Elixir: Getting Started

Plataformatec

July 2014

1 Interactive Elixir

Welcome!

In this tutorial we are going to show you how to get started with Elixir. This chapter will cover installation and how to get started with the Interactive Elixir shell called IEx.

If you find any errors in the tutorial or on the website, please report a bug or send a pull request to our issue tracker. If you suspect it is a language bug, please let us know in the language issue tracker.

Let's get started!

1.1 Installing Erlang

The only prerequisite for Elixir is Erlang, version 17.0 or later, which can be easily installed with Precompiled packages. In case you want to install it directly from source, it can be found on the Erlang website or by following the excellent tutorial available in the Riak documentation.

For Windows developers, we recommend the precompiled packages. Those on a UNIX platform can probably get Erlang installed via one of the many package management tools.

> Note: Although many package management tools provide Erlang, Elixir requires Erlang 17 which has been released just recently. So have that in mind before picking your Erlang installation.

After Erlang is installed, you should be able to open up the command line (or command prompt) and check the Erlang version by typing `erl`. You will see some information as follows:

```
Erlang/OTP 17 (erts-6) [64-bit] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]
```

Notice that depending on how you installed Erlang, Erlang binaries won't be available in your PATH. Be sure to have Erlang binaries in your PATH, otherwise Elixir won't work!

After Erlang is up and running, it is time to install Elixir. You can do that via Distributions, Precompiled package or Compiling from Source.

1.2 Distributions

This tutorial requires Elixir v0.14 or later and it may be available in some distributions:

- Homebrew for Mac OS X
- Update your homebrew to latest with `brew update`
- Install Elixir: `brew install elixir`
- Fedora 17+ and Fedora Rawhide
- `sudo yum -y install elixir`
- Arch Linux (on AUR)
- `yaourt -S elixir`
- openSUSE (and SLES 11 SP3+)
- Add Erlang devel repo with `zypper ar -f obs://devel:languages:erlang/erlang`
- Install Elixir: `zypper in elixir`
- Gentoo
- `emerge --ask dev-lang/elixir`
- Chocolatey for Windows
- `cinst elixir`

If you don't use any of the distributions above, don't worry, we also provide a precompiled package!

1.3 Precompiled package

Elixir provides a precompiled package for every release. After downloading and unzip-ing the package, you are ready to run the `elixir` and `iex` commands from the `bin` directory. It is recommended that you also add Elixir's `bin` path to your PATH environment variable to ease development.

1.4 Compiling from source (Unix and MinGW)

You can download and compile Elixir in few steps. You can get the latest stable release here, unpack it and then run `make` inside the unpacked directory. After that, you are ready to run the `elixir` and `iex` commands from the `bin` directory. It is recommended that you add Elixir's `bin` path to your PATH environment variable to ease development:

```
$ export PATH="$PATH:/path/to/elixir/bin"
```

In case you are feeling a bit more adventurous, you can also compile from master:

```
$ git clone https://github.com/elixir-lang/elixir.git
$ cd elixir
$ make clean test
```

If the tests pass, you are ready to go. Otherwise, feel free to open an issue in the issues tracker on Github.

1.5 Interactive mode

When you install Elixir, you will have three new executables: `iex`, `elixir` and `elixirc`. If you compiled Elixir from source or are using a packaged version, you can find these inside the `bin` directory.

For now, let's start by running `iex` which stands for Interactive Elixir. In interactive mode, we can type any Elixir expression and get its result straight away. Let's warm up with some basic expressions:

```
teractive Elixir - press Ctrl+C to exit (type h() ENTER for help)

x> 40 + 2

x> "hello" <> " world"
ello world"
```

It seems we are ready to go! We will use the interactive shell quite a lot in the next chapters to get a bit more familiar with the language constructs and basic types, starting in the next chapter.

2 Basic types

In this chapter we will learn more about Elixir basic types: integers, floats, atoms, lists and strings. Some basic types are:

```
x> 1          # integer
x> 0x1F       # integer
x> 1.0        # float
x> :atom      # atom / symbol
x> "elixir"   # string
x> [1, 2, 3]  # list
x> {1, 2, 3}  # tuple
```

2.1 Basic arithmetic

Open up `iex` and type the following expressions:

```
x> 1 + 2

```

```
x> 5 * 5

x> 10 / 2
0
```

Notice that `10 / 2` returned a float `5.0` instead of an integer `5`. This is expected. In Elixir, the operator `/` always returns a float. If you want to do integer division or get the division remainder, you can invoke the `div` and `rem` functions:

```
x> div(10, 2)

x> div 10, 2

x> rem 10, 3

```

Notice that parentheses are not required in order to invoke a function.

Elixir also supports shortcut notations for entering binary, octal and hexadecimal numbers:

```
x> 0b1010

x> 0777
1
x> 0x1F

```

Float numbers require a dot followed by at least one digit and also support `e` for the exponent number:

```
x> 1.0
0
x> 1.0e-10
0e-10
```

Floats in Elixir are 64 bit double precision.

2.2 Booleans

Elixir supports `true` and `false` as booleans.

```
x> true
ue
x> true == false
lse
```

Elixir provides a bunch of predicate functions to check for a value type. For example, the `is_boolean/1` function can be used to check if a value is a boolean or not:

> Note: Functions in Elixir are identified by name and by number of arguments (i.e. arity). Therefore, `is_boolean/1` identifies a function named `is_boolean` that takes 1 argument. `is_boolean/2` identifies a different (nonexistent) function with the same name but different arity.

```
x> is_boolean(true)
ue
x> is_boolean(1)
lse
```

You can also use `is_integer/1`, `is_float/1` or `is_number/1` to check, respectively, if an argument is an integer, a float or either.

> Note: at any moment you can type `h` in the shell to print information on how to use the shell. The `h` helper can also be used to access documentation for any function. For example, typing `h is_integer/1` is going to print the documentation for the `is_integer/1` function. It also works with operators and other constructs (try `h ==/2`).

2.3 Atoms

Atoms are constants where their name is their own value. Some other languages call these symbols.

```
x> :hello
ello
x> :hello == :world
lse
```

The booleans `true` and `false` are, in fact, atoms:

```
x> true == :true
ue
```

```
x> is_atom(false)
ue
```

2.4 Strings

Strings in Elixir are inserted in between double quotes, and they are encoded in UTF-8:

```
x> "hell"
ell"
```

Elixir also supports string interpolation:

```
x> "hell #{:world}"
ell world"
```

Strings can have line breaks in them or introduce them using escape sequences:

```
x> "hello
.> world"
ello\nworld"
x> "hello\nworld"
ello\nworld"
```

You can print a string using the `IO.puts/1` function from the `IO` module:

```
x> IO.puts "hello\nworld"
llo
rld
k
```

Notice the `IO.puts/1` function returns the atom `:ok` as result after printing.

Strings in Elixir are represented internally by binaries which are sequences of bytes:

```
x> is_binary("hell")
ue
```

We can also get the number of bytes in a string:

```
x> byte_size("hell")

```

Notice the number of bytes in that string is 6, even though it has 5 characters. That's because the character "" takes 2 bytes to be represented in UTF-8. We can get the actual length of the string, based on the number of characters, by using the `String.length/1` function:

```
x> String.length("hell")

```

The String module contains a bunch of functions that operate on strings as defined in the Unicode standard:

```
x> String.upcase("hell")
ELL"
```

Keep in mind `single-quoted` and `double-quoted` strings are not equivalent in Elixir as they are represented by different types:

```
x> 'hell' == "hell"
lse
```

We will talk more about Unicode support and the difference between single and double-quoted strings in the "Binaries, strings and char lists" chapter.

2.5 Anonymous functions

Functions are delimited by the keywords `fn` and `end`:

```
x> add = fn a, b -> a + b end
unction<12.71889879/2 in :erl_eval.expr/5>
x> is_function(add)
ue
x> is_function(add, 2)
ue
x> is_function(add, 1)
lse
x> add.(1, 2)

```

Functions are "first class citizens" in Elixir meaning they can be passed as arguments to other functions just as integers and strings can. In the example, we have passed the function in the variable `add` to the `is_function/1` function which correctly returned `true`. We can also check the arity of the function by calling `is_function/2`.

Note a dot (`.`) in between the variable and parenthesis is required to invoke an anonymous function.

Anonymous functions are closures, and as such they can access variables that are in scope when the function is defined:

```
x> add_two = fn a -> add.(a, 2) end
unction<6.71889879/1 in :erl_eval.expr/5>
x> add_two.(2)

```

Keep in mind that a variable assigned inside a function does not affect its surrounding environment:

```
x> x = 42

x> (fn -> x = 0 end).()

x> x

```

2.6 (Linked) Lists

Elixir uses square brackets to specify a list of values. Values can be of any type:

```
x> [1, 2, true, 3]
, 2, true, 3]
x> length [1, 2, 3]

```

Two lists can be concatenated and subtracted using the `++/2` and `--/2` operators:

```
x> [1, 2, 3] ++ [4, 5, 6]
, 2, 3, 4, 5, 6]
x> [1, true, 2, false, 3, true] -- [true, false]
, 2, 3, true]
```

Throughout the tutorial, we will talk a lot about the head and tail of a list. The head is the first element of a list and the tail is the remainder of a list. They can be retrieved with the functions `hd/1` and `tl/1`. Let's assign a list to a variable and retrieve its head and tail:

```
x> list = [1,2,3]
x> hd(list)

x> tl(list)
, 3]
```

Getting the head or the tail of an empty list is an error:

```
x> hd []
 (ArgumentError) argument error
```

Oops!

2.7 Tuples

Elixir uses curly brackets to define tuples. As lists, tuples can hold any value:

```
x> {:ok, "hello"}
ok, "hello"}
x> tuple_size {:ok, "hello"}

```

Tuples store elements contiguously in memory. This means accessing a tuple element per index or getting the tuple size is a fast operation (indexes start from zero):

```
x> tuple = {:ok, "hello"}
ok, "hello"}
x> elem(tuple, 1)
ello"
x> tuple_size(tuple)

```

It is also possible to set an element at a particular index in a tuple with `put_elem/3`:

```
x> tuple = {:ok, "hello"}
ok, "hello"}
x> put_elem(tuple, 1, "world")
ok, "world"}
x> tuple
ok, "hello"}
```

Notice that `put_elem/3` returned a new tuple. The original tuple stored in the `tuple` variable was not modified because Elixir data types are immutable. By being immutable, Elixir code is easier to reason about as you never need to worry if a particular code is mutating your data structure in place.

By being immutable, Elixir also helps eliminate common cases where concurrent code has race conditions because two different entities are trying to change a data structure at the same time.

2.8 Lists or tuples?

What is the difference between lists and tuples?

Lists are stored in memory as linked lists. This means each element in a list points to the next element, and then to the next element, until it reaches the end of a list. We call each of those pairs in a list a **cons cell**:

```
x> list = [1|[2|[3|[]]]]
, 2, 3]
```

This means accessing the length of a list is a linear operation: we need to traverse the whole list in order to figure out its size. Updating a list is fast as long as we are prepending elements:

```
x> [0] ++ list
, 1, 2, 3]
x> list ++ [4]
, 2, 3, 4]
```

The first operation is fast because we are simply adding a new cons that points to the remaining of `list`. The second one is slow because we need to rebuild the whole list and add a new element to the end.

Tuples, on the other hand, are stored contiguously in memory. This means getting the tuple size or accessing an element by index is fast. However, updating or adding elements to tuples is expensive because it requires copying the whole tuple in memory.

Those performance characteristics dictate the usage of those data structures. One very common use case for tuples is to use them to return extra information from a function. For example, `File.read/1` is a function that can be used to read file contents and it returns tuples:

```
x> File.read("path/to/existing/file")
ok, "... contents ..."}
x> File.read("path/to/unknown/file")
error, :enoent}
```

If the path given to `File.read/1` exists, it returns a tuple with the atom `:ok` as first element and the file contents as second. Otherwise, it returns a tuple with `:error` and the error reason.

Most of the time, Elixir is going to guide you to do the right thing. For example, there is a `elem/2` function to access a tuple item but there is no built-in equivalent for lists:

```
x> tuple = {:ok, "hello"}
ok, "hello"}
x> elem(tuple, 1)
ello"
```

When "counting" the number of elements in a data structure, Elixir also abides by a simple rule: the function should be named `size` if the operation is in constant time (i.e. the value is pre-calculated) or `length` if the operation requires explicit counting.

For example, we have used 4 counting functions so far: `byte_size/1` (for the number of bytes in a string), `tuple_size/1` (for the tuple size), `length/1` (for the list length) and `String.length/1` (for the number of characters in a string). That said, we use `byte_size` to get the number of bytes in a string, which is cheap, but retrieving the number of unicode characters uses `String.length`, since the whole string needs to be iterated.

Elixir also provides `Port`, `Reference` and `PID` as data types (usually used in process communication), and we will take a quick look at them when talking about processes. For now, let's take a look at some of the basic operators that go with our basic types.

3 Basic operators

In the previous chapter, we saw Elixir provides `+`, `-`, `*`, `/` as arithmetic operators, plus the functions `div/2` and `rem/2` for integer division and remainder.

Elixir also provides `++` and `--` to manipulate lists:

```
x> [1,2,3] ++ [4,5,6]
,2,3,4,5,6]
x> [1,2,3] -- [2]
,3]
```

String concatenation is done with <>:

```
x> "foo" <> "bar"
oobar"
```

Elixir also provides three boolean operators: `or`, `and` and `not`. These operators are strict in the sense that they expect a boolean (`true` or `false`) as their first argument:

```
x> true and true
ue
x> false or is_atom(:example)
ue
```

Providing a non-boolean will raise an exception:

```
x> 1 and true
 (ArgumentError) argument error
```

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

```
x> false and error("This error will never be raised")
lse

x> true or error("This error will never be raised")
ue
```

> Note: If you are an Erlang developer, `and` and `or` in Elixir actually map to the `andalso` and `orelse` operators in Erlang.

Besides these boolean operators, Elixir also provides ||, `&&` and `!` which accept arguments of any type. For these operators, all values except `false` and `nil` will evaluate to true:

```
or
x> 1 || true

x> false || 11


and
x> nil && 13
l
x> true && 17


!
x> !true
lse
x> !1
lse
x> !nil
ue
```

As a rule of thumb, use `and`, `or` and `not` when you are expecting booleans. If any of the arguments are non-boolean, use `&&`, || and `!`.

Elixir also provides `==`, `!=`, `===`, `!==`, `<=`, `>=`, < and > as comparison operators:

```
x> 1 == 1
ue
x> 1 != 2
ue
x> 1 < 2
ue
```

The difference between `==` and `===` is that the latter is more strict when comparing integers and floats:

```
x> 1 == 1.0
ue
x> 1 === 1.0
lse
```

In Elixir, we can compare two different data types:

```
x> 1 < :atom
ue
```

The reason we can compare different data types is pragmatism. Sorting algorithms don't need to worry about different data types in order to sort. The overall sorting order is defined below:

```
number < atom < reference < functions < port < pid < tuple < maps < list < bitstring
```

You don't actually need to memorize this ordering, but it is important just to know an order exists.

Well, that is it for the introduction. In the next chapter, we are going to discuss some basic functions, data type conversions and a bit of control-flow.

4 Pattern matching

In this chapter, we will show how the `=` operator in Elixir is actually a match operator and how to use it to pattern match inside data structures. Finally, we will learn about the pin operator `^` used to access previously bound values.

4.1 The match operator

We have used the `=` operator a couple times to assign variables in Elixir.

```
x> x = 1

x> x

```

In Elixir, the `=` operator is actually called *the match operator*. Let's see why:

```
x> 1 = x

x> 2 = x
 (MatchError) no match of right hand side value: 1
```

Notice that `1 = x` is a valid expression, and it matched because both the left and right side are equal to 1. When the sides do not match, a `MatchError` is raised.

A variable can only be assigned on the left side of `=`:

```
x> 1 = unknown
 (RuntimeError) undefined function: unknown/0
‘‘

nce there is no variable ‘unknown‘ previously defined, Elixir imagined you were trying to ca

  Pattern matching

e match operator is not only used to match against simple values, but it is also useful for

‘iex
x> {a, b, c} = {:hello, "world", 42}
hello, "world", 42}
x> a
ello
x> b
orld"
```

A pattern match will error in case the sides can't match. This is, for example, the case when the tuples have different sizes:

```
x> {a, b, c} = {:hello, "world"}
 (MatchError) no match of right hand side value: {:hello, "world"}
```

And also when comparing different types:

```
x> {a, b, c} = [:hello, "world", "!"]
 (MatchError) no match of right hand side value: [:hello, "world", "!"]
```

More interestingly, we can match on specific values. The example below asserts that the left side will only match the right side when the right side is a tuple that starts with the atom `:ok`:

```
x> {:ok, result} = {:ok, 13}
ok, 13}
x> result


x> {:ok, result} = {:error, :oops}
 (MatchError) no match of right hand side value: {:error, :oops}
```

We can pattern match on lists:

```
x> [a, b, c] = [1, 2, 3]
, 2, 3]
x> a

```

A list also supports matching on its own head and tail:

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

x> tail
, 3]
```

Similar to the `hd/1` and `tl/1` functions, we can't match an empty list with a head and tail pattern:

```
x> [h|t] = []
 (MatchError) no match of right hand side value: []
```

The `[head | tail]` format is not only used on pattern matching but also for prepending items to a list:

```
x> list = [1, 2, 3]
, 2, 3]
x> [0|list]
, 1, 2, 3]
```

Pattern matching allows developers to easily destructure data types such as tuples and lists. As we will see in following chapters, it is one of the foundations of recursion in Elixir and applies to other types as well, like maps and binaries.

4.2 The pin operator

Variables in Elixir can be rebound:

```
x> x = 1

x> x = 2

```

The pin operator ^ can be used when there is no interest in rebinding a variable but rather in matching against its value prior to the match:

```
x> x = 1

x> ^x = 2
 (MatchError) no match of right hand side value: 2
x> {x, ^x} = {2, 1}
, 1}
x> x

```

Notice that if a variable is mentioned more than once in a pattern, all references should bind to the same pattern:

```
x> {x, x} = {1, 1}

x> {x, x} = {1, 2}
 (MatchError) no match of right hand side value: {1, 2}
```

In some cases, you don't care about a particular value in a pattern. It is a common practice to bind those values to the underscore, _. For example, if only the head of the list matters to us, we can assign the tail to underscore:

```
x> [h | _] = [1, 2, 3]
, 2, 3]
x> h

```

The variable _ is special in that it can never be read from. Trying to read from it gives an unbound variable error:

```
x> _
 (CompileError) iex:1: unbound variable _
```

Although pattern matching allows us to build powerful constructs, its usage is limited. For instance, you cannot make function calls on the left side of a match. The following example is invalid:

```
x> length([1,[2],3]) = 3
 (ErlangError) erlang error :illegal_pattern
```

This finishes our introduction to pattern matching. As we will see in the next chapter, pattern matching is very common in many language constructs.

5 case, cond and if

In this chapter, we will learn about the `case`, `cond` and `if` control-flow structures.

5.1 case

`case` allows us to compare a value against many patterns until we find a matching one:

```
x> case {1, 2, 3} do
.>   {4, 5, 6} ->
.>     "This clause won't match"
.>   {1, x, 3} ->
.>     "This clause will match and bind x to 2 in this clause"
.>   _ ->
.>     "This clause would match any value"
.> end
```

If you want to pattern match against an existing variable, you need to use the ^ operator:

```
x> x = 1

x> case 10 do
.>   ^x -> "Won't match"
.>   _  -> "Will match"
.> end
```

Clauses also allow extra conditions to be specified via guards:

```
x> case {1, 2, 3} do
.>   {1, x, 3} when x > 0 ->
.>     "Will match"
.>   _ ->
.>     "Won't match"
.> end
```

The first clause above will only match when `x` is positive.

5.2 Expressions in guard clauses.

The Erlang VM only allows a limited set of expressions in guards:

- comparison operators (`==`, `!=`, `===`, `!==`, >, <, <=, >=)
- boolean operators (`and`, `or`) and negation operators (`not`, `!`)
- arithmetic operators (`+`, `-`, `*`, `/`)
- <> and `++` as long as the left side is a literal
- the `in` operator
- all the following type check functions:
  - `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`
- plus these functions:
  - `abs(number)`
  - `bit_size(bitstring)`
  - `byte_size(bitstring)`
  - `div(integer, integer)`
  - `elem(tuple, n)`
  - `hd(list)`
  - `length(list)`
  - `map_size(map)`
  - `node()`
  - `node(pid | ref | port)`
  - `rem(integer, integer)`
  - `round(number)`
  - `self()`

- `tl(list)`
- `trunc(number)`
- `tuple_size(tuple)`

Keep in mind errors in guards do not leak but simply make the guard fail:

```
x> hd(1)
 (ArgumentError) argument error
  :erlang.hd(1)
x> case 1 do
.>   x when hd(x) -> "Won't match"
.>   x -> "Got: #{x}"
.> end
ot 1"
```

If none of the clauses match, an error is raised:

```
x> case :ok do
.>   :error -> "Won't match"
.> end
 (CaseClauseError) no case clause matching: :ok
```

Note anonymous functions can also have multiple clauses and guards:

```
x> f = fn
.>   x, y when x > 0 -> x + y
.>   x, y -> x * y
.> end
unction<12.71889879/2 in :erl_eval.expr/5>
x> f.(1, 3)

x> f.(-1, 3)

```

The number of arguments in each anonymous function clause needs to be the same, otherwise an error is raised.

5.3 cond

`case` is useful when you need to match against different values. However, in many circumstances, we want to check different conditions and find the first one that evaluates to true. In such cases, one may use `cond`:

```
x> cond do
.>   2 + 2 == 5 ->
.>     "This will not be true"
.>   2 * 2 == 3 ->
.>     "Nor this"
.>   1 + 1 == 2 ->
.>     "But this will"
.> end
ut this will"
```

This is equivalent to `else if` clauses in many imperative languages (although used way less frequently here).

If none of the conditions return true, an error is raised. For this reason, it may be necessary to add a last condition equal to `true`, which will always match:

```
x> cond do
.>   2 + 2 == 5 ->
.>     "This is never true"
.>   2 * 2 == 3 ->
.>     "Nor this"
.>   true ->
.>     "This is always true (equivalent to else)"
.> end
```

Finally, note `cond` considers any value besides `nil` and `false` to be true:

```
x> cond do
.>   hd([1,2,3]) ->
.>     "1 is considered as true"
.> end
 is considered as true"
```

5.4 if and unless

Besides `case` and `cond`, Elixir also provides the macros `if/2` and `unless/2` which are useful when you need to check for just one condition:

```
x> if true do
.>   "This works!"
.> end
```

```
his works!"
x> unless true do
.>   "This will never be seen"
.> end
l
```

If the condition given to `if/2` returns `false` or `nil`, the body given between `do/end` is not executed and it simply returns `nil`. The opposite happens with `unless/2`.

They also support `else` blocks:

```
x> if nil do
.>   "This won't be seen"
.> else
.>   "This will"
.> end
his will"
```

> Note: An interesting note regarding `if/2` and `unless/2` is that they are implemented as macros in the language; they aren't special language constructs as they would be in many languages. You can check the documentation and the source of `if/2` in the `Kernel` module docs. The `Kernel` module is also where operators like `+/2` and functions like `is_function/2` are defined, all automatically imported and available in your code by default.

5.5 `do` blocks

At this point, we have learned four control structures: `case`, `cond`, `if` and `unless`, and they were all wrapped in `do/end` blocks. It happens we could also write `if` as follows:

```
x> if true, do: 1 + 2

```

In Elixir, `do/end` blocks are a convenience for passing a group of expressions to `do:`. These are equivalent:

```
x> if true do
.>   a = 1 + 2
.>   a + 10
.> end
```

```
x> if true, do: (
.>   a = 1 + 2
.>   a + 10
.> )
```

We say the second syntax is using **keyword lists**. We can pass `else` using this syntax:

```
x> if false, do: :this, else: :that
hat
```

It is important to keep one small detail in mind when using `do`/`end` blocks: they always bind to the farthest function call. For example, the following expression:

```
x> is_number if true do
.>  1 + 2
.> end
```

Would be parsed as:

```
x> is_number(if true) do
.>  1 + 2
.> end
```

Which leads to an undefined function error as Elixir attempts to invoke `is_number/2`. Adding explicit parentheses is enough to resolve the ambiguity:

```
x> is_number(if true do
.>  1 + 2
.> end)
ue
```

Keyword lists play an important role in the language and are quite common in many functions and macros. We will explore them a bit more in a future chapter. Now it is time to talk about "Binaries, strings and char lists".

6 Binaries, strings and char lists

In "Basic Types", we learned about strings and used the `is_binary/1` function for checks:

```
x> string = "hello"
ello"
x> is_binary string
ue
```

In this chapter, we will understand what binaries are, how they associate with strings, and what a single-quoted value, `'like this'`, means in Elixir.

6.1 UTF-8 and Unicode

A string is a UTF-8 encoded binary. In order to understand exactly what we mean by that, we need to understand the difference between bytes and code points.

The Unicode standard assigns code points to many of the characters we know. For example, the letter `a` has code point `97` while the letter has code point `322`. When writing the string `"heo"` to disk, we need to convert this code point to bytes. If we adopted a rule that said one byte represents one code point, we wouldn't be able to write `"heo"`, because it uses the code point `322` for , and one byte can only represent a number from `0` to `255`. But of course, given you can actually read `"heo"` on your screen, it must be represented *somehow*. That's where encodings come in.

When representing code points in bytes, we need to encode them somehow. Elixir chose the UTF-8 encoding as its main and default encoding. When we say a string is a UTF-8 encoded binary, we mean a string is a bunch of bytes organized in a way to represent certain code points, as specified by the UTF-8 encoding.

Since we have code points like assigned with the number `322`, we actually need more than one byte to represent it. That's why we see a difference when we calculate the `byte_size/1` of a string compared to its `String.length/1`:

```
x> string = "heo"
eo"
x> byte_size string

x> String.length string

```

UTF-8 requires one byte to represent the code points `h`, `e` and `o`, but two bytes to represent . In Elixir, you can get a code point's value by using `?`:

```
x> ?a

x> ?
2
```

You can also use the functions in the `String` module to split a string in its code points:

```
x> String.codepoints("heo")
h", "e", "", "", "o"]
```

You will see that Elixir has excellent support for working with strings. It also supports many of the Unicode operations. In fact, Elixir passes all the tests showcased in the "The string type is broken" article.

However, strings are just part of the story. If a string is a binary, and we have used the `is_binary/1` function, Elixir must have an underlying type empowering strings. And it does. Let's talk about binaries!

6.2 Binaries (and bitstrings)

In Elixir, you can define a binary using <<>>:

```
x> <<0, 1, 2, 3>>
0, 1, 2, 3>>
x> byte_size <<0, 1, 2, 3>>

```

A binary is just a sequence of bytes. Of course, those bytes can be organized in any way, even in a sequence that does not make them a valid string:

```
x> String.valid?(<<239, 191, 191>>)
lse
```

The string concatenation operation is actually a binary concatenation operator:

```
x> <<0, 1>> <> <<2, 3>>
0, 1, 2, 3>>
```

A common trick in Elixir is to concatenate the null byte <<0>> to a string to see its inner binary representation:

```
x> "heo" <> <<0>>
104, 101, 197, 130, 197, 130, 111, 0>>
```

Each number given to a binary is meant to represent a byte and therefore must go up to 255. Binaries allow modifiers to be given to store numbers bigger than 255 or to convert a code point to its utf8 representation:

```
x> <<255>>
255>>
x> <<256>> # truncated
0>>
x> <<256 :: size(16)>> # use 16 bits (2 bytes) to store the number
1, 0>>
x> <<256 :: utf8>> # the number is a code point
"
x> <<256 :: utf8, 0>>
196, 128, 0>>
```

If a byte has 8 bits, what happens if we pass a size of 1 bit?

```
x> <<1 :: size(1)>>
1 :: size(1)>>
x> <<2 :: size(1)>> # truncated
0>>
x> is_binary(<< 1 :: size(1)>>)
lse
x> is_bitstring(<< 1 :: size(1)>>)
ue
x> bit_size(<< 1 :: size(1)>>)

```

The value is no longer a binary, but a bitstring – just a bunch of bits! So a binary is a bitstring where the number of bits is divisible by 8!

We can also pattern match on binaries / bitstrings:

```
x> <<0, 1, x>> = <<0, 1, 2>>
0, 1, 2>>
x> x

x> <<0, 1, x>> = <<0, 1, 2, 3>>
 (MatchError) no match of right hand side value: <<0, 1, 2, 3>>
```

Note each entry in the binary is expected to match exactly 8 bits. However, we can match on the rest of the binary modifier:

```
x> <<0, 1, x :: binary>> = <<0, 1, 2, 3>>
0, 1, 2, 3>>
x> x
2, 3>>
```

The pattern above only works if the binary is at the end of <<>>. Similar results can be retrieved with the string concatenation operator <>:

```
x> "he" <> rest = "hello"
ello"
x> rest
lo"
```

This finishes our tour of bitstrings, binaries and strings. A string is a UTF-8 encoded binary, and a binary is a bitstring where the number of bits is divisible by 8. Although this shows the flexibility Elixir provides to work with bits and bytes, 99% of the time you will be working with binaries and using the `is_binary/1` and `byte_size/1` functions.

6.3 Char lists

A char list is nothing more than a list of characters:

```
x> 'heo'
04, 101, 322, 322, 111]
x> is_list 'heo'
ue
x> 'hello'
ello'
```

You can see that, instead of containing bytes, a char list contains the code points of the characters in between single-quotes (note that iex will only output code points if any of the chars is outside the ASCII range). So while double-quotes represent a string (i.e. a binary), single-quotes represents a char list (i.e. a list).

In practice, char lists are used mostly when interfacing with Erlang, in particular old libraries that do not accept binaries as arguments. You can convert a char list to a string and back by using the `to_string/1` and `to_char_list/1` functions:

```
x> to_char_list "heo"
04, 101, 322, 322, 111]
x> to_string 'heo'
eo"
x> to_string :hello
ello"
x> to_string 1
"
```

Note that those functions are polymorphic. They not only convert char lists to strings, but also integers to strings, atoms to strings, and so on.

With binaries, strings, and char lists out of the way, it is time to talk about key-value data structures.

7 Keywords, maps and dicts

So far we haven't discussed any associative data structures, i.e. data structures that are able to associate a certain value (or multiple values) to a key. Different languages call these different names like dictionaries, hashes, associative arrays, maps, etc.

In Elixir, we have two main associative data structures: keyword lists and maps. It's time to learn more about them!

7.1 Keyword lists

In many functional programming languages, it is common to use a list of 2-item tuples as the representation of an associative data structure. In Elixir, when we have a list of tuples and the first item of the tuple (i.e. the key) is an atom, we call it a keyword list:

```
x> list = [{:a, 1}, {:b, 2}]
: 1, b: 2]
x> list == [a: 1, b: 2]
ue
x> list[:a]

```

As you can see above, Elixir supports a special syntax for defining such lists, and underneath they just map to a list of tuples. Since they are simply lists, all operations available to lists, including their performance characteristics, also apply to keyword lists.

For example, we can use `++` to add new values to a keyword list:

```
x> list ++ [c: 3]
: 1, b: 2, c: 3]
x> [a: 0] ++ list
: 0, a: 1, b: 2]
```

Note that values added to the front are the ones fetched on lookup:

```
x> new_list = [a: 0] ++ list
: 0, a: 1, b: 2]
x> new_list[:a]

```

Keyword lists are important because they have two special characteristics:

- They keep the keys ordered as given by the developer.
- They allow a key to be given more than once.

For example, the Ecto library makes use of both features to provide an elegant DSL for writing database queries:

```
ery = from w in Weather,
    where: w.prcp > 0,
    where: w.temp < 20,
   select: w
```

Those features are what prompted keyword lists to be the default mechanism for passing options to functions in Elixir. In chapter 5, when we discussed the `if/2` macro, we mentioned the following syntax is supported:

```
x> if false, do: :this, else: :that
hat
```

The `do:` and `else:` pairs are keyword lists! In fact, the call above is equivalent to:

```
x> if(false, [do: :this, else: :that])
hat
```

In general, when the keyword list is the last argument of a function, the square brackets are optional.

In order to manipulate keyword lists, Elixir provides the `Keyword` module. Remember though keyword lists are simply lists, and as such they provide the same linear performance characteristics as lists. The longer the list, the longer it will take to find a key, to count the number of items, and so on. For this reason, keyword lists are used in Elixir mainly as options. If you need to store many items or guarantee one-key associates with at maximum one-value, you should use maps instead.

Note we can also pattern match on keyword lists:

```
x> [a: a] = [a: 1]
: 1]
x> a

x> [a: a] = [a: 1, b: 2]
 (MatchError) no match of right hand side value: [a: 1, b: 2]
x> [b: b, a: a] = [a: 1, b: 2]
 (MatchError) no match of right hand side value: [a: 1, b: 2]
```

However this is rarely done in practice since pattern matching on lists require the number of items and their order to match.

7.2 Maps

Whenever you need a key-value store, maps are the "go to" data structure in Elixir. A map is created using the `%{}` syntax:

```
x> map = %{:a => 1, 2 => :b}
2 => :b, :a => 1}
x> map[:a]

x> map[2]

```

Compared to keyword lists, we can already see two differences:

- Maps allow any value as key.
- Maps' keys do not follow any ordering.

If you pass duplicate keys when creating a map, the last one wins:

```
x> %{1 => 1, 1 => 2}
1 => 2}
```

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

```
x> map = %{a: 1, b: 2}
a: 1, b: 2}
```

In contrast to keyword lists, maps are very useful with pattern matching:

```
x> %{} = %{:a => 1, 2 => :b}
:a => 1, 2 => :b}
x> %{:a => a} = %{:a => 1, 2 => :b}
:a => 1, 2 => :b}
x> a

x> %{:c => c} = %{:a => 1, 2 => :b}
 (MatchError) no match of right hand side value: %{2 => :b, :a => 1}
```

As shown above, a map matches as long as the given keys exist in the given map. Therefore, an empty map matches all maps.

One interesting property about maps is that they provide a particular syntax for updating and accessing atom keys:

```
x> map = %{:a => 1, 2 => :b}
:a => 1, 2 => :b}
x> map.a

x> %{map | :a => 2}
:a => 2, 2 => :b}
x> %{map | :c => 3}
 (ArgumentError) argument error
```

Both access and update syntaxes above require the given keys to exist. For example, the last line failed because there is no `:c` in the map. This is very useful when you are working with maps where you only expect certain keys to exist.

In future chapters, we will also learn about structs, which provide compile-time guarantees and the foundation for polymorphism in Elixir. Structs are built on top of maps where the update guarantees above are proven to be quite useful.

Manipulating maps is done via the `Map` module, it provides a very similar API to the `Keyword` module. This is because both modules implement the `Dict` behaviour.

> Note: Maps were recently introduced into the Erlang VM with EEP 43. Erlang 17 provides a partial implementation of the EEP, where only "small maps" are supported. This means maps have good performance characteristics only when storing at maximum a couple of dozens keys. To fill in this gap, Elixir also provides the `HashDict` module which uses a hashing algorithm to provide a dictionary that supports hundreds of thousands keys with good performance.

7.3 Dicts

In Elixir, both keyword lists and maps are called dictionaries. In other words, a dictionary is like an interface (we call them behaviours in Elixir) and both keyword lists and maps modules implement this interface.

This interface is defined in the the `Dict` module module which also provides an API that delegates to the underlying implementations:

```
x> keyword = []

x> map = %{}
}
x> Dict.put(keyword, :a, 1)
: 1]
x> Dict.put(map, :a, 1)
a: 1}
```

The `Dict` module allows any developer to implement their own variation of `Dict`, with specific characteristics, and hook into existing Elixir code. The `Dict` module also provides functions that are meant to work across dictionaries. For example, `Dict.equal?/2` can compare two dictionaries of any kind.

That said, you may be wondering, which of `Keyword`, `Map` or `Dict` modules should you use in your code? The answer is: it depends.

If your code is expecting a keyword as an argument, explicitly use the `Keyword` module. If you want to manipulate a map, use the `Map` module. However, if your API is meant to work with any dictionary, use the `Dict` module (and make sure to write tests that pass different dict implementations as arguments).

This concludes our introduction to associative data structures in Elixir. You will find out that given keyword lists and maps, you will always have the right tool to tackle problems that require associative data structures in Elixir.

8 Modules

In Elixir we group several functions into modules. We've already used many different modules in the previous chapters like the `String` module:

```
x> String.length "hello"

```

In order to create our own modules in Elixir, we use the `defmodule` macro. We use the `def` macro to define functions in that module:

```
x> defmodule Math do
.>   def sum(a, b) do
.>     a + b
.>   end
.> end

x> Math.sum(1, 2)

```

In the following sections, our examples are going to get a bit more complex, and it can be tricky to type them all in the shell. It's about time for us to learn how to compile Elixir code and also how to run Elixir scripts.

8.1 Compilation

Most of the time it is convenient to write modules into files so they can be compiled and reused. Let's assume we have a file named `math.ex` with the following contents:

```
fmodule Math do
def sum(a, b) do
  a + b
end
d
```

This file can be compiled using `elixirc`:

```
elixirc math.ex
```

This will generate a file named `Elixir.Math.beam` containing the bytecode for the defined module. If we start `iex` again, our module definition will be available (provided that `iex` is started in the same directory the bytecode file is in):

```
x> Math.sum(1, 2)

```

Elixir projects are usually organized into three directories:

- ebin - contains the compiled bytecode
- lib - contains elixir code (usually `.ex` files)
- test - contains tests (usually `.exs` files)

When working on actual projects, the build tool called `mix` will be responsible for compiling and setting up the proper paths for you. For learning purposes, Elixir also supports a scripted mode which is more flexible and does not generate any compiled artifacts.

8.2 Scripted mode

In addition to the Elixir file extension `.ex`, Elixir also supports `.exs` files for scripting. Elixir treats both files exactly the same way, the only difference is in intention. `.ex` files are meant to be compiled while `.exs` files are used for scripting, without the need for compilation. For instance, we can create a file called `math.exs`:

```
1 fmodule Math do
2 def sum(a, b) do
3   a + b
4 end
5 d
6
7 .puts Math.sum(1, 2)
```

And execute it as:

```
elixir math.exs
```

The file will be compiled in memory and executed, printing "3" as the result. No bytecode file will be created. In the following examples, we recommend you write your code into script files and execute them as shown above.

8.3 Named functions

Inside a module, we can define functions with `def/2` and private functions with `defp/2`. A function defined with `def/2` can be invoked from other modules while a private function can only be invoked locally.

```
1 fmodule Math do
2 def sum(a, b) do
3   do_sum(a, b)
4 end
```

```
5
6  defp do_sum(a, b) do
7    a + b
8  end
9  d
10
11 th.sum(1, 2)     #=> 3
12 th.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

Function declarations also support guards and multiple clauses. If a function has several clauses, Elixir will try each clause until it finds one that matches. Here is an implementation of a function that checks if the given number is zero or not:

```
1  fmodule Math do
2  def zero?(0) do
3    true
4  end
5
6  def zero?(x) when is_number(x) do
7    false
8  end
9  d
10
11 th.zero?(0)   #=> true
12 th.zero?(1)   #=> false
13
14 th.zero?([1,2,3])
15 > ** (FunctionClauseError)
```

Giving an argument that does not match any of the clauses raises an error.

8.4 Function capturing

Throughout this tutorial, we have been using the notation `name/arity` to refer to functions. It happens that this notation can actually be used to retrieve a named function as a function type. Let's start `iex` and run the `math.exs` file defined above:

```
$ iex math.exs
```

```
1  x> Math.zero?(0)
2  ue
3  x> fun = &Math.zero?/1
```

```
ath.zero?/1
x> is_function fun
ue
x> fun.(0)
ue
‘‘

cal or imported functions, like ‘is_function/1‘, can be captured without the module:

‘iex
x> &is_function/1
erlang.is_function/1
x> (&is_function/1).(fun)
ue
```

Note the capture syntax can also be used as a shortcut for creating functions:

```
x> fun = &(&1 + 1)
unction<6.71889879/1 in :erl_eval.expr/5>
x> fun.(1)

```

The `&1` represents the first argument passed into the function. `&(&1+1)` above is exactly the same as `fn x -> x + 1 end`. The syntax above is useful for short function definitions. You can read more about the capture operator `&` in the `Kernel.SpecialForms` documentation.

8.5 Default arguments

Named functions in Elixir also support default arguments:

```
fmodule Concat do
def join(a, b, sep \\ " ") do
  a <> sep <> b
end
d

.puts Concat.join("Hello", "world")      #=> Hello world
.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

Any expression is allowed to serve as a default value, but it won't be evaluated during the function definition; it will simply be stored for later use. Every time the function is invoked and any of its default values have to be used, the expression for that default value will be evaluated:

```
fmodule DefaultTest do
def dowork(x \\ IO.puts "hello") do
  x
end
d
```

```
x> DefaultTest.dowork 123
3
x> DefaultTest.dowork
llo
k
```

If a function with default values has multiple clauses, it is recommended to create a function head (without an actual body), just for declaring defaults:

```
fmodule Concat do
def join(a, b \\ nil, sep \\ " ")

def join(a, b, _sep) when nil?(b) do
  a
end

def join(a, b, sep) do
  a <> sep <> b
end
d

.puts Concat.join("Hello", "world")       #=> Hello world
.puts Concat.join("Hello", "world", "_") #=> Hello_world
.puts Concat.join("Hello")                #=> Hello
```

When using default values, one must be careful to avoid overlapping function definitions. Consider the following example:

```
fmodule Concat do
def join(a, b) do
  IO.puts "***First join"
  a <> b
end

def join(a, b, sep \\ " ") do
```

```
  IO.puts "***Second join"
  a <> sep <> b
end
d
```

If we save the code above in a file named "concat.ex" and compile it, Elixir will emit the following warning:

```
concat.exs:7: this clause cannot match because a previous clause at line 2 always matches
```

The compiler is telling us that invoking the `join` function with two arguments will always choose the first definition of `join` whereas the second one will only be invoked when three arguments are passed:

```
$ iex concat.exs
```

```
x> Concat.join "Hello", "world"
*First join
elloworld"
```

```
x> Concat.join "Hello", "world", "_"
*Second join
ello_world"
```

This finishes our short introduction to modules. In the next chapters, we will learn how to use named functions for recursion, explore Elixir lexical directives that can be used for importing functions from other modules and discuss module attributes.

9 Recursion

Due to immutability, loops in Elixir (and in functional programming languages) are written differently from conventional imperative languages. For example, in an imperative language, one would write:

```
r(i = 0; i < array.length; i++) {
array[i] = array[i] * 2

```

In the example above, we are mutating the array and the helper variable `i`. That's not possible in Elixir. Instead, functional languages rely on recursion: a function is called recursively until a condition is reached that stops the recursive action from continuing. Consider the example below that prints a string an arbitrary amount of times:

```
fmodule Recursion do
def print_multiple_times(msg, n) when n <= 1 do
  IO.puts msg
end

def print_multiple_times(msg, n) do
  IO.puts msg
  print_multiple_times(msg, n - 1)
end
d

cursion.print_multiple_times("Hello!", 3)
Hello!
Hello!
Hello!
```

Similar to case, a function may have many clauses. A particular clause is executed when the arguments passed to the function match the clause's argument patterns and its guard evaluates to `true`.

Above when `print_multiple_times/2` is initially called, the argument `n` is equal to `3`.

The first clause has a guard which says use this definition if and only if `n` is less than or equal to `1`. Since this is not the case, Elixir proceeds to the next clause's definition.

The second definition matches the pattern and has no guard so it will be executed. It first prints our `msg` and then calls itself passing `n - 1` (`2`) as the second argument. Our `msg` is printed and `print_multiple_times/2` is called again this time with the second argument set to `1`.

Because `n` is now set to `1`, the guard to our first definition of `print_multiple_times/2` evaluates to true, and we execute this particular definition. The `msg` is printed, and there is nothing left to execute.

We defined `print_multiple_times/2` so that no matter what number is passed as the second argument it either triggers our first definition (known as a "base case") or it triggers our second definition which will ensure that we get exactly 1 step closer to our base case.

Let's now see how we can use the power of recursion to sum a list of numbers.

```
fmodule Math do
def sum_list([head|tail], accumulator) do
  sum_list(tail, head + accumulator)
end

def sum_list([], accumulator) do
  accumulator
```

```
end
d

th.sum_list([1, 2, 3], 0) #=> 6
```

We invoke `sum_list` with a list `[1,2,3]` and the initial value `0` as arguments. We will try each clause until we find one that matches according to the pattern matching rules. In this case, the list `[1,2,3]` matches against `[head|tail]` which assigns `head = 1` and `tail = [2,3]` while `accumulator` is set to `0`.

Then, we add the head of the list to the accumulator `head + accumulator` and call `sum_list` again, recursively, passing the tail of the list as its first argument. The tail will once again match `[head|tail]` until the list is empty, as seen below:

```
m_list [1, 2, 3], 0
m_list [2, 3], 1
m_list [3], 3
m_list [], 6
```

When the list is empty, it will match the final clause which returns the final result of `6`.

The process of taking a list and "reducing" it down to one value is known as a "reduce" algorithm and is central to functional programming.

What if we instead want to double all of the values in our list?

```
fmodule Math do
def double_each([head|tail]) do
  [head * 2| double_each(tail)]
end

def double_each([]) do
  []
end
d

th.double_each([1, 2, 3]) #=> [2, 4, 6]
```

Here we have used recursion to traverse a list doubling each element and returning a new list. The process of taking a list and "mapping" over it is known as a "map" algorithm.

Recursion and tail call optimization are an important part of Elixir and are commonly used to create loops. However, when programming Elixir you will rarely use recursion as above to manipulate lists.

The Enum module, which we are going to study in the next chapter, already provides many conveniences for working with lists. For instance, the examples above could be written as:

```
x> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc end)

x> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
, 4, 6]
```

Or, using the capture syntax:

```
x> Enum.reduce([1, 2, 3], 0, &+/2)

x> Enum.map([1, 2, 3], &(&1 * 2))
, 4, 6]
```

So let's take a deeper look at Enumerables and Streams.

10 Enumerables and Streams

10.1 Enumerables

Elixir provides the concept of enumerables and the Enum module to work with them. We have already learned two enumerables: lists and maps.

```
x> Enum.map([1, 2, 3], fn x -> x * 2 end)
, 4, 6]
x> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k * v end)
, 12]
```

The Enum module provides a huge range of functions to transform, sort, group, filter and retrieve items from enumerables. It is one of the modules developers use frequently in their Elixir code.

Elixir also provides ranges:

```
x> Enum.map(1..3, fn x -> x * 2 end)
, 4, 6]
x> Enum.reduce(1..3, 0, &+/2)

```

Since the Enum module was designed to work across different data types, its API is limited to functions that are useful across many data types. For specific operations, you may need to reach to modules specific to the data types. For example, if you want to insert an element at a given position in a list, you should use the `List.insert_at/3` function from the `List` module, as it would make little sense to insert a value into, for example, a range.

We say the functions in the `Enum` module are polymorphic because they can work with diverse data types. In particular, the functions in the `Enum` module can work with any data type that implements the `Enumerable` protocol. We are going to discuss Protocols in a later chapter, for now we are going to move on to a specific kind of enumerable called streams.

10.2 Eager vs Lazy

All the functions in the `Enum` module are eager. Many functions expect an enumerable and return a list back:

```
x> odd? = &(rem(&1, 2) != 0)
unction<6.80484245/1 in :erl_eval.expr/5>
x> Enum.filter(1..3, odd?)
, 3]
```

This means that when performing multiple operations with `Enum`, each operation is going to generate an intermediate list until we reach the result:

```
x> 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
00000000
```

The example above has a pipeline of operations. We start with a range and then multiply each element in the range by 3. This first operation will now create and return a list with `100_000` items. Then we keep all odd elements from the list, generating a new list, now with `50_000` items, and then we sum all entries.

As an alternative, Elixir provides the `Stream` module which supports lazy operations:

```
x> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
00000000
```

Instead of generating intermediate lists, streams create a series of computations that are invoked only when we pass it to the `Enum` module. Streams are useful when working with large, *possibly infinite*, collections.

10.3 Streams

Streams are lazy, composable enumerables.

They are lazy because, as shown in the example above, `1..100_000 |> Stream.map(&(&1 * 3))` returns a data type, an actual stream, that represents the `map` computation over the range `1..100_000`:

```
x> 1..100_000 |> Stream.map(&(&1 * 3))
tream<1..100_000, funs: [#Function<34.16982430/1 in Stream.map/2>]>
```

Furthermore, they are composable because we can pipe many stream operations:

```
x> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?)
tream<1..100_000, funs: [...]>
```

Many functions in the `Stream` module accept any enumerable as argument and return a stream as result. It also provides functions for creating streams, possibly infinite. For example, `Stream.cycle/1` can be used to create a stream that cycles a given enumerable infinitely. Be careful to not call a function like `Enum.map/2` on such streams, as they would cycle forever:

```
x> stream = Stream.cycle([1, 2, 3])
unction<15.16982430/2 in Stream.cycle/1>
x> Enum.take(stream, 10)
, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

On the other hand, `Stream.unfold/2` can be used to generate values from a given initial value:

```
x> stream = Stream.unfold("heo", &String.next_codepoint/1)
unction<15.16982430/2 in Stream.cycle/1>
x> Enum.take(stream, 3)
h", "e", ""]
```

Another interesting function is `Stream.resource/3` which can be used to wrap around resources, guaranteeing they are opened right before enumeration and closed afterwards, even in case of failures. For example, we can use it to stream a file:

```
x> stream = File.stream!("path/to/file")
unction<18.16982430/2 in Stream.resource/3>
x> Enum.take(stream, 10)
```

The example above will fetch the first 10 lines of the file you have selected. This means streams can be very useful for handling large files or even slow resources like network resources.

The amount of functions and functionality in `Enum` and `Stream` modules can be daunting at first but you will get familiar with them case by case. In particular, focus on the `Enum` module first and only move to `Stream` for the particular scenarios where laziness is required to either deal with slow resources or large, possibly infinite, collections.

Next we'll look at a feature central to Elixir, Processes, which allows us to write concurrent, parallel and distributed programs in an easy and understandable way.

11 Processes

In Elixir, all code runs inside processes. Processes are isolated from each other, run concurrent to one another and communicate via message passing. Processes are not only the basis for concurrency in Elixir, but they also provide the means for building distributed and fault-tolerant programs.

Elixir's processes should not be confused with operating system processes. Processes in Elixir are extremely lightweight in terms of memory and CPU (unlike threads in many other programming languages). Because of this, it is not uncommon to have dozens of thousands of processes running simultaneously.

In this chapter, we will learn about the basic constructs for spawning new processes, as well as sending and receiving messages between different processes.

11.1 spawn

The basic mechanism for spawning new processes is with the auto-imported `spawn/1` function:

```
x> spawn fn -> 1 + 2 end
ID<0.43.0>
```

`spawn/1` takes a function which it will execute in another process.

Notice `spawn/1` returns a PID (process identifier). At this point, the process you spawned is very likely dead. The spawned process will execute the given function and exit after the function is done:

```
x> pid = spawn fn -> 1 + 2 end
ID<0.44.0>
x> Process.alive?(pid)
lse
```

> Note: you will likely get different process identifiers than the ones we are getting in this guide.

We can retrieve the PID of the current process by calling `self/0`:

```
x> self()
ID<0.41.0>
x> Process.alive?(self())
ue
```

Processes get much more interesting when we are able to send and receive messages.

11.2 send and receive

We can send messages to a process with `send/2` and receive them with `receive/1`:

```
x> send self(), {:hello, "world"}
hello, "world"}
x> receive do
.>   {:hello, msg}  -> msg
.>   {:world, msg} -> "won't match"
.> end
orld"
```

When a message is sent to a process, the message is stored in the process mailbox. The `receive/2` block goes through the current process mailbox searching for a message that matches any of the given patterns. `receive/1` supports many clauses, like `case/2`, as well as guards in the clauses.

If there is no message in the mailbox matching any of the patterns, the current process will wait until a matching message arrives. A timeout can also be specified:

```
x> receive do
.>   {:hello, msg}  -> msg
.> after
.>   1_000 -> "nothing after 1s"
```

```
5 .> end
6 othing after 1s"
```

A timeout of 0 can be given when you already expect the message to be in the mailbox.

Let's put all together and send messages in between processes:

```
1 x> parent = self()
2 ID<0.41.0>
3 x> spawn fn -> send(parent, {:hello, self()}) end
4 ID<0.48.0>
5 x> receive do
6 .>   {:hello, pid} -> "Got hello from #{inspect pid}"
7 .> end
8 ot hello from #PID<0.48.0>"
```

While in the shell, you may find the helper `flush/0` quite useful. It flushes and prints all the messages in the mailbox.

```
1 x> send self(), :hello
2 ello
3 x> flush()
4 ello
5 k
```

11.3 Links

The most common form of spawning in Elixir is actually via `spawn_link/1`. Before we show an example with `spawn_link/1`, let's try to see what happens when a process fails:

```
1 x> spawn fn -> raise "oops" end
2 ID<0.58.0>
```

Well... nothing happened. That's because processes are isolated. If we want the failure in one process to propagate to another one, we should link them. This can be done with `spawn_link/1`:

```
1 x> spawn_link fn -> raise "oops" end
2 ID<0.60.0>
3  (EXIT from #PID<0.41.0>) an exception was raised:
```

```
** (RuntimeError) oops
    :erlang.apply/2
```

When a failure happens in the shell, the shell automatically traps the failure and shows it nicely formatted. In order to understand what would really happen in our code, let's use `spawn_link/1` inside a file and run it:

```
spawn.exs
awn_link fn -> raise "oops" end

ceive do
:hello -> "let's wait until the process fails"
d
```

This time the process failed and brought the parent process down as they are linked. Linking can also be done manually by calling `Process.link/2`. We recommend you to take a look at the `Process` module for other functionality provided by processes.

Process and links play an important role when building fault-tolerant systems. In Elixir applications, we often link our processes to supervisors which will detect when a process die and start a new process in its place. This is only possible because processes are isolated and don't share anything by default. And if processes are isolated, there is no way a failure in a process will crash or corrupt the state of another.

While other languages would require us to catch/handle exceptions, in Elixir we are actually fine with letting process fail because we expect supervisors to properly restart our systems. "Failing fast" is a common philosophy when writing Elixir software!

Before moving to the next chapter, let's see one of the most common use case for creating processes in Elixir.

11.4 State

We haven't talked about state so far in this guide. If you are building an application that requires state, for example, to keep your application configuration, or you need to parse a file and keep it in memory, where would you store it?

Processes are the most common answer to this question. We can write processes that loops infinitely, keeping a state, and sending and receiving messages. As an example, let's write a module that starts new processes that work as a key-value store in a file named `kv.exs`:

```
fmodule KV do
def start do
```

```
  {:ok, spawn_link(fn -> loop(%{}) end)}
end

defp loop(map) do
  receive do
    {:get, key, caller} ->
      send caller, Map.get(map, key)
      loop(map)
    {:put, key, value} ->
      loop(Map.put(map, key, value))
  end
end
d
```

Note that the `start` function basically spawns a new process that runs the `loop/1` function, starting with an empty map. The `loop/1` function then waits for messages and perform the appropriate action for each message. In case of a `:get` message, it sends a message back to the caller and calls `loop/1` again, to wait for a new message. While the `:put` message actually makes `loop/1` be invoked with a new version of the map, with the given `key` and `value` stored.

Let's give it a try by running `iex kv.exs`:

```
x> {:ok, pid} = KV.start
ID<0.62.0>
x> send pid, {:get, :hello, self()}
get, :hello, #PID<0.41.0>}
x> flush
l
```

At first, the process map has no keys, so sending a `:get` message and then flushing the current process inbox returns `nil`. Let's send a `:put` message and try it again:

```
x> send pid, {:put, :hello, :world}
ID<0.62.0>
x> send pid, {:get, :hello, self()}
get, :hello, #PID<0.41.0>}
x> flush
orld
```

Notice how the process is keeping a state and we can get and update this state by sending the process messages. In fact, any process that knows the `pid` above will be able to send it messages and manipulate the state.

It is also possible to register the pid, giving it a name, and allowing everyone that knows the name to send it messages:

```
x> Process.register(pid, :kv)
ue
x> send :kv, {:get, :hello, self()}
get, :hello, #PID<0.41.0>}
x> flush
orld
```

Using process around state and name registering are very common patterns in Elixir applications. However, most of the time, we won't implement those patterns manualy as above, but using one of the many of the abstractions that ships with Elixir. For example, Elixir provides agents which are simple abstractions around state:

```
x> {:ok, pid} = Agent.start_link(fn -> %{} end)
ok, #PID<0.72.0>}
x> Agent.update(pid, fn map -> Map.put(map, :hello, :world) end)
k
x> Agent.get(pid, fn map -> Map.get(map, :hello) end)
orld
```

A `:name` option could also be given to `Agent.start/2` and it would be automatically registered. Besides agents, Elixir provides an API for building generic servers (called GenServer), generic event managers and event handlers (called GenEvent), tasks and more, all powered by processes underneath. Those, along with supervision trees, will be explored with more detail in the Mix and OTP guide which will build a complete Elixir application from start to finish.

For now, let's move on and explore the world of I/O in Elixir.

12 IO

This chapter is a quick introduction to input/output mechanisms in Elixir and related modules, like `IO`, `File` and `Path`.

We had originally sketched this chapter to come much earlier in the getting started guide. However, we noticed the IO system provides a great opportunity to shed some light on some philosophies and curiosities of Elixir and the VM.

12.1 The IO module

The `IO` module in Elixir is the main mechanism for reading and writing to the standard io (`:stdio`), standard error (`:stderr`), files and other IO devices. Usage of the module is pretty straight-forward:

```
x> IO.puts "hello world"
ello world"
k
x> IO.gets "yes or no? "
s or no? yes
es\n"
```

By default, the functions in the IO module use the standard input and output. We can pass the `:stderr` as argument to write to the standard error device:

```
x> IO.puts :stderr, "hello world"
ello world"
k
```

12.2 The File module

The `File` module contains functions that allows us to open files as IO devices. By default, files are opened in binary mode, which requires developers to use the specific `IO.binread/2` and `IO.binwrite/2` functions from the `IO` module:

```
x> {:ok, file} = File.open "hello", [:write]
ok, #PID<0.47.0>}
x> IO.binwrite file, "world"
k
x> File.close file
k
x> File.read "hello"
ok, "world"}
```

A file can also be opened with `:utf8` encoding which allows the remaining functions in the `IO` module to be used:

```
x> {:ok, file} = File.open "another", [:write, :utf8]
ok, #PID<0.48.0>}
```

Besides functions for opening, reading and writing files, the `File` module has many functions that work on the file system. Those functions are named after their UNIX equivalents. For example, `File.rm/1` can be used to remove files, `File.mkdir/1` to create directories, `File.mkdir_p/1` creates directories guaranteeing their parents exists and there is even `File.cp_r/2` and `File.rm_rf/2` which copy and remove files and directories recursively.

You will also notice that functions in the `File` module have two variants, one with `!` (bang) in its name and others without. For example, when we read the "hello" file above, we have used the one without `!`. Let's try some new examples:

```
x> File.read "hello"
ok, "world"}
x> File.read! "hello"
orld"
x> File.read "unknown"
error, :enoent}
x> File.read! "unknown"
 (File.Error) could not read file unknown: no such file or directory
```

Notice that when the file does not exist, the version with `!` raises an error. That said, the version without `!` is preferred when you want to handle different outcomes with pattern matching. However, if you expect the file to be there, the bang variation is more useful as it raises a meaningful error message. That said, never write:

```
ok, body} = File.read(file)
```

Instead write:

```
se File.read(file) do
{:ok, body} -> # handle ok
{:error, r} -> # handle error
d
```

or

```
le.read!(file)
```

12.3 The Path module

The majority of the functions in the File module expects paths as arguments. Most commonly, those paths will be binaries and they can be manipulated with the `Path` module:

```
x> Path.join("foo", "bar")
oo/bar"
```

```
x> Path.expand("~/hello")
Users/jose/hello"
```

With this we have covered the main modules for doing IO and interacting with the file system. Next we will discuss some curiosities and advanced topics regarding IO. Those sections are not necessary to write Elixir code, so feel free to skip them, but they do provide an overview of how the IO system is implemented in the VM and other curiosities.

12.4 Processes and group leaders

You may have noticed that `File.open/2` returned a tuple containing a PID:

```
x> {:ok, file} = File.open "hello", [:write]
ok, #PID<0.47.0>}
```

That's because the IO module actually works with processes. When you say `IO.write(pid, binary)`, the IO module will send a message to the process with the desired operation. Let's see what happens if we use our own process:

```
x> pid = spawn fn ->
.>  receive do: (msg -> IO.inspect msg)
.> end
ID<0.57.0>
x> IO.write(pid, "hello")
io_request, #PID<0.41.0>, #PID<0.57.0>, {:put_chars, :unicode, "hello"}}
 (ErlangError) erlang error: :terminated
```

After `IO.write/2`, we can see the request sent by the IO module printed, which then fails since the IO module expected some kind of result that we did not supply.

The `StringIO` module provides an implementation of the IO device messages on top of a string:

```
x> {:ok, pid} = StringIO.open("hello")
ok, #PID<0.43.0>}
x> IO.read(pid, 2)
e"
```

By modelling IO devices with processes, the Erlang VM allows different nodes in the same network to exchange file processes to read/write files in between nodes. Of all IO devices, there is one that is special to each process, called group leader.

When you write to `:stdio`, you are actually sending a message to the group leader, which writes to STDIO file descriptor:

```
x> IO.puts :stdio, "hello"
llo
k
x> IO.puts Process.group_leader, "hello"
llo
k
```

The group leader can be configured per process and is used in different situations. For example, when executing code in a remote terminal, it guarantees messages in a remote node are redirected and printed in the terminal that triggered the request.

12.5 `iodata` and `chardata`

In all examples above, we have used binaries/strings when writing to files. In the chapter "Binaries, strings and char lists", we mentioned how strings are simply bytes while char lists are lists with code points.

The functions in `IO` and `File` also allow lists to be given as arguments. Not only that, they also allow a mixed list of lists, integers and binaries to be given:

```
x> IO.puts 'hello world'
llo world
k
x> IO.puts ['hello', ?\s, "world"]
llo world
k
```

However, this requires some attention. A list may represent either a bunch of bytes or a bunch of characters and which one to use depends on the encoding of the IO device. If the file is opened without encoding, the file is expected to be in raw mode, and the functions in the `IO` module starting with `bin*` must be used. Those functions expect an `iodata` as argument, i.e. it expects a list of integers representing bytes and binaries to be given.

On the other hand, `:stdio` and files opened with `:utf8` encoding work with the remaining functions in the `IO` module and those expect a `char_data` as argument, i.e. they expect a list of characters or strings to be given.

Although this is a subtle difference, you only need to worry about those details if you intend to pass lists to those functions. Binaries are already represented by the underlying bytes and as such their representation is always raw.

This finishes our tour on IO devices and IO related functionality. We have learned about four Elixir modules, `IO`, `File`, `Path` and `StringIO`, as well as how

the VM uses processes for the underlying IO mechanisms and how to use (char and io) data for IO operations.

13 alias, require and import

In order to facilitate software reuse, Elixir provides three directives. As we are going to see below, they are called directives because they have **lexical scope**.

13.1 alias

`alias` allows you to set up aliases for any given module name. Imagine our `Math` module uses a special list implementation for doing math specific operations:

```
1 fmodule Math do
2 alias Math.List, as: List
3 d
```

From now on, any reference to `List` will automatically expand to `Math.List`. In case one wants to access the original `List`, it can be done by accessing the module via `Elixir`:

```
1 st.flatten              #=> uses Math.List.flatten
2 ixir.List.flatten       #=> uses List.flatten
3 ixir.Math.List.flatten  #=> uses Math.List.flatten
```

> Note: All modules defined in Elixir are defined inside a main Elixir namespace. However, for convenience, you can omit "Elixir." when referencing them.

Aliases are frequently used to define shortcuts. In fact, calling `alias` without an `as` option sets the alias automatically to the last part of the module name, for example:

```
1 ias Math.List
```

Is the same as:

```
1 ias Math.List, as: List
```

Note that `alias` is **lexically scoped**, which allows you to set aliases inside specific functions:

```
fmodule Math do
def plus(a, b) do
  alias Math.List
  # ...
end

def minus(a, b) do
  # ...
end
d
```

In the example above, since we are invoking `alias` inside the function `plus/2`, the alias will just be valid inside the function `plus/2`. `minus/2` won't be affected at all.

13.2 require

Elixir provides macros as a mechanism for meta-programming (writing code that generates code).

Macros are chunks of code that are executed and expanded at compilation time. This means, in order to use a macro, we need to guarantee its module and implementation are available during compilation. This is done with the `require` directive:

```
x> Integer.odd?(3)
 (CompileError) iex:1: you must require Integer before invoking the macro Integer.odd?/1
x> require Integer
l
x> Integer.odd?(3)
ue
```

In Elixir, `Integer.odd?/1` is defined as a macro so it can be used as guards. This means that, in order to invoke `Integer.odd?/1`, we need to first require the `Integer` module.

In general a module does not need to be required before usage, except if we want to use the macros available in that module. An attempt to call a macro that was not loaded will raise an error. Note that like the `alias` directive, `require` is also lexically scoped. We will talk more about macros in a later chapter.

13.3 import

We use `import` whenever we want to easily access functions or macros from other modules without using the qualified name. For instance, if we want to use the `duplicate` function from `List` several times, we can simply import it:

```
x> import List, only: [duplicate: 2]
1
x> duplicate :ok, 3
ok, :ok, :ok]
```

In this case, we are importing only the function `duplicate` (with arity 2) from `List`. Although `only:` is optional, its usage is recommended. `except` could also be given as an option.

`import` also supports `:macros` and `:functions` to be given to `:only`. For example, to import all macros, one could write:

```
port Integer, only: :macros
```

Or to import all functions, you could write:

```
port Integer, only: :functions
```

Note that `import` is also **lexically scoped**, this means we can import specific macros inside specific functions:

```
fmodule Math do
def some_function do
  import List, only: [duplicate: 2]
  # call duplicate
end
d
```

In the example above, the imported `List.duplicate/2` is only visible within that specific function. `duplicate/2` won't be available in any other function in that module (or any other module for that matter).

Note that importing a module automatically requires it.

13.4 Aliases

At this point, you may be wondering, what exactly an Elixir alias is and how it is represented?

An alias in Elixir is a capitalized identifier (like `String`, `Keyword`, etc) which is converted to an atom during compilation. For instance, the `String` alias translates by default to the atom `:"Elixir.String"`:

```
x> is_atom(String)
ue
x> to_string(String)
lixir.String"
x> :"Elixir.String"
ring
```

By using the `alias/2` directive, we are simply changing what an alias translates to.

Aliases work as described because in the Erlang VM (and consequently Elixir), modules are represented by atoms. For example, that's the mechanism we use to call Erlang modules:

```
x> :lists.flatten([1,[2],3])
, 2, 3]
```

This is also the mechanism that allows us to dynamically call a given function in a module:

```
x> mod = :lists
ists
x> mod.flatten([1,[2],3])
,2,3]
```

In other words, we are simply calling the function `flatten` on the atom `:lists`.

13.5 Nesting

Now that we have talked about aliases, we can talk about nesting and how it works in Elixir. Consider the following example:

```
fmodule Foo do
defmodule Bar do
end
d
```

The example above will define two modules `Foo` and `Foo.Bar`. The second can be accessed as `Bar` inside `Foo` as long as they are in the same lexical scope. If later the developer decides to move `Bar` to another file, it will need to be referenced by its full name (`Foo.Bar`) or an alias needs to be set using the `alias` directive discussed above.

In other words, the code above is exactly the same as:

```
fmodule Elixir.Foo do
defmodule Elixir.Foo.Bar do
end
alias Elixir.Foo.Bar, as: Bar
d
```

As we will see in later chapters, aliases also play a crucial role in macros, to guarantee they are hygienic. With this we are almost finishing our tour about Elixir modules, the last topic to cover is module attributes.

14 Module attributes

Module attributes in Elixir serve three purposes:

1. They serve to annotate the module, often with information to be used by the user or the VM.
2. They work as constants.
3. They work as a temporary module storage to be used during compilation.

Let's check each case, one by one.

14.1 As annotations

Elixir brings the concept of module attributes from Erlang. For example:

```
fmodule MyServer do
@vsn 2
d
```

In the example above, we are explicitly setting the version attribute for that module. `@vsn` is used by the code reloading mechanism in the Erlang VM to check if a module has been updated or not. If no version is specified, the version is set to the MD5 checksum of the module functions.

Elixir has a handful of reserved attributes. Here are just a few of them, the most commonly used ones:

- `@moduledoc` - provides documentation for the current module.
- `@doc` - provides documentation for the function or macro that follows the attribute.
- `@behaviour` - (notice the British spelling) used for specifying an OTP or user-defined behaviour.
- `@before_compile` - provides a hook that will be invoked before the module is compiled. This makes it possible to inject functions inside the module exactly before compilation.

`@moduledoc` and `@doc` are by far the most used attributes, and we expect you to use them a lot. Elixir treats documentation as first-class and provides many functions to access documentation:

```
x> defmodule MyModule do
.>  @moduledoc "It does **x**"
.>
.>  @doc """
.>  Returns the version
.>  """
.>  def version, do: 1
.> end
module, MyModule, <<70, 79, 82, ...>>, {:version, 0}}
x> h MyModule
MyModule

 does **x**

x> h MyModule.version
def version()

turns the version
```

Elixir promotes the use of markdown with heredocs to write readable documentation. Heredocs are multiline strings, they start and end with triple quotes, keeping the formatting of the inner text:

```
fmodule Math do
@moduledoc """
This module provides mathematical functions
as sin, cos and constants like pi.

## Examples

    Math.pi
    #=> 3.1415...

"""
d
```

We also provide a tool called ExDoc which is used to generate HTML pages from the documentation.

You can take a look at the docs for Module for a complete list of supported attributes. Elixir also uses attributes to define typespecs, via:

- `@spec` - provides a specification for a function.
- `@callback` - provides a specification for the behaviour callback.
- `@type` - defines a type to be used in `@spec`.
- `@typep` - defines a private type to be used in `@spec`.
- `@opaque` - defines an opaque type to be used in `@spec`.

This section covers built-in attributes. However, attributes can also be used by developers or extended by libraries to support custom behaviour.

14.2 As constants

Elixir developers will often use module attributes to be used as constants:

```
1 fmodule MyServer do
2 @initial_state %{host: "147.0.0.1", port: 3456}
3 IO.inspect @initial_state
4 d
```

> Note: Unlike Erlang, user defined attributes are not stored in the module by default. The value exists only during compilation time. A developer can configure an attribute to behave closer to Erlang by calling `Module.register_attribute/3`.

Trying to access an attribute that was not defined will print a warning:

```
1 fmodule MyServer do
2 @unknown
3 d
4 rning: undefined module attribute @unknown, please remove access to @unknown or explicitly s
```

Finally, attributes can also be read inside functions:

```
1 fmodule MyServer do
2 @my_data 14
3 def first_data, do: @my_data
4 @my_data 13
5 def second_data, do: @my_data
6 d
7
8 Server.first_data #=> 14
9 Server.second_data #=> 13
```

Notice that reading an attribute inside a function takes a snapshot of its current value. In other words, the value is read at compilation time and not at runtime. As we are going to see, this makes attributes useful to be used as storage during module compilation.

14.3 As temporary storage

One of the projects in the Elixir organization is the `Plug` project, which is meant to be a common foundation for building web libraries and frameworks in Elixir.

The Plug library also allows developers to define their own plugs which can be run in a web server:

```
fmodule MyPlug do
use Plug.Builder

plug :set_header
plug :send_ok

def set_header(conn, _opts) do
  put_resp_header(conn, "x-header", "set")
end

def send_ok(conn, _opts) do
  send(conn, 200, "ok")
end
d

.puts "Running MyPlug with Cowboy on http://localhost:4000"
ug.Adapters.Cowboy.http MyPlug, []
```

In the example above, we have used the `plug/1` macro to connect functions that will be invoked when there is a web request. Internally, every time you call `plug/1`, the Plug library stores the given argument in a `@plugs` attribute. Just before the module is compiled, Plug runs a callback that defines a method (`call/2`) which handles http requests. This method will run all plugs inside `@plugs` in order.

In order to understand the underlying code, we'd need macros, so we will revisit this pattern in the meta-programming guide. However the focus here is exactly on how using module attributes as storage allow developers to create DSLs.

Another example comes from the ExUnit framework which uses module attributes as annotation and storage:

```
fmodule MyTest do
use ExUnit.Case

@tag :external
test "contacts external service" do
  # ...
end
d
```

Tags in ExUnit are used to annotate tests. Tags can be later used to filter tests. For example, you can avoid running external tests on your machine because they are slow and dependent on other services, while they can still be enabled in your build system.

We hope this section shines some light on how Elixir supports meta-programming and how module attributes play an important role when doing so.

In the next chapters we'll explore structs and protocols before moving to exception handling and other constructs like sigils and comprehensions.

15 Structs

In early chapters, we have learned about maps:

```
x> map = %{a: 1, b: 2}
a: 1, b: 2}
x> map[:a]

x> %{map | a: 3}
a: 3, b: 2}
```

Structs are extensions on top of maps that bring default values, compile-time guarantees and polymorphism into Elixir.

To define a struct, we just need to call `defstruct/1` inside a module:

```
x> defmodule User do
.>   defstruct name: "jose", age: 27
.> end
module, User,
<70, 79, 82, ...>>, {:__struct__, 0}}
```

We can now create "instances" of this struct by using the `%User{}` syntax:

```
x> %User{}
ser{name: "jose", age: 27}
x> %User{name: "eric"}
ser{name: "eric", age: 27}
x> is_map(%User{})
ue
```

Structs give compile-time guarantees that the provided fields exist in the struct:

```
x> %User{oops: :field}
 (CompileError) iex:3: unknown key :oops for struct User
```

When discussing maps, we demonstrated how we can access and update existing fields of a map. The same applies to structs:

```
x> jose = %User{}
ser{name: "jose", age: 27}
x> jose.name
ose"
x> eric = %{jose | name: "eric"}
ser{name: "eric", age: 27}
x> %{eric | oops: :field}
 (ArgumentError) argument error
```

By using the update syntax, the VM is aware no new keys will be added to the map/struct, allowing the maps to share their structure in memory. In the example above, both `jose` and `eric` share the same key structure in memory.

Structs can also be used in pattern matching and they guarantee the structs are of the same type:

```
x> %User{name: name} = jose
ser{name: "jose", age: 27}
x> name
ose"
x> %User{} = %{}
 (MatchError) no match of right hand side value: %{}
```

Matching works because structs store a field named `__struct__` inside the map:

```
x> jose.__struct__
er
```

Overall, a struct is just a bare map with default fields. Notice we say it is a bare map because none of the protocols implemented for maps are available for structs. For example, you can't enumerate nor access a struct:

```
x> user = %User{}
ser{name: "jose", age: 27}
x> user[:name]
 (Protocol.UndefinedError) protocol Access not implemented for %User{age: 27, name: "jose"}
```

A struct also is not a dictionary and therefore can't be used with the `Dict` module:

```
x> Dict.get(%User{}, :name)
 (ArgumentError) unsupported dict: %User{name: "jose", age: 27}
```

We will cover how structs interacts with protocols in the next chapter.

16 Protocols

Protocols are a mechanism to achieve polymorphism in Elixir. Dispatching on a protocol is available to any data type as long as it implements the protocol. Let's see an example.

In Elixir, only `false` and `nil` are treated as false. Everything else evaluates to true. Depending on the application, it may be important to specify a `blank?` protocol that returns a boolean for other data types that should be considered blank. For instance, an empty list or an empty binary could be considered blanks.

We could define this protocol as follows:

```
fprotocol Blank do
@doc "Returns true if data is considered blank/empty"
def blank?(data)
d
```

The protocol expects a function called `blank?` that receives one argument to be implemented. We can implement this protocol for different Elixir data types as follows:

```
Integers are never blank
fimpl Blank, for: Integer do
def blank?(_), do: false
d

Just empty list is blank
fimpl Blank, for: List do
def blank?([]), do: true
def blank?(_),  do: false
d

Just empty map is blank
fimpl Blank, for: Map do
# Keep in mind we could not pattern match on %{} because
# it matches on all maps. We can however check if the size
# is zero (and size is a fast operation).
def blank?(map), do: map_size(map) == 0
d

Just the atoms false and nil are blank
fimpl Blank, for: Atom do
def blank?(false), do: true
def blank?(nil),   do: true
def blank?(_),     do: false
d
```

And we would do so for all native data types. The types available are:

- `Atom`
- `BitString`
- `Float`
- `Function`
- `Integer`
- `List`
- `Map`
- `PID`
- `Port`
- `Reference`
- `Tuple`

Now with the protocol defined and implementations in hand, we can invoke it:

```
x> Blank.blank?(0)
lse
x> Blank.blank?([])
ue
x> Blank.blank?([1, 2, 3])
lse
```

Passing a data type that does not implement the protocol raises an error:

```
x> Blank.blank?("hello")
 (Protocol.UndefinedError) protocol Blank not implemented for "hello"
```

16.1 Protocols and structs

The power of Elixir's extensibility comes when protocols and structs are used together.

In the previous chapter, we have learned that although structs are maps, they do not share protocol implementations with maps. Let's define a `User` struct as in the previous chapter:

```
x> defmodule User do
.>   defstruct name: "jose", age: 27
.> end
module, User,
<70, 79, 82, ...>>, {:__struct__, 0}}
```

And then check:

```
x> Blank.blank?(%{})
ue
x> Blank.blank?(%User{})
 (Protocol.UndefinedError) protocol Blank not implemented for %User{age: 27, name: "jose"}
```

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

```
fimpl Blank, for: User do
def blank?(_), do: false
d
```

If desired you could come up with your own semantics for a user being blank. Not only that, you could use structs to build more robust data types, like queues, and implemented all relevant protocols, like `Enumerable` and possibly `Blank` for such data type.

In many cases though, developers may want to provide a default implementation for structs, as explicitly implementing the protocol for all structs can be tedious. That's when falling back to Any comes in handy.

16.2 Falling back to Any

It may be convenient to provide a default implementation for all types. This can be achieved by setting `@fallback_to_any` to `true` in the protocol definition:

```
1 fprotocol Blank do
2 @fallback_to_any true
3 def blank?(data)
4 d
```

Which can now be implemented as:

```
1 fimpl Blank, for: Any do
2 def blank?(_), do: false
3 d
```

Now all data types (including structs) that we have not implemented the `Blank` protocol for will be considered non-blank.

16.3 Built-in protocols

Elixir ships with some built-in protocols. In previous chapters, we have discussed the `Enum` module which provides many functions that work with any data structure that implements the `Enumerable` protocol:

```
1 x> Enum.map [1, 2, 3], fn(x) -> x * 2 end
2 ,4,6]
3 x> Enum.reduce 1..3, 0, fn(x, acc) -> x + acc end
4
```

Another useful example is the `String.Chars` protocol, which specifies how to convert a data structure with characters to a string. It's exposed via the `to_string` function:

```
x> to_string :hello
ello"
```

Notice that string interpolation in Elixir calls the `to_string` function:

```
x> "age: #{25}"
ge: 25"
```

The snippet above only works because numbers implement the `String.Chars` protocol. Passing a tuple, for example, will lead to an error:

```
x> tuple = {1, 2, 3}
, 2, 3}
x> "tuple: #{tuple}"
 (Protocol.UndefinedError) protocol String.Chars not implemented for {1, 2, 3}
```

When there is a need to "print" a more complex data structure, one can simply use the `inspect` function, based on the `Inspect` protocol:

```
x> "tuple: #{inspect tuple}"
uple: {1, 2, 3}"
```

The `Inspect` protocol is the protocol used to transform any data structure into a readable textual representation. This is what tools like IEx use to print results:

```
x> {1, 2, 3}
,2,3}
x> %User{}
ser{name: "jose", age: 27}
```

Keep in mind that, by convention, whenever the inspected value starts with `#`, it is representing a data structure in non-valid Elixir syntax. This means the inspect protocol is not reversible as information may be lost along the way:

```
x> inspect &(&1+2)
Function<6.71889879/1 in :erl_eval.expr/5>"
```

There are other protocols in Elixir but this covers the most common ones. In the next chapter we will learn a bit more about error handling and exceptions in Elixir.

17 try, catch and rescue

Elixir has three error mechanisms: errors, throws and exits. In this chapter we will explore each of them and include remarks about when each should be used.

17.1 Errors

A sample error can be retrieved by trying to add a number into an atom:

```
x> :foo + 1
 (ArithmeticError) bad argument in arithmetic expression
   :erlang.+(:foo, 1)
```

A runtime error can be raised any time by using the `raise/1` macro:

```
x> raise "oops"
 (RuntimeError) oops
```

Other errors can be raised with `raise/2` passing the error name and a list of keyword arguments:

```
x> raise ArgumentError, message: "invalid argument foo"
 (ArgumentError) invalid argument foo
```

You can also define your own errors using the `defexception/2` macro. The most common case is to define an exception with a message field:

```
x> defexception MyError, message: "default message"
x> raise MyError
 (MyError) default message
x> raise MyError, message: "custom message"
 (MyError) custom message
```

Exceptions can be rescued by using the `try/rescue` construct:

```
x> try do
.>   raise "oops"
.> rescue
.>   e in RuntimeError -> e
.> end
ntimeError[message: "oops"]
```

The example above rescues the runtime error and returns the error itself which is then printed in the `iex` session. In practice Elixir developers rarely use the `try/rescue` construct though. For example, many languages would force you to rescue an error when a file cannot open successfully. Elixir instead provides a `File.read/1` function which returns a tuple containing information if the file was opened with success or not:

```
x> File.read "hello"
error, :enoent}
x> File.write "hello", "world"
k
x> File.read "hello"
ok, "world"}
‘‘

ere is no ‘try/rescue‘ here. In case you want to handle multiple outcomes of opening a file,

‘iex
x> case File.read "hello" do
.>   {:ok, body} -> IO.puts "got ok"
.>   {:error, body} -> IO.puts "got error"
.> end
```

At the end of the day, it is up to your application to decide if an error while opening a file is exceptional or not. That's why Elixir doesn't impose exceptions on `File.read/1` and many other functions. Instead we leave it up to the developer to choose the best way to proceed.

For the cases where you do expect a file to exist (and the lack of a file is truly an error) you can simply use `File.read!/1`:

```
x> File.read! "unknown"
 (File.Error) could not read file unknown: no such file or directory
  (elixir) lib/file.ex:305: File.read!/1
```

In other words, we avoid using `try/rescue` because **we don't use errors for control flow**. In Elixir, we take errors literally: they are reserved to unexpected and/or exceptional situations. In case you actually need flow control constructs, throws must be used. That's what we are going to see next.

17.2 Throws

In Elixir, one can throw a value to be caught later. `throw` and `catch` are reserved for situations where it is not possible to retrieve a value unless by using `throw` and `catch`.

Those situations are quite uncommon in practice unless when interfacing with a library that does not provide the proper APIs. For example, let's imagine the `Enum` module did not provide any API for finding a value and we need to find the first number that is a multiple of 13:

```
x> try do
.>   Enum.each -50..50, fn(x) ->
.>     if rem(x, 13) == 0, do: throw(x)
.>   end
.>   "Got nothing"
.> catch
.>   x -> "Got #{x}"
.> end
ot -39"
```

However, in practice one can simply use `Enum.find/2`:

```
x> Enum.find -50..50, &(rem(&1, 13) == 0)
9
```

17.3 Exits

Every Elixir code runs inside processes that communicates with each other. When a process dies, it sends an `exit` signal. A process can also die by explicitly sending an exit signal:

```
x> spawn_link fn -> exit(1) end
ID<0.56.0>
 (EXIT from #PID<0.56.0>) 1
```

In the example above, the linked process died by sending an `exit` signal with value of 1. The Elixir shell automatically handles those messages and prints them to the terminal.

`exit` can also be "caught" using `try/catch`:

```
x> try do
.>   exit "I am exiting"
.> catch
.>   :exit, _ -> "not really"
.> end
ot really"
```

Using `try/catch` is already uncommon and using it to catch exits is even more rare.

`exit` signals are an important part of the fault tolerant system provided by the Erlang VM. Processes usually run under supervision trees which are themselves processes that just wait for `exit` signals of the supervised processes. Once an exit signal is received, the supervision strategy kicks in and the supervised process is restarted.

It is exactly this supervision system that makes constructs like `try/catch` and `try/rescue` so uncommon in Elixir. Instead of rescuing a certain error, we'd rather "fail fast" since the supervision tree will guarantee our application will go back to a known initial state after the error.

17.4 After

Sometimes it is necessary to use `try/after` to guarantee a resource is cleaned up after some particular action. For example, we can open a file and guarantee it is closed with `try/after` block:

```
x> {:ok, file} = File.open "sample", [:utf8, :write]
x> try do
.>   IO.write file, "jos"
.>   raise "oops, something went wrong"
.> after
.>   File.close(file)
.> end
 (RuntimeError) oops, something went wrong
```

17.5 Variables scope

It is important to bear in mind that variables defined inside `try/catch/rescue/after` blocks do not leak to the outer context. This is because the `try` block may fail and as such the variables may never be bound in the first place. In other words, this code is invalid:

```
x> try do
.>   from_try = true
.> after
.>   from_after = true
.> end
x> from_try
 (RuntimeError) undefined function: from_try/0
x> from_after
 (RuntimeError) undefined function: from_after/0
```

This finishes our introduction to `try`, `catch` and `rescue`. You will find they are used less frequently in Elixir than in other languages although they may be handy in some situations where a library or some particular code is not playing "by the rules".

It is time to talk about some Elixir constructs like comprehensions and sigils.

18 Comprehensions

In Elixir, it is common to loop over Enumerables, often filtering some results, and mapping to another list of values. Comprehensions are syntax sugar for such constructs, grouping those common tasks into the `for` special form.

For example, we can get all the square values of elements in a list as follows:

```
x> for n <- [1, 2, 3, 4], do: n * n
, 4, 9, 16]
```

A comprehension is made of three parts: generators, filters and collectables.

18.1 Generators and filters

In the expression above, `n <- [1, 2, 3, 4]` is the generator. It is literally generating values to be used in the comprehensions. Any enumerable can be passed in the right-hand side the generator expression:

```
x> for n <- 1..4, do: n * n
, 4, 9, 16]
```

Generator expressions also support pattern matching, ignoring all non-matching patterns. Imagine that instead of a range, we have a keyword list where the key is the atom `:good` or `:bad` and we only want to calculate the square of the good values:

```
x> values = [good: 1, good: 2, bad: 3, good: 4]
x> for {:good, n} <- values, do: n * n
, 4, 16]
```

Alternatively, filters can be used to filter some particular elements out. For example, we can get the square of only odd numbers:

```
x> require Integer
x> for n <- 1..4, Integer.odd?(n), do: n * n
, 9]
```

A filter will keep all values except `nil` or `false`.

Comprehensions in general provide a much more concise representation than using the equivalent functions from the `Enum` and `Stream` modules. Furthermore, comprehensions also allow multiple generators and filters to be given. Here is an example that receives a list of directories and deletes all files in those directories:

```
r dir  <- dirs,
  file <- File.ls!(dir),
  path = Path.join(dir, file),
  File.regular?(path) do
File.rm!(path)
d
```

Keep in mind that variable assignments inside the comprehension, be it in generators, filters or inside the block, are not reflected outside of the comprehension.

18.2 Bitstring generators

Bitstring generators are also supported and are very useful when you need to organize bitstring streams. The example below receives a list of pixels from a binary with their respective red, green and blue values and convert them into triplets:

```
x> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
x> for <<r::8, g::8, b::8 <- pixels>>, do: {r, g, b}
213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

A bitstring generator can be mixed with the "regular" enumerable generators and provide filters as well.

18.3 Into

In the examples above, the comprehension returned a list as a result.

However, the result of a comprehension can be inserted into different data structures by passing the `:into` option. For example, we can use bitstring generators with the `:into` option to easily remove all spaces in a string:

```
x> for <<c <- " hello world ">>, c != ?\s, into: "", do: <<c>>
elloworld"
```

Sets, maps and other dictionaries can also be given with the `:into` option. In general, the `:into` accepts any structure as long as it implements the `Collectable` protocol.

For example, the `IO` module provides streams, that are both `Enumerable` and `Collectable`. You can implement an echo terminal that returns whatever is typed, but in upcase, using comprehensions:

```
x> stream = IO.stream(:stdio, :line)
x> for line <- stream, into: stream do
.>   String.upcase(line) <> "\n"
.> end
```

Now type any string into the terminal and you will see the same value will be printed in upcase. Unfortunately, this example also got your shell stuck in the comprehension, so you will need to hit `Ctrl+C` twice to get out of it. :)

19 Sigils

We have already learned Elixir provides double-quoted strings and single-quoted char lists. However, this only covers the surface of structures that have textual representation in the language. Atoms are, for example, another structure which are mostly created via the `:atom` representation.

One of Elixir's goals is extensibility: developers should be able to extend the language to particular domains. Computer science has become such a wide field that it is impossible for a language to tackle many fields as part of its core. Our best bet is to rather make the language extensible, so developers, companies and communities can extend the language to their relevant domains.

In the chapter, we are going to explore sigils, which are one of the mechanisms provided by the language for working with textual representations.

19.1 Regular expressions

Sigils start with the tilde (`~`) character which is followed by a letter and then a separator. The most common sigil in Elixir is `~r` for regular expressions:

```
A regular expression that returns true if the text has foo or bar
x> regex = ~r/foo|bar/
/foo|bar/
x> "foo" =~ regex
ue
x> "bat" =~ regex
lse
```

Elixir provides Perl-compatible regular expressions (regexes), as implemented by the PCRE library. Regexes also support modifiers. For example, the `i` modifier makes a regular expression case insensitive:

```
x> "HELLO" =~ ~r/hello/
lse
x> "HELLO" =~ ~r/hello/i
ue
```

Check out the `Regex` module for more information on other modifiers and the supported operations with regular expressions.

So far, all examples have used `/` to delimit a regular expression. However sigils support 8 different separators:

```
/hello/
|hello|
"hello"
'hello'
(hello)
[hello]
{hello}
<hello>
```

The reasoning in supporting different operators is that different separators can be more convenient to different sigils. For example, using parentheses for regular expressions may be a confusing choice as they can get mixed with the parentheses inside the regex. However, parentheses can be handy for other sigils, as we will see in the next section.

19.2 Strings, char lists and words sigils

Besides regular expressions, Elixir ships with three other sigils.

The `~s` sigil is used to generate strings, similar to double quotes:

```
x> ~s(this is a string with "quotes")
his is a string with \"quotes\""
```

While `~c` is used to generate char lists:

```
x> ~c(this is a string with "quotes")
his is a string with "quotes"'
```

The `~w` sigil is used to generate a list of words separated by white space:

```
x> ~w(foo bar bat)
foo", "bar", "bat"]
```

The `~w` sigil also accepts the `c`, `s` and `a` modifiers to choose the format of the result:

```
x> ~w(foo bar bat)a
foo, :bar, :bat]
```

Besides lowercase sigils, Elixir supports uppercase sigils. While both `~s` and `~S` will return strings, the first one allows escape codes and interpolation while the second does not:

```
x> ~s(String with escape codes \x26 interpolation)
tring with escape codes & interpolation"
x> ~S(String without escape codes and without #{interpolation})
tring without escape codes and without \#{interpolation}"
```

The following escape codes can be used in strings and char lists:

- `\"` – double quote
- `\'` – single quote
- `\\` – single backslash
- `\a` – bell/alert
- `\b` – backspace
- `\d` - delete
- `\e` - escape
- `\f` - form feed
- `\n` – newline
- `\r` – carriage return
- `\s` – space
- `\t` – tab
- `\v` – vertical tab
- `\DDD`, `\DD`, `\D` - character with octal representation DDD, DD or D (example: `\377`)
- `\xDD` - character with hexadecimal representation DD (example: `\x13`)
- `\x{D...}` - character with hexadecimal representation with one or more hexadecimal digits (example: `\x{abc13}`)

Sigils also support heredocs which is when tripe double- or single-quotes are used as separators:

```
x> ~s"""
.> this is
.> a heredoc string
.> """
```

The most common case for heredoc sigils is when writing documentation. For example, if you need to write escape characters in your documentation, it can become error prone as we would need to double-escape some characters:

```
oc """
nverts double-quotes to single-quotes.

 Examples

  iex> convert("\\\"foo\\\"")
  "'foo'"

"
f convert(...)
```

By using using `~S`, we can avoid this problem altogether:

```
oc ~S"""
nverts double-quotes to single-quotes.

 Examples

  iex> convert("\"foo\"")
  "'foo'"

"
f convert(...)
```

19.3 Custom sigils

As hinted at the beginning of this chapter, sigils in Elixir are extensible. In fact, the sigil `~r/foo/i` is equivalent to calling the `sigil_r` function with two arguments:

```
x> sigil_r(<<"foo">>, 'i')
"foo"i
```

That said, we can access the documentation for the `~r` sigil via the `sigil_r` function:

```
x> h sigil_r
.
```

We can also provide our own sigils by simply implementing the proper function. For example, let's implement the `~i(13)` sigil that returns an integer:

```
x> defmodule MySigils do
.>   def sigil_i(binary, []), do: binary_to_integer(binary)
.> end
x> import MySigils
x> ~i(13)

```

Sigils can also be used to do compile-time work with the help of macros. For example, regular expressions in Elixir are compiled into efficient representation during compilation of the source code, therefore skipping this step at runtime. If you have interest in the subject, we recommend you to learn more about macros and check how those sigils are implemented in the `Kernel` module.

20 Where to go next

Eager to learn more? Keep on reading!

20.1 Build your first Elixir project

In order to get your first project started, Elixir ships with a build tool called Mix. You can get your new project started by simply running:

```
mix new path/to/new/project
```

We have written a guide that covers how to build an Elixir application, with its own supervision tree, configuration, tests and more. The application works as a distributed key-value store where we organize key-value pairs into buckets and distribute those buckets across multiple nodes:

- Mix and OTP

20.2 Community and other resources

On the sidebar, you can find the link to some Elixir books and screencasts. They are plenty of Elixir resources out there, like conference talks, open source projects, and other learning material produced by the community.

Remember that in case of any difficulties, you can always visit the **#elixir-lang** channel on **irc.freenode.net** or send a message to the mailing list. You can be sure that there will be someone willing to help. And to keep posted on the latest news and announcements, follow the blog and follow language developments on the elixir-core mailing list.

Don't forget you can also check the source code of Elixir itself, which is mostly written in Elixir (mainly the `lib` directory), or explore Elixir's documentation.

20.3 A Byte of Erlang

As the main page of this site puts it:

> Elixir is a programming language built on top of the Erlang VM.

Sooner than later, an Elixir developer will want to interface with existing Erlang libraries. Here's a list of online resources that cover Erlang's fundamentals and its more advanced features:

- This Erlang Syntax: A Crash Course provides a concise intro to Erlang's syntax. Each code snippet is accompanied by equivalent code in Elixir. This is an opportunity for you to not only get some exposure to the Erlang's syntax but also review some of the things you have learned in the present guide.
- Erlang's official website has a short tutorial with pictures that briefly describe Erlang's primitives for concurrent programming.
- Learn You Some Erlang for Great Good! is an excellent introduction to Erlang, its design principles, standard library, best practices and much more. If you are serious about Elixir, you'll want to get a solid understanding of Erlang principles. Once you have read through the crash course mentioned above, you'll be able to safely skip the first couple of chapters in the book that mostly deal with the syntax. When you reach The Hitchhiker's Guide to Concurrency chapter, that's where the real fun starts.