...>       {:error, _} -> false
...>      end
true
```

Read Doctests online: https://riptutorial.com/elixir/topic/2708/doctests

Chapter 14: Ecto

Examples

Adding a Ecto.Repo in an elixir program

This can be done in 3 steps :

1. You must define an elixir module which use Ecto.Repo and register your app as an otp_app.

```
defmodule Repo do
  use Ecto.Repo, otp_app: :custom_app
end
```

2. You must also define some config for the Repo which will allow you to connect to the database. Here is an example with postgres.

```
config :custom_app, Repo,
   adapter: Ecto.Adapters.Postgres,
   database: "ecto_custom_dev",
   username: "postgres_dev",
   password: "postgres_dev",
   hostname: "localhost",
  # OR use a URL to connect instead
  url: "postgres://postgres_dev:postgres_dev@localhost/ecto_custom_dev"
```

3. Before using Ecto in your application, you need to ensure that Ecto is started before your app is started. It can be done with registering Ecto in lib/custom_app.ex as a supervisor.

```
    def start(_type, _args) do
      import Supervisor.Spec

      children = [
       supervisor(Repo, [])
      ]

      opts = [strategy: :one_for_one, name: MyApp.Supervisor]
      Supervisor.start_link(children, opts)
    end
```

"and" clause in a Repo.get_by/3

If you have an Ecto.Queryable, named Post, which has a title and an description.

You can fetch the Post with title: "hello" and description : "world" by performing :

```
MyRepo.get_by(Post, [title: "hello", description: "world"])
```

All of this is possible because Repo.get_by expects in second argument a Keyword List.

Querying with dynamic fields

To query a field which name is contained in a variable, use the field function.

```
some_field = :id
some_value = 10

from p in Post, where: field(p, ^some_field) == ^some_value
```

Add custom data types to migration and to schema

(From this answer)

The example below adds an enumerated type to a postgres database.

First, edit the **migration file** (created with `mix ecto.gen.migration`):

```
def up do
  # creating the enumerated type
  execute("CREATE TYPE post_status AS ENUM ('published', 'editing')")

  # creating a table with the column
  create table(:posts) do
    add :post_status, :post_status, null: false
  end
end

def down do
  drop table(:posts)
  execute("DROP TYPE post_status")
end
```

Second, in the **model file** either add a field with an Elixir type :

```
schema "posts" do
  field :post_status, :string
end
```

or implement the `Ecto.Type` behaviour.

A good example for the latter is the `ecto_enum` package and it can be used as a template. Its usage is well documented on its **github page**.

This commit shows an example usage in a Phoenix project from adding enum_ecto to the project and using the enumerated type in views and models.

Read Ecto online: https://riptutorial.com/elixir/topic/6524/ecto

Chapter 15: Erlang

Examples

Using Erlang

Erlang modules are available as atoms. For example, the Erlang math module is available as `:math`:

```
iex> :math.pi
3.141592653589793
```

Inspect an Erlang module

Use `module_info` on Erlang modules you wish to inspect:

```
iex> :math.module_info
 [module: :math,
  exports: [pi: 0, module_info: 0, module_info: 1, pow: 2, atan2: 2, sqrt: 1,
   log10: 1, log2: 1, log: 1, exp: 1, erfc: 1, erf: 1, atanh: 1, atan: 1,
   asinh: 1, asin: 1, acosh: 1, acos: 1, tanh: 1, tan: 1, sinh: 1, sin: 1,
   cosh: 1, cos: 1],
  attributes: [vsn: [113168357788724588783826225069997113388]],
  compile: [options: [{:outdir,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/../ebin'},
    {:i,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/../include'},
    {:i,
     '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-
18.3/lib/stdlib/src/../../kernel/include'},
    :warnings_as_errors, :debug_info], version: '6.0.2',
   time: {2016, 3, 16, 16, 40, 35},
   source: '/private/tmp/erlang20160316-36404-xtp7cq/otp-OTP-18.3/lib/stdlib/src/math.erl'],
  native: false,
  md5: <<85, 35, 110, 210, 174, 113, 103, 228, 63, 252, 81, 27, 224, 15, 64,
    44>>]
```

Read Erlang online: https://riptutorial.com/elixir/topic/2716/erlang

Chapter 16: ExDoc

Examples

Introduction

To generate documentation in `HTML` format from `@doc` and `@moduledoc` attributes in your source code, add `ex_doc` and a markdown processor, right now ExDoc supports Earmark, Pandoc, Hoedown and Cmark, as dependencies into your `mix.exs` file:

```
# config/mix.exs

def deps do
  [{:ex_doc, "~> 0.11", only: :dev},
   {:earmark, "~> 0.1", only: :dev}]
end
```

If you want to use another Markdown processor, you can find more information in the Changing the Markdown tool section.

You can use Markdown within Elixir `@doc` and `@moduledoc` attributes.

Then, run `mix docs`.

One thing to keep in mind is that ExDoc allows configuration parameters, such as:

```
  def project do
    [app: :my_app,
     version: "0.1.0-dev",
     name: "My App",
     source_url: "https://github.com/USER/APP",
     homepage_url: "http://YOUR_PROJECT_HOMEPAGE",
     deps: deps(),
     docs: [logo: "path/to/logo.png",
            output: "docs",
            main: "README",
            extra_section: "GUIDES",
            extras: ["README.md", "CONTRIBUTING.md"]]]
  end
```

You can see more information about this configuration options with `mix help docs`

Read ExDoc online: https://riptutorial.com/elixir/topic/3582/exdoc

Chapter 17: ExUnit

Examples

Asserting Exceptions

Use `assert_raise` to test if an exception was raised. `assert_raise` takes in an Exception and a function to be executed.

```
test "invalid block size" do
  assert_raise(MerkleTree.ArgumentError, (fn() -> MerkleTree.new ["a", "b", "c"] end))
end
```

Wrap any code you want to test in an anonymous function and pass it to `assert_raise`.

Read ExUnit online: https://riptutorial.com/elixir/topic/3583/exunit

Chapter 18: Functional programming in Elixir

Introduction

Let's try to implement the basic higher orders functions like map and reduce using Elixir

Examples

Map

Map is a function which will take an array and a function and return an array after applying that function to ***each element*** in that list

```
defmodule MyList do
  def map([], _func) do
    []
  end

  def map([head | tail], func) do
    [func.(head) | map(tail, func)]
  end
end
```

Copy paste in `iex` and execute:

```
MyList.map [1,2,3], fn a -> a * 5 end
```

Shorthand syntax is `MyList.map [1,2,3], &(&1 * 5)`

Reduce

Reduce is a function which will take an array, function and accumulator and use ***accumulator as seed to start the iteration with the first element to give next accumulator and the iteration continues for all the elements in the array*** (refer below example)

```
defmodule MyList do
  def reduce([], _func, acc) do
    acc
  end

  def reduce([head | tail], func, acc) do
    reduce(tail, func, func.(acc, head))
  end
end
```

Copy paste the above snippet in iex:

1. To add all numbers in an array: `MyList.reduce [1,2,3,4], fn acc, element -> acc + element end, 0`

2. To mutliply all numbers in an array: `MyList.reduce [1,2,3,4], fn acc, element -> acc * element end, 1`

Explanation for example 1:

```
Iteration 1 => acc = 0, element = 1 ==> 0 + 1 ===> 1 = next accumulator
Iteration 2 => acc = 1, element = 2 ==> 1 + 2 ===> 3 = next accumulator
Iteration 3 => acc = 3, element = 3 ==> 3 + 3 ===> 6 = next accumulator
Iteration 4 => acc = 6, element = 4 ==> 6 + 4 ===> 10 = next accumulator = result(as all
elements are done)
```

Filter the list using reduce

```
MyList.reduce [1,2,3,4], fn acc, element -> if rem(element,2) == 0 do acc else acc ++
[element] end end, []
```

Read Functional programming in Elixir online: https://riptutorial.com/elixir/topic/10186/functional-programming-in-elixir

Chapter 19: Functions

Examples

Anonymous Functions

In Elixir, a common practice is to use anonymous functions. Creating an anonymous function is simple:

```
iex(1)> my_func = fn x -> x * 2 end
#Function<6.52032458/1 in :erl_eval.expr/5>
```

The general syntax is:

```
fn args -> output end
```

For readability, you may put parenthesis around the arguments:

```
iex(2)> my_func = fn (x, y) -> x*y end
#Function<12.52032458/2 in :erl_eval.expr/5>
```

To invoke an anonymous function, call it by the assigned name and add `.` between the name and arguments.

```
iex(3)>my_func.(7, 5)
35
```

It is possible to declare anonymous functions without arguments:

```
iex(4)> my_func2 = fn -> IO.puts "hello there" end
iex(5)> my_func2.()
hello there
:ok
```

Using the capture operator

To make anonymous functions more concise you can use the **capture operator** `&`. For example, instead of:

```
iex(5)> my_func = fn (x) -> x*x*x end
```

You can write:

```
iex(6)> my_func = &(&1*&1*&1)
```

With multiple parameters, use the number corresponding to each argument, counting from `1`:

```
iex(7)> my_func = fn (x, y) -> x + y end

iex(8)> my_func = &(&1 + &2)    # &1 stands for x and &2 stands for y

iex(9)> my_func.(4, 5)
9
```

Multiple bodies

An anonymous function can also have multiple bodies (as a result of pattern matching):

```
my_func = fn
  param1 -> do_this
  param2 -> do_that
end
```

When you call a function with multiple bodies Elixir attempts to match the parameters you have provided with the proper function body.

Keyword lists as function parameters

Use keyword lists for 'options'-style parameters that contains multiple key-value pairs:

```
def myfunc(arg1, opts \\ []) do
  # Function body
end
```

We can call the function above like so:

```
iex> myfunc "hello", pizza: true, soda: false
```

which is equivalent to:

```
iex> myfunc("hello", [pizza: true, soda: false])
```

The argument values are available as `opts.pizza` and `opts.soda` respectively.
Alternatively, you could use atoms: `opts[:pizza]` and `opts[:soda]`.

Named Functions & Private Functions

Named Functions

```
defmodule Math do
    # one way
    def add(a, b) do
        a + b
```

```
    end

    # another way
    def subtract(a, b), do: a - b
end

iex> Math.add(2, 3)
5
:ok
iex> Math.subtract(5, 2)
3
:ok
```

Private Functions

```
defmodule Math do
    def sum(a, b) do
        add(a, b)
    end

    # Private Function
    defp add(a, b) do
        a + b
    end
end

iex> Math.add(2, 3)
** (UndefinedFunctionError) undefined function Math.add/2
Math.add(3, 4)
iex> Math.sum(2, 3)
5
```

Pattern Matching

Elixir matches a function call to its body based on the value of its arguments.

```
defmodule Math do
    def factorial(0): do: 1
    def factorial(n): do: n * factorial(n - 1)
end
```

Here, factorial of positive numbers matches the second clause, while `factorial(0)` matches the first. (ignoring negative numbers for the sake of simplicity). Elixir tries to match the functions from top to bottom. If the second function is written above the first, we will an unexpected result as it goes to an endless recursion. Because `factorial(0)` matches to `factorial(n)`

Guard clauses

Guard clauses enables us to check the arguments before executing the function. Guard clauses are usually preferred to `if` and `cond` due to their readability, and to make a certain optimization technique easier for the compiler. The first function definition where all guards match is executed.

Here is an example implementation of the factorial function using guards and pattern matching.

```
defmodule Math do
    def factorial(0), do: 1
    def factorial(n) when n > 0: do: n * factorial(n - 1)
end
```

The first pattern matches if (and only if) the argument is `0`. If the argument is not `0`, the pattern match fails and the next function below is checked.

That second function definition has a guard clause: `when n > 0`. This means that this function only matches if the argument `n` is greater than `0`. After all, the mathematical factorial function is not defined for negative integers.

If neither function definition (including their pattern matching and guard clauses) match, a `FunctionClauseError` will be raised. This happens for this function when we pass a negative number as the argument, since it is not defined for negative numbers.

Note that this `FunctionClauseError` itself, is not a mistake. Returning `-1` or `0` or some other "error value" as is common in some other languages would hide the fact that you called an undefined function, hiding the source of the error, possibly creating a huge painful bug for a future developer.

Default Parameters

You can pass default parameters to any named function using the syntax: `param \\ value`:

```
defmodule Example do
    def func(p1, p2 \\ 2) do
        IO.inspect [p1, p2]
    end
end

Example.func("a")     # => ["a", 2]
Example.func("b", 4) # => ["b", 4]
```

Capture functions

Use `&` to capture functions from other modules. You can use the captured functions directly as function parameters or within anonymous functions.

```
Enum.map(list, fn(x) -> String.capitalize(x) end)
```

Can be made more concise using `&`:

```
Enum.map(list, &String.capitalize(&1))
```

Capturing functions without passing any arguments require you to explicitly specify its arity, e.g. `&String.capitalize/1`:

```
defmodule Bob do
  def say(message, f \\ &String.capitalize/1) do
    f.(message)
```

```
  end
end
```

Read Functions online: https://riptutorial.com/elixir/topic/2442/functions

Chapter 20: Getting help in IEx console

Introduction

IEx provides access to Elixir documentation. When Elixir is installed on your system you can start IEx e.g. with `iex` command in a terminal. Then type `h` command on IEx command line followed by the function name prepended by its module name e.g. `h List.foldr`

Examples

Listing Elixir modules and functions

To get the list of Elixir modules just type

```
h Elixir.[TAB]
```

Pressing [TAB] autocompletes modules and functions names. In this case it lists all modules. To find all functions in a module e.g. `List` use

```
h List.[TAB]
```

Read Getting help in IEx console online: https://riptutorial.com/elixir/topic/10780/getting-help-in-iex-console

Chapter 21: IEx Console Tips & Tricks

Examples

Recompile project with `recompile`

```
iex(1)> recompile
Compiling 1 file (.ex)
:ok
```

See documentation with `h`

```
iex(1)> h List.last

                                  def last(list)

Returns the last element in list or nil if list is empty.

Examples

| iex> List.last([])
| nil
|
| iex> List.last([1])
| 1
|
| iex> List.last([1, 2, 3])
| 3
```

Get value from last command with `v`

```
iex(1)> 1 + 1
2
iex(2)> v
2
iex(3)> 1 + v
3
```

See also: Get the value of a row with `v`

Get the value of a previous command with `v`

```
iex(1)> a = 10
10
iex(2)> b = 20
20
iex(3)> a + b
30
```

You can get a specific row passing the index of the row:

```
iex(4)> v(3)
30
```

You can also specify an index relative to the current row:

```
iex(5)> v(-1) # Retrieves value of row (5-1) -> 4
30
iex(6)> v(-5) # Retrieves value of row (5-4) -> 1
10
```

The value can be reused in other calculations:

```
iex(7)> v(2) * 4
80
```

If you specify a non-existing row, `IEx` will raise an error:

```
iex(7)> v(100)
** (RuntimeError) v(100) is out of bounds
    (iex) lib/iex/history.ex:121: IEx.History.nth/2
    (iex) lib/iex/helpers.ex:357: IEx.Helpers.v/1
```

Exit IEx console

1. Use Ctrl + C, Ctrl + C to exit

```
iex(1)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

2. Use `Ctrl+ \` to immediately exit

See information with `i`

```
iex(1)> i :ok
Term
  :ok
Data type
  Atom
Reference modules
  Atom
iex(2)> x = "mystring"
"mystring"
iex(3)> i x
Term
  "mystring"
Data type
  BitString
Byte size
  8
Description
  This is a string: a UTF-8 encoded binary. It's printed surrounded by
  "double quotes" because all UTF-8 encoded codepoints in it are printable.
```

```
Raw representation
  <<109, 121, 115, 116, 114, 105, 110, 103>>
Reference modules
  String, :binary
```

Creating PID

This is useful when you didn't store the PID from a previous command

```
iex(1)> self()
#PID<0.138.0>
iex(2)> pid("0.138.0")
#PID<0.138.0>
iex(3)> pid(0, 138, 0)
#PID<0.138.0>
```

Have your aliases ready when you start IEx

If you put your commonly used aliases into an `.iex.exs` file at the root of your app, IEx will load them for you on startup.

```
alias App.{User, Repo}
```

Persistent history

By default, user input history in `IEx` do not persist across different sessions.

`erlang-history` adds history support to both the Erlang shell and `IEx`:

```
git clone git@github.com:ferd/erlang-history.git
cd erlang-history
sudo make install
```

You can now access your previous inputs using the up and down arrow keys, even across different `IEx` sessions.

When Elixir console is stuck...

Sometimes you might accidentally run something in the shell that ends up waiting forever, and thus blocking the shell:

```
iex(2)> receive do _ -> :stuck end
```

In that case, press Ctrl-g. You'll see:

```
User switch command
```

Enter these commands in order:

- `k` (to kill the shell process)

- `s` (to start a new shell process)
- `c` (to connect to the new shell process)

You'll end up in a new Erlang shell:

```
Eshell V8.0.2  (abort with ^G)
1>
```

To start an Elixir shell, type:

```
'Elixir.IEx.CLI':local_start().
```

(don't forget the final dot!)

Then you'll see a new Elixir shell process coming up:

```
Interactive Elixir (1.3.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> "I'm back"
"I'm back"
iex(2)>
```

To escape from “awaiting-for-more-input” mode (due to unclosed quotation mark, bracket etc,) type **`#iex:break`**, followed by carriage return (␍):

```
iex(1)> "Hello, "world"
...(1)>
...(1)> #iex:break
** (TokenMissingError) iex:1: incomplete expression

iex(1)>
```

the above is specifically useful when copy-pasting a relatively huge snippet turns the console to “awaiting-for-more-input” mode.

break out of incomplete expression

When you have entered something into IEx which expects a completion, such as a multiline string, IEx will change the prompt to indicate that it is waiting for you finish by changing the prompt to have an ellipsis (`...`) rather than `iex`.

If you find that IEx is waiting for you to finish an expression but you aren't sure what it needs to terminate the expression, or you simply want to abort this line of input, enter `#iex:break` as the console input. This will cause IEx to throw a `TokenMissingError` and cancel waiting for any more input, returning you to a standard "top-level" console input.

```
iex:1> "foo"
"foo"
iex:2> "bar
...:2> #iex:break
** (TokenMissingError) iex:2: incomplete expression
```

More info is available at the IEx documentation.

Load a module or script into the IEx session

If you have an elixir file; a script or a module and want to load it into the current IEx session, you can use the `c/1` method:

```
iex(1)> c "lib/utils.ex"
iex(2)> Utils.some_method
```

This will compile and load the module in IEx, and you'll be able to call all of it's public methods.

For scripts, it will immediately execute the contents of the script:

```
iex(3)> c "/path/to/my/script.exs"
Called from within the script!
```

Read IEx Console Tips & Tricks online: https://riptutorial.com/elixir/topic/1283/iex-console-tips---tricks

Chapter 22: Installation

Examples

Fedora Installation

```
dnf install erlang elixir
```

OSX Installation

On OS X and MacOS, Elixir can be installed via the common package managers:

Homebrew

```
$ brew update
$ brew install elixir
```

Macports

```
 $ sudo port install elixir
```

Debian/Ubuntu Installation

```
# Fetch and install package to setup access to the official APT repository
wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb
sudo dpkg -i erlang-solutions_1.0_all.deb

# Update package index
sudo apt-get update

# Install Erlang and Elixir
sudo apt-get install esl-erlang
sudo apt-get install elixir
```

Gentoo/Funtoo Installation

Elixir is available in main packages repository.
Update the packages list before installing any package:

```
emerge --sync
```

This is one step installation:

```
emerge --ask dev-lang/elixir
```

Read Installation online: https://riptutorial.com/elixir/topic/4208/installation

Chapter 23: Join Strings

Examples

Using String Interpolation

```
iex(1)> [x, y] = ["String1", "String2"]
iex(2)> "#{x} #{y}"
# "String1 String2"
```

Using IO List

```
["String1", " ", "String2"] |> IO.iodata_to_binary
# "String1 String2"
```

This will gives some performances boosts as strings not duplicated in memory.
Alternative method:

```
iex(1)> IO.puts(["String1", " ", "String2"])
# String1 String2
```

Using Enum.join

```
Enum.join(["String1", "String2"], " ")
# "String1 String2"
```

Read Join Strings online: https://riptutorial.com/elixir/topic/9202/join-strings

Chapter 24: Lists

Syntax

- []
- [1, 2, 3, 4]
- [1, 2] ++ [3, 4] # -> [1,2,3,4]
- hd([1, 2, 3, 4]) # -> 1
- tl([1, 2, 3, 4]) # -> [2,3,4]
- [head | tail]
- [1 | [2, 3, 4]] # -> [1,2,3,4]
- [1 | [2 | [3 | [4 | []]]]] -> [1,2,3,4]
- 'hello' = [?h, ?e, ?l, ?l, ?o]
- keyword_list = [a: 123, b: 456, c: 789]
- keyword_list[:a] # -> 123

Examples

Keyword Lists

Keyword lists are lists where each item in the list is a tuple of an atom followed by a value.

```
keyword_list = [{:a, 123}, {:b, 456}, {:c, 789}]
```

A shorthand notation for writing keyword lists is as follows:

```
keyword_list = [a: 123, b: 456, c: 789]
```

Keyword lists are useful for creating ordered key-value pair data structures, where multiple items can exist for a given key.

The first item in a keyword list for a given key can be obtained like so:

```
iex> keyword_list[:b]
456
```

A use case for keyword lists could be a sequence of named tasks to run:

```
defmodule TaskRunner do
  def run_tasks(tasks) do
    # Call a function for each item in the keyword list.
    # Use pattern matching on each {:key, value} tuple in the keyword list
    Enum.each(tasks, fn
      {:delete, x} ->
        IO.puts("Deleting record " <> to_string(x) <> "...")
      {:add, value} ->
```

```
        IO.puts("Adding record \"" <> value <> "\"...")
      {:update, {x, value}} ->
        IO.puts("Setting record " <> to_string(x) <> " to \"" <> value <> "\"...")
    end)
  end
end
```

This code can be called with a keyword list like so:

```
iex> tasks = [
...>   add: "foo",
...>   add: "bar",
...>   add: "test",
...>   delete: 2,
...>   update: {1, "asdf"}
...> ]

iex> TaskRunner.run_tasks(tasks)
Adding record "foo"...
Adding record "bar"...
Adding record "test"...
Deleting record 2...
Setting record 1 to "asdf"...
```

Char Lists

Strings in Elixir are "binaries". However, in Erlang code, strings are traditionally "char lists", so when calling Erlang functions, you may have to use char lists instead of regular Elixir strings.

While regular strings are written using double quotes `"`, char lists are written using single quotes `'`:

```
string = "Hello!"
char_list = 'Hello!'
```

Char lists are simply lists of integers representing the code points of each character.

```
'hello' = [104, 101, 108, 108, 111]
```

A string can be converted to a char list with `to_charlist/1`:

```
iex> to_charlist("hello")
'hello'
```

And the reverse can be done with `to_string/1`:

```
iex> to_string('hello')
"hello"
```

Calling an Erlang function and converting the output to a regular Elixir string:

```
iex> :os.getenv |> hd |> to_string
"PATH=/usr/local/bin:/usr/bin:/bin"
```

Cons Cells

Lists in Elixir are linked lists. This means that each item in a list consists of a value, followed by a pointer to the next item in the list. This is implemented in Elixir using cons cells.

Cons cells are simple data structures with a "left" and a "right" value, or a "head" and a "tail".

A `|` symbol can be added before the last item in a list to notate an (improper) list with a given head and tail. The following is a single cons cell with `1` as the head and `2` as the tail:

```
[1 | 2]
```

The standard Elixir syntax for a list is actually equivalent to writing a chain of nested cons cells:

```
[1, 2, 3, 4] = [1 | [2 | [3 | [4 | []]]]]
```

The empty list `[]` is used as the tail of a cons cell to represent the end of a list.

All lists in Elixir are equivalent to the form `[head | tail]`, where `head` is the first item of the list and `tail` is the rest of the list, minus the head.

```
iex> [head | tail] = [1, 2, 3, 4]
[1, 2, 3, 4]
iex> head
1
iex> tail
[2, 3, 4]
```

Using the `[head | tail]` notation is useful for pattern matching in recursive functions:

```
def sum([]), do: 0

def sum([head | tail]) do
  head + sum(tail)
end
```

Mapping Lists

`map` is a function in functional programming which given a list and a function, returns a new list with the function applied to each item in that list. In Elixir, the `map/2` function is in the Enum module.

```
iex> Enum.map([1, 2, 3, 4], fn(x) -> x + 1 end)
[2, 3, 4, 5]
```

Using the alternative capture syntax for anonymous functions:

```
iex> Enum.map([1, 2, 3, 4], &(&1 + 1))
[2, 3, 4, 5]
```

Referring to a function with capture syntax:

```
iex> Enum.map([1, 2, 3, 4], &to_string/1)
["1", "2", "3", "4"]
```

Chaining list operations using the pipe operator:

```
iex> [1, 2, 3, 4]
...> |> Enum.map(&to_string/1)
...> |> Enum.map(&("Chapter " <> &1))
["Chapter 1", "Chapter 2", "Chapter 3", "Chapter 4"]
```

List Comprehensions

Elixir doesn't have loops. Instead of them for lists there are great `Enum` and `List` modules, but there are also List Comprehensions.

List Comprehensions can be useful to:

- create new lists

```
iex(1)> for value <- [1, 2, 3], do: value + 1
[2, 3, 4]
```

- filtering lists, using `guard` expressions but you use them without `when` keyword.

```
iex(2)> odd? = fn x -> rem(x, 2) == 1 end
iex(3)> for value <- [1, 2, 3], odd?.(value), do: value
[1, 3]
```

- create custom map, using `into` keyword:

```
iex(4)> for value <- [1, 2, 3], into: %{}, do: {value, value + 1}
%{1 => 2, 2=>3, 3 => 4}
```

Combined example

```
iex(5)> for value <- [1, 2, 3], odd?.(value), into: %{}, do: {value, value * value}
%{1 => 1, 3 => 9}
```

Summary

List Comprehensions:

- uses `for..do` syntax with additional guards after commas and `into` keyword when returning other structure than lists ie. map.

- in other cases return new lists
- doesn't support accumulators
- can't stop processing when certain condition is met
- `guard` statements have to be first in order after `for` and before `do` or `into` symbols. Order of symbols doesn't matter

According to these constraints List Comprehensions are limited only for simple usage. In more advanced cases using functions from `Enum` and `List` modules would be the best idea.

List difference

```
iex> [1, 2, 3] -- [1, 3]
[2]
```

`--` removes the first occurrence of an item on the left list for each item on the right.

List Membership

Use `in` operator to check if an element is a member of a list.

```
iex> 2 in [1, 2, 3]
true
iex> "bob" in [1, 2, 3]
false
```

Converting Lists to a Map

Use `Enum.chunk/2` to group elements into sub-lists, and `Map.new/2` to convert it into a Map:

```
[1, 2, 3, 4, 5, 6]
|> Enum.chunk(2)
|> Map.new(fn [k, v] -> {k, v} end)
```

Would give:

```
%{1 => 2, 3 => 4, 5 => 6}
```

Read Lists online: https://riptutorial.com/elixir/topic/1279/lists

Chapter 25: Maps and Keyword Lists

Syntax

- map = %{} // creates an empty map
- map = %{:a => 1, :b => 2} // creates a non-empty map
- list = [] // creates an empty list
- list = [{:a, 1}, {:b, 2}] // creates a non-empty keyword list

Remarks

Elixir provides two associative data structures: *maps* and *keyword lists*.

Maps are the Elixir key-value (also called dictionary or hash in other languages) type.

Keyword lists are tuples of key/value that associate a value to a certain key. They are generally used as options for a function call.

Examples

Creating a Map

Maps are the Elixir key-value (also called dictionary or hash in other languages) type. You create a map using the `%w{}` syntax:

```
%{} // creates an empty map
%{:a => 1, :b => 2} // creates a non-empty map
```

Keys and values can use be any type:

```
%{"a" => 1, "b" => 2}
%{1 => "a", 2 => "b"}
```

Moreover, you can have maps with mixed types for both keys and values":

```
// keys are integer or strings
%{1 => "a", "b" => :foo}
// values are string or nil
%{1 => "a", 2 => nil}
```

When all the keys in a map are atoms, you can use the keyword syntax for convenience:

```
%{a: 1, b: 2}
```

Creating a Keyword List

Keyword lists are tuples of key/value, generally used as options for a function call.

```
[{:a, 1}, {:b, 2}] // creates a non-empty keyword list
```

Keyword lists can have the same key repeated more than once.

```
[{:a, 1}, {:a, 2}, {:b, 2}]
[{:a, 1}, {:b, 2}, {:a, 2}]
```

Keys and values can be any type:

```
[{"a", 1}, {:a, 2}, {2, "b"}]
```

Difference between Maps and Keyword Lists

Maps and keyword lists have different application. For instance, a map cannot have two keys with the same value and it's not ordered. Conversely, a Keyword list can be a little bit hard to use in pattern matching in some cases.

Here's a few use cases for maps vs keyword lists.

Use keyword lists when:

- you need the elements to be ordered
- you need more than one element with the same key

Use maps when:

- you want to pattern-match against some keys/values
- you don't need more than one element with the same key
- whenever you don't explicitly need a keyword list

Read Maps and Keyword Lists online: https://riptutorial.com/elixir/topic/2706/maps-and-keyword-lists

Chapter 26: Metaprogramming

Examples

Generate tests at compile time

```
defmodule ATest do
  use ExUnit.Case

  [{1, 2, 3}, {10, 20, 40}, {100, 200, 300}]
  |> Enum.each(fn {a, b, c} ->
    test "#{a} + #{b} = #{c}" do
      assert unquote(a) + unquote(b) = unquote(c)
    end
  end)
end
```

Output:

```
.

  1) test 10 + 20 = 40 (Test.Test)
     test.exs:6
     match (=) failed
     code: 10 + 20 = 40
     rhs:  40
     stacktrace:
       test.exs:7

.

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
3 tests, 1 failure
```

Read Metaprogramming online: https://riptutorial.com/elixir/topic/4069/metaprogramming

Chapter 27: Mix

Examples

Create a Custom Mix Task

```
# lib/mix/tasks/mytask.ex
defmodule Mix.Tasks.MyTask do
  use Mix.Task

  @shortdoc "A simple mix task"
  def run(_) do
    IO.puts "YO!"
  end
end
```

Compile and run:

```
$ mix compile
$ mix my_task
"YO!"
```

Custom mix task with command line arguments

In a basic implementation the task module must define a `run/1` function that takes a list of arguments. E.g. `def run(args) do ... end`

```
defmodule Mix.Tasks.Example_Task do
  use Mix.Task

  @shortdoc "Example_Task prints hello + its arguments"
  def run(args) do
    IO.puts "Hello #{args}"
  end
end
```

Compile and run:

```
$ mix example_task world
"hello world"
```

Aliases

Elixir allows you to add aliases for your mix commands. Cool thing if you want to save yourself some typing.

Open `mix.exs` in your Elixir project.

First, add `aliases/0` function to the keyword list that the `project` function returns. *Adding `()` at the*

end of the aliases function will prevent compiler from throwing a warning.

```
def project do
  [app: :my_app,
   ...
   aliases: aliases()]
end
```

Then, define your `aliases/0` function (e.g. at the bottom of your `mix.exs` file).

```
...

defp aliases do
  [go: "phoenix.server",
   trident: "do deps.get, compile, go"]
end
```

You can now use `$ mix go` to run your Phoenix server (if you're running a Phoenix application). And use `$ mix trident` to tell mix to fetch all dependencies, compile, and run the server.

Get help on available mix tasks

To list available mix tasks use:

```
mix help
```

To get help on a specific task use `mix help task` e.g.:

```
mix help cmd
```

Read Mix online: https://riptutorial.com/elixir/topic/3585/mix

Chapter 28: Modules

Remarks

Module Names

In Elixir, module names such as `IO` or `String` are just atoms under the hood and are converted to the form `:"Elixir.ModuleName"` at compile time.

```
iex(1)> is_atom(IO)
true
iex(2)> IO == :"Elixir.IO"
true
```

Examples

List a module's functions or macros

The `__info__/1` function takes one of the following atoms:

- `:functions` - Returns a keyword list of public functions along with their arities
- `:macros` - Returns a keyword list of public macros along with their arities

To list the `Kernel` module's functions:

```
iex> Kernel.__info__ :functions
[!=: 2, !==: 2, *: 2, +: 1, +: 2, ++: 2, -: 1, -: 2, --: 2, /: 2, <: 2, <=: 2,
 ==: 2, ===: 2, =~: 2, >: 2, >=: 2, abs: 1, apply: 2, apply: 3, binary_part: 3,
 bit_size: 1, byte_size: 1, div: 2, elem: 2, exit: 1, function_exported?: 3,
 get_and_update_in: 3, get_in: 2, hd: 1, inspect: 1, inspect: 2, is_atom: 1,
 is_binary: 1, is_bitstring: 1, is_boolean: 1, is_float: 1, is_function: 1,
 is_function: 2, is_integer: 1, is_list: 1, is_map: 1, is_number: 1, is_pid: 1,
 is_port: 1, is_reference: 1, is_tuple: 1, length: 1, macro_exported?: 3,
 make_ref: 0, ...]
```

Replace `Kernel` with any module of your choosing.

Using modules

Modules have four associated keywords to make using them in other modules: `alias`, `import`, `use`, and `require`.

`alias` will register a module under a different (usually shorter) name:

```
defmodule MyModule do
  # Will make this module available as `CoolFunctions`
  alias MyOtherModule.CoolFunctions
  # Or you can specify the name to use
```

```
  alias MyOtherModule.CoolFunctions, as: CoolFuncs
end
```

`import` will make all the functions in the module available with no name in front of them:

```
defmodule MyModule do
  import Enum
  def do_things(some_list) do
    # No need for the `Enum.` prefix
    join(some_list, " ")
  end
end
```

`use` allows a module to inject code into the current module - this is typically done as part of a framework that creates its own functions to make your module confirm to some behaviour.

`require` loads macros from the module so that they can be used.

Delegating functions to another module

Use `defdelegate` to define functions that delegate to functions of the same name defined in another module:

```
defmodule Math do
  defdelegate pi, to: :math
end


iex> Math.pi
3.141592653589793
```

Read Modules online: https://riptutorial.com/elixir/topic/2721/modules

Chapter 29: Nodes

Examples

List all visible nodes in the system

```
iex(bob@127.0.0.1)> Node.list
[:"frank@127.0.0.1"]
```

Connecting nodes on the same machine

Start two named nodes in two terminal windows:

```
>iex --name bob@127.0.0.1
iex(bob@127.0.0.1)>
>iex --name frank@127.0.0.1
iex(frank@127.0.0.1)>
```

Connect two nodes by instructing one node to connect:

```
iex(bob@127.0.0.1)> Node.connect :"frank@127.0.0.1"
true
```

The two nodes are now connected and aware of each other:

```
iex(bob@127.0.0.1)> Node.list
[:"frank@127.0.0.1"]
iex(frank@127.0.0.1)> Node.list
[:"bob@127.0.0.1"]
```

You can execute code on other nodes:

```
iex(bob@127.0.0.1)> greet = fn() -> IO.puts("Hello from #{inspect(Node.self)}") end
iex(bob@127.0.0.1)> Node.spawn(:"frank@127.0.0.1", greet)
#PID<9007.74.0>
Hello from :"frank@127.0.0.1"
:ok
```

Connecting nodes on different machines

Start a named process on one IP address:

```
$ iex --name foo@10.238.82.82 --cookie chocolate
iex(foo@10.238.82.82)> Node.ping :"bar@10.238.82.85"
:pong
iex(foo@10.238.82.82)> Node.list
[:"bar@10.238.82.85"]
```

Start another named process on a different IP address:

```
$ iex --name bar@10.238.82.85 --cookie chocolate
iex(bar@10.238.82.85)> Node.list
[:"foo@10.238.82.82"]
```

Read Nodes online: https://riptutorial.com/elixir/topic/2065/nodes

Chapter 30: Operators

Examples

The Pipe Operator

The Pipe Operator `|>` takes the result of an expression on the left and feeds it as the first parameter to a function on the right.

```
expression |> function
```

Use the Pipe Operator to chain expressions together and to visually document the flow of a series of functions.

Consider the following:

```
Oven.bake(Ingredients.Mix([:flour, :cocoa, :sugar, :milk, :eggs, :butter]), :temperature)
```

In the example, `Oven.bake` comes before `Ingredients.mix`, but it is executed last. Also, it may not be obvious that `:temperature` is a parameter of `Oven.bake`

Rewriting this example using the Pipe Operator:

```
[:flour, :cocoa, :sugar, :milk, :eggs, :butter]
|> Ingredients.mix
|> Oven.bake(:temperature)
```

gives the same result, but the order of execution is clearer. Furthermore, it is clear that `:temperature` is a parameter to the `Oven.bake` call.

Note that when using the Pipe Operator, the first parameter for each function is relocated to before the Pipe Operator, and so the function being called appears to have one fewer parameter. For instance:

```
Enum.each([1, 2, 3], &(&1+1)) # produces [2, 3, 4]
```

is the same as:

```
[1, 2, 3]
|> Enum.each(&(&1+1))
```

Pipe operator and parentheses

Parentheses are needed to avoid ambiguity:

```
foo 1 |> bar 2 |> baz 3
```

Should be written as:

```
foo(1) |> bar(2) |> baz(3)
```

Boolean operators

There are two kinds of boolean operators in Elixir:

- boolean operators (they expect either `true` or `false` as their first argument)

```
x or y      # true if x is true, otherwise y

x and y     # false if x is false, otherwise y

not x       # false if x is true, otherwise true
```

All of booleans operators will raise `ArgumentError` if first argument won't be strictly boolean value, which means only `true` or `false` (`nil` is not boolean).

```
iex(1)> false and 1 # return false
iex(2)> false or 1 # return 1
iex(3)> nil and 1 # raise (ArgumentError) argument error: nil
```

- relaxed boolean operators (work with any type, everything that neither `false` nor `nil` is considered as `true`)

```
x || y     # x if x is true, otherwise y

x && y     # y if x is true, otherwise false

!x         # false if x is true, otherwise true
```

Operator `||` will always return first argument if it's truthy (Elixir treats everything except `nil` and `false` to be true in comparisions), otherwise will return second one.

```
iex(1)> 1 || 3 # return 1, because 1 is truthy
iex(2)> false || 3 # return 3
iex(3)> 3 || false # return 3
iex(4)> false || nil # return nil
iex(5)> nil || false # return false
```

Operator `&&` will always return second argument if it's truthy. Otherwise will return respectively to the arguments, `false` or `nil`.

```
iex(1)> 1 && 3 # return 3, first argument is truthy
iex(2)> false && 3 # return false
iex(3)> 3 && false # return false
iex(4)> 3 && nil # return nil
iex(5)> false && nil # return false
iex(6)> nil && false # return nil
```

Both `&&` and `||` are short-circuit operators. They only execute the right side if the left side is not enough to determine the result.

Operator `!` will return boolean value of negation of current term:

```
iex(1)> !2 # return false
iex(2)> !false # return true
iex(3)> !"Test" # return false
iex(4)> !nil # return true
```

Simple way to get boolean value of selected term is to simply double this operator:

```
iex(1)> !!true # return true
iex(2)> !!"Test" # return true
iex(3)> !!nil # return false
iex(4)> !!false # return false
```

Comparison operators

Equality:

- value equality `x == y` (`1 == 1.0 # true`)
- value inequality `x == y` (`1 != 1.0 # false`)
- strict equality `x === y` (`1 === 1.0 # false`)
- strict inequality `x === y` (`1 !== 1.0 # true`)

Comparison:

- `x > y`
- `x >= y`
- `x < y`
- `x <= y`

If types are compatible, comparison uses natural ordering. Otherwise there is general types comparison rule:

```
number < atom < reference < function < port < pid < tuple < map < list < binary
```

Join operators

You can join (concatenate) binaries (including strings) and lists:

```
iex(1)> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]

iex(2)> [1, 2, 3, 4, 5] -- [1, 3]
[2, 4, 5]

iex(3)> "qwe" <> "rty"
"qwerty"
```

'In' operator

`in` operator allows you to check whether a list or a range includes an item:

```
iex(4)> 1 in [1, 2, 3, 4]
true

iex(5)> 0 in (1..5)
false
```

Read Operators online: https://riptutorial.com/elixir/topic/1161/operators

Chapter 31: Optimization

Examples

Always measure first!

These are general tips that in general improve performance. If your code is slow, it is always important to profile it to figure out what parts are slow. Guessing is **never** enough. Improving the execution speed of something that only takes up 1% of the execution time probably isn't worth the effort. Look for the big time sinks.

To get somewhat accurate numbers, make sure the code you are optimizing is executed for at least one second when profiling. If you spend 10% of the execution time in that function, make sure the complete program execution takes up at least 10 seconds, and make sure you can run the same exact data through the code multiple times, to get repeatable numbers.

ExProf is simple to get started with.

Read Optimization online: https://riptutorial.com/elixir/topic/6062/optimization

Chapter 32: Pattern matching

Examples

Pattern matching functions

```
#You can use pattern matching to run different
#functions based on which parameters you pass

#This example uses pattern matching to start,
#run, and end a recursive function

defmodule Counter do
    def count_to do
        count_to(100, 0) #No argument, init with 100
    end

    def count_to(counter) do
        count_to(counter, 0) #Initialize the recursive function
    end

    def count_to(counter, value) when value == counter do
        #This guard clause allows me to check my arguments against
        #expressions. This ends the recursion when the value matches
        #the number I am counting to.
        :ok
    end

    def count_to(counter, value) do
        #Actually do the counting
        IO.puts value
        count_to(counter, value + 1)
    end
end
```

Pattern matching on a map

```
%{username: username} = %{username: "John Doe", id: 1}
# username == "John Doe"
```

```
%{username: username, id: 2} = %{username: "John Doe", id: 1}
** (MatchError) no match of right hand side value: %{id: 1, username: "John Doe"}
```

Pattern matching on a list

You can also pattern match on Elixir Data Structures such as Lists.

Lists

Matching on a list is quite simple.

```
[head | tail] = [1,2,3,4,5]
# head == 1
# tail == [2,3,4,5]
```

This works by matching the first (or more) elements in the list to the left hand side of the `|` (pipe) and the rest of the list to the right hand side variable of the `|`.

We can also match on specific values of a list:

```
[1,2 | tail] = [1,2,3,4,5]
# tail = [3,4,5]

[4 | tail] = [1,2,3,4,5]
** (MatchError) no match of right hand side value: [1, 2, 3, 4, 5]
```

Binding multiple consecutive values on the left of the `|` is also allowed:

```
[a, b | tail] = [1,2,3,4,5]
# a == 1
# b == 2
# tail = [3,4,5]
```

Even more complex - we can match on a specific value, and match that against a variable:

```
iex(11)> [a = 1 | tail] = [1,2,3,4,5]
# a == 1
```

Get the sum of a list using pattern matching

```
defmodule Math do
  # We start of by passing the sum/1 function a list of numbers.
  def sum(numbers) do
    do_sum(numbers, 0)
  end

  # Recurse over the list when it contains at least one element.
  # We break the list up into two parts:
  #   head: the first element of the list
  #   tail: a list of all elements except the head
  # Every time this function is executed it makes the list of numbers
  # one element smaller until it is empty.
  defp do_sum([head|tail], acc) do
    do_sum(tail, head + acc)
  end

 # When we have reached the end of the list, return the accumulated sum
  defp do_sum([], acc), do: acc
end
```

Anonymous functions

```
f = fn
  {:a, :b} -> IO.puts "Tuple {:a, :b}"
```

```
  [] -> IO.puts "Empty list"
end

f.({:a, :b}) # Tuple {:a, :b}
f.([])       # Empty list
```

Tuples

```
{ a, b, c } = { "Hello", "World", "!" }

IO.puts a # Hello
IO.puts b # World
IO.puts c # !

# Tuples of different size won't match:

{ a, b, c } = { "Hello", "World" } # (MatchError) no match of right hand side value: {
"Hello", "World" }
```

Reading a File

Pattern matching is useful for an operation like file reading which returns a tuple.

If the file `sample.txt` contains `This is a sample text`, then:

```
{ :ok, file } = File.read("sample.txt")
# => {:ok, "This is a sample text"}

file
# => "This is a sample text"
```

Otherwise, if the file does not exist:

```
{ :ok, file } = File.read("sample.txt")
# => ** (MatchError) no match of right hand side value: {:error, :enoent}

{ :error, msg } = File.read("sample.txt")
# => {:error, :enoent}
```

Pattern matching anonymous functions

```
fizzbuzz = fn
  (0, 0, _) -> "FizzBuzz"
  (0, _, _) -> "Fizz"
  (_, 0, _) -> "Buzz"
  (_, _, x) -> x
end

my_function = fn(n) ->
  fizzbuzz.(rem(n, 3), rem(n, 5), n)
end
```

Read Pattern matching online: https://riptutorial.com/elixir/topic/1602/pattern-matching

Chapter 33: Polymorphism in Elixir

Introduction

Polymorphism is the provision of a single interface to entities of different types. Basically, it allows different data types respond to the same function. So, the same function shapes for different data types to accomplish the same behavior. Elixir language has `protocols` to implement polymorphism with a clean way.

Remarks

If you want to cover all data types you can define an implementation for `Any` data type. Lastly, if you have time, check the source code of Enum and String.Char, which are good examples of polymorphism in core Elixir.

Examples

Polymorphism with Protocols

Let's implement a basic protocol that converts Kelvin and Fahrenheit temperatures to Celsius.

```
defmodule Kelvin do
  defstruct name: "Kelvin", symbol: "K", degree: 0
end

defmodule Fahrenheit do
  defstruct name: "Fahrenheit", symbol: "°F", degree: 0
end

defmodule Celsius do
  defstruct name: "Celsius", symbol: "°C", degree: 0
end

defprotocol Temperature do
  @doc """
  Convert Kelvin and Fahrenheit to Celsius degree
  """
  def to_celsius(degree)
end

defimpl Temperature, for: Kelvin do
  @doc """
  Deduct 273.15
  """
  def to_celsius(kelvin) do
    celsius_degree = kelvin.degree - 273.15
    %Celsius{degree: celsius_degree}
  end
end

defimpl Temperature, for: Fahrenheit do
```

```
  @doc """
  Deduct 32, then multiply by 5, then divide by 9
  """
  def to_celsius(fahrenheit) do
    celsius_degree = (fahrenheit.degree - 32) * 5 / 9
    %Celsius{degree: celsius_degree}
  end
end
```

Now, we implemented our converters for the Kelvin and Fahrenheit types. Let's make some conversions:

```
iex> fahrenheit = %Fahrenheit{degree: 45}
%Fahrenheit{degree: 45, name: "Fahrenheit", symbol: "°F"}
iex> celsius = Temperature.to_celsius(fahrenheit)
%Celsius{degree: 7.22, name: "Celsius", symbol: "°C"}
iex> kelvin = %Kelvin{degree: 300}
%Kelvin{degree: 300, name: "Kelvin", symbol: "K"}
iex> celsius = Temperature.to_celsius(kelvin)
%Celsius{degree: 26.85, name: "Celsius", symbol: "°C"}
```

Let's try to convert any other data type which has no implementation for `to_celsius` function:

```
iex> Temperature.to_celsius(%{degree: 12})
** (Protocol.UndefinedError) protocol Temperature not implemented for %{degree: 12}
    iex:11: Temperature.impl_for!/1
    iex:15: Temperature.to_celsius/1
```

Read Polymorphism in Elixir online: https://riptutorial.com/elixir/topic/9519/polymorphism-in-elixir

Chapter 34: Processes

Examples

Spawning a Simple Process

In the following example, the `greet` function inside `Greeter` module is run in a separate process:

```
defmodule Greeter do
    def greet do
        IO.puts "Hello programmer!"
    end
end

iex> spawn(Greeter, :greet, [])
Hello
#PID<0.122.0>
```

Here `#PID<0.122.0>` is the *process identifier* for the spawned process.

Sending and Receiving Messages

```
defmodule Processes do
    def receiver do
        receive do
            {:ok, val} ->
                IO.puts "Received Value: #{val}"
            _ ->
                IO.puts "Received something else"
        end
    end
end
```

```
iex(1)> pid = spawn(Processes, :receiver, [])
#PID<0.84.0>
iex(2)> send pid, {:ok, 10}
Received Value: 10
{:ok, 10}
```

Recursion and Receive

Recursion can be used to receive multiple messages

```
defmodule Processes do
    def receiver do
        receive do
            {:ok, val} ->
                IO.puts "Received Value: #{val}"
            _ ->
                IO.puts "Received something else"
        end
```

```
        receiver
    end
end
```

```
iex(1)> pid = spawn Processes, :receiver, []
#PID<0.95.0>
iex(2)> send pid, {:ok, 10}
Received Value: 10
{:ok, 10}
iex(3)> send pid, {:ok, 42}
{:ok, 42}
Received Value: 42
iex(4)> send pid, :random
:random
Received something else
```

Elixir will use a tail-call recursion optimisation as long as the function call is the last thing that happens in the function as it is in the example.

Read Processes online: https://riptutorial.com/elixir/topic/3173/processes

Chapter 35: Protocols

Remarks

A note on structs

Instead of sharing protocol implementation with maps, structs require their own protocol implementation.

Examples

Introduction

Protocols enable polymorphism in Elixir. Define protocols with `defprotocol`:

```
defprotocol Log do
  def log(value, opts)
end
```

Implement a protocol with `defimpl`:

```
require Logger
# User and Post are custom structs

defimpl Log, for: User do
  def log(user, _opts) do
    Logger.info "User: #{user.name}, #{user.age}"
  end
end

defimpl Log, for: Post do
  def log(user, _opts) do
    Logger.info "Post: #{post.title}, #{post.category}"
  end
end
```

With the above implementations, we can do:

```
iex> Log.log(%User{name: "Yos", age: 23})
22:53:11.604 [info]  User: Yos, 23
iex> Log.log(%Post{title: "Protocols", category: "Protocols"})
22:53:43.604 [info]  Post: Protocols, Protocols
```

Protocols let you dispatch to any data type, so long as it implements the protocol. This includes some built-in types such as `Atom`, `BitString`, `Tuples`, and others.

Read Protocols online: https://riptutorial.com/elixir/topic/3487/protocols

Chapter 36: Sigils

Examples

Build a list of strings

```
iex> ~w(a b c)
["a", "b", "c"]
```

Build a list of atoms

```
iex> ~w(a b c)a
[:a, :b, :c]
```

Custom sigils

Custom sigils can be made by creating a method `sigil_X` where X is the letter you want to use (this can only be a single letter).

```
defmodule Sigils do
  def sigil_j(string, options) do
    # Split on the letter p, or do something more useful
    String.split string, "p"
  end
  # Use this sigil in this module, or import it to use it elsewhere
end
```

The `options` argument is a binary of the arguments given at the end of the sigil, for example:

```
~j/foople/abc # string is "foople", options are 'abc'
# ["foo", "le"]
```

Read Sigils online: https://riptutorial.com/elixir/topic/2204/sigils

Chapter 37: State Handling in Elixir

Examples

Managing a piece of state with an Agent

The simplest way to wrap and access a piece of state is `Agent`. The module allows one to spawn a process that keeps an arbitrary data structure and allows one to send messages to read and update that structure. Thanks to this the access to the structure is automatically serialized, as the process only handles one message at a time.

```
iex(1)> {:ok, pid} = Agent.start_link(fn -> :initial_value end)
{:ok, #PID<0.62.0>}
iex(2)> Agent.get(pid, &(&1))
:initial_value
iex(3)> Agent.update(pid, fn(value) -> {value, :more_data} end)
:ok
iex(4)> Agent.get(pid, &(&1))
{:initial_value, :more_data}
```

Read State Handling in Elixir online: https://riptutorial.com/elixir/topic/6596/state-handling-in-elixir

Chapter 38: Stream

Remarks

Streams are composable, lazy enumerables.

Due to their laziness, streams are useful when working with large (or even infinite) collections. When chaining many operations with `Enum`, intermediate lists are created, while `Stream` creates a recipe of computations that are executed at a later moment.

Examples

Chaining multiple operations

`Stream` is especially useful when you want to run multiple operations on a collection. This is because `Stream` is lazy and only does one iteration (whereas `Enum` would do multiple iterations, for example).

```
numbers = 1..100
|> Stream.map(fn(x) -> x * 2 end)
|> Stream.filter(fn(x) -> rem(x, 2) == 0 end)
|> Stream.take_every(3)
|> Enum.to_list

[2, 8, 14, 20, 26, 32, 38, 44, 50, 56, 62, 68, 74, 80, 86, 92, 98, 104, 110,
 116, 122, 128, 134, 140, 146, 152, 158, 164, 170, 176, 182, 188, 194, 200]
```

Here, we chained 3 operations (`map`, `filter` and `take_every`), but the final iteration was only done after `Enum.to_list` was called.

What `Stream` does internally, is that it waits until actual evaluation is required. Before that, it creates a list of all the functions, but once evaluation is needed, it does goes through the collection once, running all the functions on every item. This makes it more efficient than `Enum`, which in this case would do 3 iterations, for example.

Read Stream online: https://riptutorial.com/elixir/topic/2553/stream

Chapter 39: Strings

Remarks

A `String` in Elixir is a `UTF-8` encoded binary.

Examples

Convert to string

Use Kernel.inspect to convert anything to string.

```
iex> Kernel.inspect(1)
"1"
iex> Kernel.inspect(4.2)
"4.2"
iex> Kernel.inspect %{pi: 3.14, name: "Yos"}
"%{pi: 3.14, name: \"Yos\"}"
```

Get a substring

```
iex> my_string = "Lorem ipsum dolor sit amet, consectetur adipiscing elit."
iex> String.slice my_string, 6..10
"ipsum"
```

Split a string

```
iex> String.split("Elixir, Antidote, Panacea", ",")
["Elixir", "Antidote", "Panacea"]
```

String Interpolation

```
iex(1)> name = "John"
"John"
iex(2)> greeting = "Hello, #{name}"
"Hello, John"
iex(3)> num = 15
15
iex(4)> results = "#{num} item(s) found."
"15 item(s) found."
```

Check if String contains Substring

```
iex(1)> String.contains? "elixir of life", "of"
true
iex(2)> String.contains? "elixir of life", ["life", "death"]
true
```

```
iex(3)> String.contains? "elixir of life", ["venus", "mercury"]
false
```

Join Strings

You can concatenate strings in Elixir using the `<>` operator:

```
"Hello" <> "World"                          # => "HelloWorld"
```

For a `List` of Strings, you can use `Enum.join/2`:

```
Enum.join(["A", "few", "words"], " ")       # => "A few words"
```

Read Strings online: https://riptutorial.com/elixir/topic/2618/strings

Chapter 40: Task

Syntax

- Task.async(fun)
- Task.await(task)

Parameters

Parameter	Details
fun	The function that should be executed in a separate process.
task	The task returned by `Task.async`.

Examples

Doing work in the background

```
task = Task.async(fn -> expensive_computation end)
do_something_else
result = Task.await(task)
```

Parallel processing

```
crawled_site = ["http://www.google.com", "http://www.stackoverflow.com"]
|> Enum.map(fn site -> Task.async(fn -> crawl(site) end) end)
|> Enum.map(&Task.await/1)
```

Read Task online: https://riptutorial.com/elixir/topic/7588/task

Chapter 41: Tips and Tricks

Introduction

Elixir Advanced tips and tricks which save our time while coding.

Examples

Creating Custom Sigils and Documenting

Each x sigil call respective sigil_x definition

Defining Custom Sigils

```
defmodule MySigils do
  #returns the downcasing string if option l is given then returns the list of downcase
letters
  def sigil_l(string,[]), do: String.Casing.downcase(string)
  def sigil_l(string,[?l]), do: String.Casing.downcase(string) |> String.graphemes

  #returns the upcasing string if option l is given then returns the list of downcase letters
  def sigil_u(string,[]), do: String.Casing.upcase(string)
  def sigil_u(string,[?l]), do: String.Casing.upcase(string) |> String.graphemes
end
```

Multiple [OR]

This is just the other way of writing Multiple OR conditions. This is not the recommended approach because in regular approach when the condition evaluates to true, it stops executing the remaining conditions which save the time of evaluation, unlike this approach which evaluates all conditions first in the list. This is just bad but good for discoveries.

```
# Regular Approach
find = fn(x) when x>10 or x<5 or x==7 -> x end

# Our Hack
hell = fn(x) when true in [x>10,x<5,x==7] -> x end
```

iex Custom Configuration - iex Decoration

Copy the content into a file and save the file as .iex.exs in your ~ home directory and see the magic. You can also download the file HERE

```
# IEx.configure colors: [enabled: true]
# IEx.configure colors: [ eval_result: [ :cyan, :bright ] ]
IO.puts IO.ANSI.red_background() <> IO.ANSI.white() <> "  ❄❄❄ Good Luck with Elixir ❄❄❄ " <> IO.ANSI.reset
Application.put_env(:elixir, :ansi_enabled, true)
IEx.configure(
```

```
 colors: [
   eval_result: [:green, :bright] ,
   eval_error: [[:red,:bright,"Bug Bug ..!!"]],
   eval_info: [:yellow, :bright ],
 ],
 default_prompt: [
   "\e[G",    # ANSI CHA, move cursor to column 1
    :white,
    "I",
    :red,
    "❤" ,       # plain string
    :green,
    "%prefix",:white,"|",
     :blue,
     "%counter",
     :white,
     "|",
    :red,
    "▶" ,        # plain string
    :white,
    "▶▶"  ,       # plain string
      # ❤ ❤-»" ,  # plain string
    :reset
  ] |> IO.ANSI.format |> IO.chardata_to_string

)
```

Read Tips and Tricks online: https://riptutorial.com/elixir/topic/10623/tips-and-tricks

Credits

S. No	Chapters	Contributors
1	Getting started with Elixir Language	alejosocorro, Andrey Chernykh, Ben Bals, Community, cwc, Delameko, Douglas Correa, helcim, I Am Batman, JAlberto, koolkat, leifg, MattW., rap-2-h, Simone Carletti, Stephan Rodemeier, Vinicius Quaiato, Yedhu Krishnan, Zimm i48
2	Basic .gitignore for elixir program	Yos Riady
3	basic use of guard clauses	alxndr
4	BEAM	Yos Riady
5	Behaviours	Yos Riady
6	Better debugging with IO.inspect and labels	leifg
7	Built-in types	Andrey Chernykh, Arithmeticbird, Oskar, TreyE, Vinicius Quaiato
8	Conditionals	Andrey Chernykh, evuez, javanut13, Musfiqur Rahman, Paweł Obrok
9	Constants	ibgib
10	Data Structures	Sam Mercier, Simone Carletti, Stephan Rodemeier, Yos Riady
11	Debugging Tips	javanut13, Paweł Obrok, Pfitz, Philippe-Arnaud de MANGOU, sbs
12	Doctests	aholt, milmazz, Philippe-Arnaud de MANGOU, Yos Riady
13	Ecto	fgutierr, Philippe-Arnaud de MANGOU, toraritte
14	Erlang	4444, Yos Riady
15	ExDoc	milmazz, Yos Riady
16	ExUnit	Yos Riady
17	Functional	Dinesh Balasubramanian

	programming in Elixir	
18	Functions	Andrey Chernykh, cwc, Dair, Eiji, Filip Haglund, PatNowak, rainteller, Simone Carletti, Stephan Rodemeier, Yedhu Krishnan , Yos Riady
19	Getting help in IEx console	helcim
20	IEx Console Tips & Tricks	alxndr, Cifer, fahrradflucht, legoscia, mudasobwa, muttonlamb, PatNowak, Paweł Obrok, sbs, Sheharyar, Simone Carletti, Stephan Rodemeier, Uniaika, Vincent, Yos Riady
21	Installation	cwc, Douglas Correa, Eiji, JAlberto, MattW.
22	Join Strings	Agung Santoso
23	Lists	Ben Bals, Candy Gumdrop, emoragaf, PatNowak, Sheharyar, Yos Riady
24	Maps and Keyword Lists	Sam Mercier, Simone Carletti, Yos Riady
25	Metaprogramming	4444, Paweł Obrok
26	Mix	4444, helcim, rainteller, Slava.K, Yos Riady
27	Modules	Alex G, javanut13, Yos Riady
28	Nodes	Yos Riady
29	Operators	alxndr, Andrey Chernykh, Dair, Gazler, Mitkins, nirev, PatNowak
30	Optimization	Filip Haglund, legoscia
31	Pattern matching	Alex Anderson, Dair, Danny Rosenblatt, evuez, Gabriel C, gmile , Harrison Lucas, javanut13, Oskar, PatNowak, theIV, Thomas, Yedhu Krishnan
32	Polymorphism in Elixir	mustafaturan
33	Processes	Alex G, Yedhu Krishnan
34	Protocols	Yos Riady
35	Sigils	javanut13, Yos Riady
36	State Handling in Elixir	Paweł Obrok

37	Stream	Oskar
38	Strings	Alex G, Sheharyar, Yos Riady
39	Task	mario
40	Tips and Tricks	Ankanna