elixir

@elixirlang / elixir-lang.org

Table of Contents

Elixir Getting Started

Introduction

Welcome!

In this tutorial we are going to teach you the Elixir foundation, the language syntax, how to define modules, how to manipulate the characteristics of common data structures and more. This chapter will focus on ensuring Elixir is installed and that you can successfully run Elixir's Interactive Shell, called IEx.

Our requirements are:

- Elixir - Version 1.2.0 onwards
- Erlang - Version 18.0 onwards

Let's get started!

> 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.

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` (or `iex.bat` if you are on Windows) which stands for Interactive Elixir. In interactive mode, we can type any Elixir expression and get its result. Let's warm up with some basic expressions.

Open up `iex` and type the following expressions:

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

iex> 40 + 2
42
iex> "hello" <> " world"
"hello 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.

> Note: if you are on Windows, you can also try `iex.bat --werl` which may provide a better experience depending on which console you are using.

Installation

If you still haven't installed Elixir, run to our installation page. Once you are done, you can run `elixir -v` to get the current Elixir version.

Running scripts

After getting familiar with the basics of the language you may want to try writing simple programs. This can be accomplished by putting the following Elixir code into a file:

```
IO.puts "Hello world from Elixir"
```

Save it as `simple.exs` and execute it with `elixir` :

```
$ elixir simple.exs
Hello world from Elixir
```

Later on we will learn how to compile Elixir code (in Chapter 8) and how to use the Mix build tool (in the Mix & OTP guide). For now, let's move on to Chapter 2.

Asking questions

When going through this getting started guide, it is common to have questions, after all, that is part of the learning process! There are many places you could ask them to learn more about Elixir:

- ## elixir-lang on freenode IRC

- Elixir on Slack
- Elixir Forum
- elixir-talk mailing list
- elixir tag on StackOverflow

When asking questions, remember these two tips:

- Instead of asking "how to do X in Elixir", ask "how to solve Y in Elixir". In other words, don't ask how to implement a particular solution, instead describe the problem at hand. Stating the problem gives more context and less bias for a correct answer.

- In case things are not working as expected, please include as much information as you can in your report, for example: your Elixir version, the code snippet and the error message alongside the error stacktrace. Use sites like Gist to paste this information.

Basic types

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

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

Basic arithmetic

Open up `iex` and type the following expressions:

```
iex> 1 + 2
3
iex> 5 * 5
25
iex> 10 / 2
5.0
```

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:

```
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

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

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

```
iex> 0b1010
10
iex> 0o777
511
iex> 0x1F
31
```

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

```
iex> 1.0
1.0
iex> 1.0e-10
1.0e-10
```

Floats in Elixir are 64 bit double precision.

You can invoke the `round` function to get the closest integer to a given float, or the `trunc` function to get the integer part of a float.

```
iex> round(3.58)
4
iex> trunc(3.58)
3
```

Booleans

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

```
iex> true
true
iex> true == false
false
```

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.

```
iex> is_boolean(true)
true
iex> is_boolean(1)
false
```

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`).

Atoms

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

```
iex> :hello
:hello
iex> :hello == :world
false
```

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

```
iex> true == :true
true
iex> is_atom(false)
true
iex> is_boolean(:false)
true
```

Strings

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

```
iex> "hellö"
"hellö"
```

> Note: if you are running on Windows, there is a chance your terminal does not use UTF-8 by default. You can change the encoding of your current session by running `chcp 65001` before entering IEx.

Elixir also supports string interpolation:

```
iex> "hellö #{:world}"
"hellö world"
```

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

```
iex> "hello
...> world"
"hello\nworld"
iex> "hello\nworld"
"hello\nworld"
```

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

```
iex> IO.puts "hello\nworld"
hello
world
:ok
```

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:

```
iex> is_binary("hellö")
true
```

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

```
iex> byte_size("hellö")
6
```

Notice the number of bytes in that string is 6, even though it has 5 characters. That's because the character "ö" 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:

```
iex> String.length("hellö")
5
```

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

```
iex> String.upcase("hellö")
"HELLÖ"
```

Anonymous functions

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

```
iex> add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> is_function(add)
true
iex> is_function(add, 2)
true
iex> is_function(add, 1)
false
iex> add.(1, 2)
3
```

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 (`.`) 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. Let's define a new anonymous function that uses the `add` anonymous function we have previously defined:

```
iex> double = fn a -> add.(a, a) end
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> double.(2)
4
```

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

```
iex> x = 42
42
iex> (fn -> x = 0 end).()
0
iex> x
42
```

The capture syntax `&amp;()` can also be used for creating anonymous functions. This type of syntax will be discussed in Chapter 8.

Tuples

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

```
iex> {:ok, "hello"}
{:ok, "hello"}
iex> tuple_size {:ok, "hello"}
2
```

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:

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
iex> tuple_size(tuple)
2
```

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

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

(Linked) Lists

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

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

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

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

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

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

```
iex> hd []
** (ArgumentError) argument error
```

Sometimes you will create a list and it will return a value in single-quotes. For example:

```
iex> [11, 12, 13]
'\v\f\r'
iex> [104, 101, 108, 108, 111]
'hello'
```

When Elixir sees a list of printable ASCII numbers, Elixir will print that as a char list (literally a list of characters). Char lists are quite common when interfacing with existing Erlang code. Whenever you see a value in IEx and you are not quite sure what it is, you can use the `i/1` to retrieve information about it:

```
iex> i 'hello'
Term
  'hello'
Data type
  List
Description
  ...
Raw representation
  [104, 101, 108, 108, 111]
Reference modules
  List
```

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

```
iex> 'hello' == "hello"
false
```

Single-quotes are char lists, double-quotes are strings. We will talk more about them in the "Binaries, strings and char lists" chapter.

Lists or tuples?

What is the difference between lists and tuples?

Lists are stored in memory as linked lists, meaning that each element in a list holds its value and points to the following element until the end of the list is reached. We call each pair of value and pointer a **cons cell**:

```
iex> list = [1 | [2 | [3 | []]]]
[1, 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:

```
iex> [0 | list]
[0, 1, 2, 3]
```

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:

```
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> 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 the first element and the file contents as the second. Otherwise, it returns a tuple with `:error` and the error description.

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:

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

When "counting" the number of elements in a data structure, Elixir also abides by a simple rule: the function is named `size` if the operation is in constant time (i.e. the value is pre-calculated) or `length` if the operation is linear (i.e. calculating the length gets slower as the input grows).

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 graphemes 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 traversed.

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.

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:

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

String concatenation is done with `&lt;&gt;`:

```
iex> "foo" <> "bar"
"foobar"
```

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:

```
iex> true and true
true
iex> false or is_atom(:example)
true
```

Providing a non-boolean will raise an exception:

```
iex> 1 and true
** (ArgumentError) argument error: 1
```

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

```
iex> false and raise("This error will never be raised")
false

iex> true or raise("This error will never be raised")
true
```

> 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 `||` , `&amp;&amp;` and `!` which accept arguments of any type. For these operators, all values except `false` and `nil` will evaluate to true:

```
# or
iex> 1 || true
1
iex> false || 11
11

# and
iex> nil && 13
nil
iex> true && 17
17

# !
iex> !true
false
iex> !1
false
iex> !nil
true
```

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

Elixir also provides `==` , `!=` , `===` , `!==` , `&lt;=` , `&gt;=` , `&lt;` and `&gt;` as comparison operators:

```
iex> 1 == 1
true
iex> 1 != 2
true
iex> 1 < 2
true
```

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

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
```

In Elixir, we can compare two different data types:

```
iex> 1 < :atom
true
```

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.

Operator table

Although we have learned only a handful of operators so far, we present below the complete operator table for Elixir ordered from higher to lower precedence for reference:

Operator	Associativity					
`@`	Unary					
`.`	Left to right					
`+` `-` `!` `^` `not` `~~~`	Unary					
`*` `/`	Left to right					
`+` `-`	Left to right					
`++` `--` `..` `&lt;&gt;`	Right to left					
`in`	Left to right					
`	> `` `&lt;&lt;&lt;` `&gt;&gt;&gt;` `~&gt;&gt;` `&lt;&lt;~` `~&gt;` `&lt;~` `&lt;~&gt;` `<`	>`	Left to right			
`&lt;` `&gt;` `&lt;=` `&gt;=`	Left to right					
`==` `!=` `=~` `===` `!==`	Left to right					
`&amp;&amp;` `&amp;&amp;&amp;` `and`	Left to right					
`					`or	Left to right
`=`	Right to left					
`=&gt;`	Right to left					
`	`	Right to left				
`::`	Right to left					
`when`	Right to left					
`&lt;-` `,` `\\`	Left to right					
`&amp;`	Unary					

We will learn the majority of those operators as we go through the getting started guide. In the next chapter, we are going to discuss some basic functions, data type conversions and a bit of control-flow.

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.

The match operator

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

```
iex> x = 1
1
iex> x
1
```

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

```
iex> 1 = x
1
iex> 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 `=`:

```
iex> 1 = unknown
** (CompileError) iex:1: undefined function unknown/0
```

Since there is no variable `unknown` previously defined, Elixir imagined you were trying to call a function named `unknown/0`, but such a function does not exist.

Pattern matching

The match operator is not only used to match against simple values, but it is also useful for destructuring more complex data types. For example, we can pattern match on tuples:

```
iex> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex> a
:hello
iex> b
"world"
```

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

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

And also when comparing different types:

```
iex> {a, b, c} = [:hello, "world", 42]
** (MatchError) no match of right hand side value: [:hello, "world", 42]
```

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`:

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

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

We can pattern match on lists:

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

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

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

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

```
iex> [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:

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

The pin operator

Variables in Elixir can be rebound:

```
iex> x = 1
1
iex> x = 2
2
```

The pin operator `^` should be used when you want to pattern match against an existing variable's value rather than rebinding the variable:

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

Because we have assigned the value of 1 to the variable x, this last example could also have been written as:

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

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

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

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

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

```
iex> _
** (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:

```
iex> length([1, [2], 3]) = 3
** (CompileError) iex:1: 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.

case, cond and if

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

`case`

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

```
iex> 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
"This clause will match and bind x to 2 in this clause"
```

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

```
iex> x = 1
1
iex> case 10 do
...>   ^x -> "Won't match"
...>   _  -> "Will match"
...> end
"Will match"
```

Clauses also allow extra conditions to be specified via guards:

```
iex> case {1, 2, 3} do
...>   {1, x, 3} when x > 0 ->
...>     "Will match"
...>   _ ->
...>     "Would match, if guard condition were not satisfied"
...> end
"Will match"
```

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

Expressions in guard clauses

Elixir imports and allows the following expressions in guards by default:

- comparison operators (`==` , `!=` , `===` , `!==` , `&gt;` , `&gt;=` , `&lt;` , `&lt;=`)
- boolean operators (`and` , `or` , `not`)
- arithmetic operations (`+` , `-` , `*` , `/`)
- arithmetic unary operators (`+` , `-`)
- the binary concatenation operator `&lt;&gt;`
- the `in` operator as long as the right side is a range or a list
- 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_nil/1`
    - `is_number/1`
    - `is_pid/1`
    - `is_port/1`
    - `is_reference/1`
    - `is_tuple/1`
- plus these functions:
    - `abs(number)`
    - `binary_part(binary, start, length)`
    - `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)`

Additionally, users may define their own guards. For example, the `Bitwise` module defines guards as functions and operators: `bnot` , `~~~` , `band` , `&amp;&amp;&amp;` , `bor` , `|||` , `bxor` , `^^^` , `bsl` , `&lt;&lt;&lt;` , `bsr` , `&gt;&gt;&gt;` .

Note that while boolean operators such as `and` , `or` , `not` are allowed in guards, the more general and short-circuiting operators `&amp;&amp;` , `||` and `!` are not.

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

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

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

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

Note anonymous functions can also have multiple clauses and guards:

```
iex> f = fn
...>   x, y when x > 0 -> x + y
...>   x, y -> x * y
...> end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> f.(1, 3)
4
iex> f.(-1, 3)
-3
```

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

```
iex> f2 = fn
...>   x, y when x > 0 -> x + y
...>   x, y, z -> x * y + z
...> end
** (CompileError) iex:1: cannot mix clauses with different arities in function definit
ion
```

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`:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But 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(`CondClauseError`) is raised. For this reason, it may be necessary to add a final condition, equal to `true`, which will always match:

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

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

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

`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:

```
iex> if true do
...>   "This works!"
...> end
"This works!"
iex> unless true do
...>   "This will never be seen"
...> end
nil
```

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:

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

`do/end` 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:

```
iex> if true, do: 1 + 2
3
```

Notice how the example above has a comma between `true` and `do:`, that's because it is using Elixir's regular syntax where each argument is separated by comma. We say this syntax is using **keyword lists**. We can pass `else` using keywords too:

```
iex> if false, do: :this, else: :that
:that
```

`do/end` blocks are a syntactic convenience built on top of the keywords one. That's why `do/end` blocks do not require a comma between the previous argument and the block. They are useful exactly because they remove the verbosity when writing blocks of code. These are equivalent:

```
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13
```

One thing to keep in mind when using `do/end` blocks is they are always bound to the outermost function call. For example, the following expression:

```
iex> is_number if true do
...>  1 + 2
...> end
** (CompileError) undefined function: is_number/2
```

Would be parsed as:

```
iex> is_number(if true) do
...>  1 + 2
...> end
** (CompileError) undefined function: is_number/2
```

which leads to an undefined function error as Elixir attempts to invoke `is_number/1`, but passing it *two* arguments (the `if true` expression - which would throw an undefined function error itself as `if` needs a second argument - the `do/end` block). Adding explicit parentheses is enough to resolve the ambiguity:

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

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".

Binaries, strings, and char lists

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

```
iex> string = "hello"
"hello"
iex> is_binary(string)
true
```

In this chapter, we will understand what binaries are, how they associate with strings, and what a single-quoted value, `&#039;like this&#039;`, means in Elixir.

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 `ł` has code point `322`. When writing the string `&quot;hełło&quot;` 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 `&quot;hełło&quot;`, because it uses the code point `322` for `ł`, and one byte can only represent a number from `0` to `255`. But of course, given you can actually read `&quot;hełło&quot;` 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 `ł` assigned to 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`:

```
iex> string = "hełło"
"hełło"
iex> byte_size(string)
7
iex> String.length(string)
5
```

> Note: if you are running on Windows, there is a chance your terminal does not use UTF-8 by default. You can change the encoding of your current session by running `chcp 65001` before entering `iex` (`iex.bat`).

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

```
iex> ?a
97
iex> ?ł
322
```

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

```
iex> String.codepoints("hełło")
["h", "e", "ł", "ł", "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 article "The string type is broken".

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!

Binaries (and bitstrings)

In Elixir, you can define a binary using `&lt;&lt;&gt;&gt;`:

```
iex> <<0, 1, 2, 3>>
<<0, 1, 2, 3>>
iex> byte_size(<<0, 1, 2, 3>>)
4
```

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:

```
iex> String.valid?(<<239, 191, 191>>)
false
```

The string concatenation operation is actually a binary concatenation operator:

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

A common trick in Elixir is to concatenate the null byte `&lt;&lt;0&gt;&gt;` to a string to see its inner binary representation:

```
iex> "hełło" <> <<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:

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

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

```
iex> <<1 :: size(1)>>
<<1::size(1)>>
iex> <<2 :: size(1)>> # truncated
<<0::size(1)>>
iex> is_binary(<< 1 :: size(1)>>)
false
iex> is_bitstring(<< 1 :: size(1)>>)
true
iex> bit_size(<< 1 :: size(1)>>)
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:

```
iex> <<0, 1, x>> = <<0, 1, 2>>
<<0, 1, 2>>
iex> x
2
iex> <<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 pattern is expected to match exactly 8 bits. If we want to match on a binary of unknown size, it is possible by using the binary modifier at the end of the pattern:

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

Similar results can be achieved with the string concatenation operator `&lt;&gt;` :

```
iex> "he" <> rest = "hello"
"hello"
iex> rest
"llo"
```

A complete reference about the binary / bitstring constructor `&lt;&lt;&gt;&gt;` can be found in the Elixir documentation. This concludes 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 for working 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.

Char lists

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

```
iex> 'hełło'
[104, 101, 322, 322, 111]
iex> is_list 'hełło'
true
iex> 'hello'
'hello'
```

You can see that, instead of containing bytes, a char list contains the code points of the characters 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:

```
iex> to_char_list "hełło"
[104, 101, 322, 322, 111]
iex> to_string 'hełło'
"hełło"
iex> to_string :hello
"hello"
iex> to_string 1
"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.

Keywords and maps

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, etc.

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

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:

```
iex> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex> list == [a: 1, b: 2]
true
iex> list[:a]
1
```

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, we can use all operations available to lists. For example, we can use `++` to add new values to a keyword list:

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

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

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

Keyword lists are important because they have three special characteristics:

- Keys must be atoms.
- Keys are ordered, as specified by the developer.
- Keys can be given more than once.

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

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

These 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:

```
iex> if false, do: :this, else: :that
:that
```

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

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

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.

Although we can pattern match on keyword lists, it is rarely done in practice since pattern matching on lists requires the number of items and their order to match:

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

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> map[:a]
1
iex> map[2]
:b
iex> map[:c]
nil
```

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

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

In contrast to keyword lists, maps are very useful with pattern matching. When a map is used in a pattern, it will always match on a subset of the given value:

```
iex> %{} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> %{:a => a} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> a
1
iex> %{: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 keys in the pattern exist in the given map. Therefore, an empty map matches all maps.

Variables can be used when accessing, matching and adding map keys:

```
iex> n = 1
1
iex> map = %{n => :one}
%{1 => :one}
iex> map[n]
:one
iex> %{^n => :one} = %{1 => :one, 2 => :two, 3 => :three}
%{1 => :one, 2 => :two, 3 => :three}
```

The `Map` module provides a very similar API to the `Keyword` module with convenience functions to manipulate maps:

```
iex> Map.get(%{:a => 1, 2 => :b}, :a)
1
iex> Map.to_list(%{:a => 1, 2 => :b})
[{2, :b}, {:a, 1}]
```

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

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

Another interesting property of maps is that they provide their own syntax for updating and accessing atom keys:

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

iex> map.a
1
iex> map.c
** (KeyError) key :c not found in: %{2 => :b, :a => 1}

iex> %{map | :a => 2}
%{2 => :b, :a => 2}
iex> %{map | :c => 3}
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

Both access and update syntaxes above require the given keys to exist. For example, accessing and updating the `:c` key failed because there is no `:c` in the map.

Elixir developers typically prefer to use the `map.field` syntax and pattern matching instead of the functions in the `Map` module when working with maps because they lead to an assertive style of programming. This blog post provides insight and examples on how you get more concise and faster software by writing assertive code in Elixir.

> Note: Maps were recently introduced into the Erlang *VM* and only from Elixir v1.2 they are capable of holding millions of keys efficiently. Therefore, if you are working with previous Elixir versions (v1.0 or v1.1) and you need to support at least hundreds of keys, you may consider using the `HashDict` module.

Nested data structures

Often we will have maps inside maps, or even keywords lists inside maps, and so forth. Elixir provides conveniences for manipulating nested data structures via the `put_in/2`, `update_in/2` and other macros giving the same conveniences you would find in imperative languages while keeping the immutable properties of the language.

Imagine you have the following structure:

```
iex> users = [
  john: %{name: "John", age: 27, languages: ["Erlang", "Ruby", "Elixir"]},
  mary: %{name: "Mary", age: 29, languages: ["Elixir", "F#", "Clojure"]}
]
[john: %{age: 27, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
 mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"], name: "Mary"}]
```

We have a keyword list of users where each value is a map containing the name, age and a list of programming languages each user likes. If we wanted to access the age for john, we could write:

```
iex> users[:john].age
27
```

It happens we can also use this same syntax for updating the value:

```
iex> users = put_in users[:john].age, 31
[john: %{age: 31, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
 mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"], name: "Mary"}]
```

The `update_in/2` macro is similar but allows us to pass a function that controls how the value changes. For example, let's remove "Clojure" from Mary's list of languages:

```
iex> users = update_in users[:mary].languages, &List.delete(&1, "Clojure")
[john: %{age: 31, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
 mary: %{age: 29, languages: ["Elixir", "F#"], name: "Mary"}]
```

There is more to learn about `put_in/2` and `update_in/2`, including the `get_and_update_in/2` that allows us to extract a value and update the data structure at once. There are also `put_in/3`, `update_in/3` and `get_and_update_in/3` which allow dynamic access into the data structure. Check their respective documentation in the `Kernel` module for more information.

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.

Modules

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

```
iex> String.length("hello")
5
```

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:

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

iex> Math.sum(1, 2)
3
```

In the following sections, our examples are going to get longer in size, 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.

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:

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

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):

```
iex> Math.sum(1, 2)
3
```

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.

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. When executed, both extensions compile and load their modules into memory, although only `.ex` files write their bytecode to disk in the format of `.beam` files.

For instance, we can create a file called `math.exs`:

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

IO.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.

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.

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

  defp do_sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)    #=> 3
IO.puts Math.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:

```
defmodule Math do
  def zero?(0) do
    true
  end

  def zero?(x) when is_integer(x) do
    false
  end
end

IO.puts Math.zero?(0)         #=> true
IO.puts Math.zero?(1)         #=> false
IO.puts Math.zero?([1, 2, 3]) #=> ** (FunctionClauseError)
IO.puts Math.zero?(0.0)       #=> ** (FunctionClauseError)
```

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

Similar to constructs like `if`, named functions support both `do:` and `do`/`end` block syntax, as we learned `do`/`end` is just a convenient syntax for the keyword list format. For example, we can edit `math.exs` to look like this:

```
defmodule Math do
  def zero?(0), do: true
  def zero?(x) when is_integer(x), do: false
end
```

And it will provide the same behaviour. You may use `do:` for one-liners but always use `do` / `end` for functions spanning multiple lines.

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. Start `iex`, running the `math.exs` file defined above:

```
$ iex math.exs
```

```
iex> Math.zero?(0)
true
iex> fun = &Math.zero?/1
&Math.zero?/1
iex> is_function(fun)
true
iex> fun.(0)
true
```

Local or imported functions, like `is_function/1`, can be captured without the module:

```
iex> &is_function/1
&:erlang.is_function/1
iex> (&is_function/1).(fun)
true
```

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

```
iex> fun = &(&1 + 1)
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> fun.(1)
2
```

The `&amp;1` represents the first argument passed into the function. `&amp;(&amp;1+1)` above is exactly the same as `fn x -&gt; x + 1 end`. The syntax above is useful for short function definitions.

If you want to capture a function from a module, you can do `&amp;Module.function()`:

```
iex> fun = &List.flatten(&1, &2)
&List.flatten/2
iex> fun.([1, [[2], 3]], [4, 5])
[1, 2, 3, 4, 5]
```

`&amp;List.flatten(&amp;1, &amp;2)` is the same as writing `fn(list, tail) -&gt; List.flatten(list, tail) end` which in this case is equivalent to `&amp;List.flatten/2`. You can read more about the capture operator `&amp;` in the `Kernel.SpecialForms` documentation.

Default arguments

Named functions in Elixir also support default arguments:

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

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.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:

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

```
iex> DefaultTest.dowork
hello
:ok
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
hello
:ok
```

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

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

  def join(a, b, _sep) when is_nil(b) do
    a
  end

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

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

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

```
defmodule 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
end
```

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

```
concat.ex:7: warning: this clause cannot match because a previous clause at line 2 alw
ays 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
```

```
iex> Concat.join "Hello", "world"
***First join
"Helloworld"
```

```
iex> Concat.join "Hello", "world", "_"
***Second join
"Hello_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.

Recursion

Loops through recursion

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

```
for(i = 0; i < sizeof(array); i++) {
  array[i] = array[i] * 2;
}
```

In the example above, we are mutating both the array and the variable `i`. Mutating is 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. No data is mutated in this process. Consider the example below that prints a string an arbitrary number of times:

```
defmodule 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
end

Recursion.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`.

When `print_multiple_times/2` is initially called in the example above, 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 in 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 one step closer to our base case.

Reduce and map algorithms

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

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

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

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

We invoke `sum_list` with the 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 binds `head` to `1` and `tail` to `[2, 3]` ; `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:

```
sum_list [1, 2, 3], 0
sum_list [2, 3], 1
sum_list [3], 3
sum_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?

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

  def double_each([]) do
    []
  end
end
```

```
iex math.exs
```

```
iex> Math.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 in Elixir you will rarely use recursion as above to manipulate lists.

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

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

Or, using the capture syntax:

```
iex> Enum.reduce([1, 2, 3], 0, &+/2)
6
iex> Enum.map([1, 2, 3], &(&1 * 2))
[2, 4, 6]
```

Let's take a deeper look at `Enumerable` s and, while we're at it, their lazy counterpart, `Stream` s.

Enumerables and Streams

Enumerables

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

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

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

The functions in the Enum module are limited to, as the name says, enumerating values in data structures. For specific operations, like inserting and updating particular elements, you may need to reach for modules specific to the data type. 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 a stream.

Eager vs Lazy

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

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

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

```
iex> 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
7500000000
```

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.

The pipe operator

The `|&gt;` symbol used in the snippet above is the **pipe operator**: it simply takes the output from the expression on its left side and passes it as the first argument to the function call on its right side. It's similar to the Unix `|` operator. Its purpose is to highlight the flow of data being transformed by a series of functions. To see how it can make the code cleaner, have a look at the example above rewritten without using the `|&gt;` operator:

```
iex> Enum.sum(Enum.filter(Enum.map(1..100_000, &(&1 * 3)), odd?))
7500000000
```

Find more about the pipe operator by reading its documentation.

Streams

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

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
7500000000
```

Streams are lazy, composable enumerables.

In the example above, `1..100_000 |&gt; Stream.map(&amp;(&amp;1 * 3))` returns a data type, an actual stream, that represents the `map` computation over the range `1..100_000`:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3))
#Stream<[enum: 1..100000, funs: [#Function<34.16982430/1 in Stream.map/2>]]>
```

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

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?)
#Stream<[enum: 1..100000, funs: [...]]>
```

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

Many functions in the `Stream` module accept any enumerable as an argument and return a stream as a result. It also provides functions for creating streams. 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:

```
iex> stream = Stream.cycle([1, 2, 3])
#Function<15.16982430/2 in Stream.cycle/1>
iex> Enum.take(stream, 10)
[1, 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:

```
iex> stream = Stream.unfold("hełło", &String.next_codepoint/1)
#Function<39.75994740/2 in Stream.unfold/2>
iex> Enum.take(stream, 3)
["h", "e", "ł"]
```

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 the case of failures. For example, we can use it to stream a file:

```
iex> stream = File.stream!("path/to/file")
#Function<18.16982430/2 in Stream.resource/3>
iex> 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 functionality in the `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.

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 tens or even hundreds 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.

`spawn`

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

```
iex> spawn fn -> 1 + 2 end
#PID<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:

```
iex> pid = spawn fn -> 1 + 2 end
#PID<0.44.0>
iex> Process.alive?(pid)
false
```

> 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`:

```
iex> self()
#PID<0.41.0>
iex> Process.alive?(self())
true
```

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

`send` and `receive`

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

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

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

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:

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

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

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

```
iex> parent = self()
#PID<0.41.0>
iex> spawn fn -> send(parent, {:hello, self()}) end
#PID<0.48.0>
iex> receive do
...>   {:hello, pid} -> "Got hello from #{inspect pid}"
...> end
"Got 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.

```
iex> send self(), :hello
:hello
iex> flush()
:hello
:ok
```

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:

```
iex> spawn fn -> raise "oops" end
#PID<0.58.0>

[error] Process #PID<0.58.00> raised an exception
** (RuntimeError) oops
    :erlang.apply/2
```

It merely logged an error but the spawning process is still running. 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`:

```
iex> spawn_link fn -> raise "oops" end
#PID<0.41.0>

** (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
spawn_link fn -> raise "oops" end

receive do
  :hello -> "let's wait until the process fails"
end
```

```
$ elixir spawn.exs

** (EXIT from #PID<0.47.0>) an exception was raised:
    ** (RuntimeError) oops
        spawn.exs:1: anonymous fn/0 in :elixir_compiler_0.__FILE__/1
```

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/1`. We recommend that you take a look at the `Process` module for other functionality provided by processes.

Processes 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 dies and start a new process in its place. This is only possible because processes are isolated and don't share anything by default. And since 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 processes fail because we expect supervisors to properly restart our systems. "Failing fast" is a common philosophy when writing Elixir software!

`spawn/1` and `spawn_link/1` are the basic primitives for creating processes in Elixir. Although we have used them exclusively so far, most of the time we are going to use abstractions that build on top of them. Let's see the most common one, called tasks.

Tasks

Tasks build on top of the spawn functions to provide better error reports and introspection:

```
iex(1)> Task.start fn -> raise "oops" end
{:ok, #PID<0.55.0>}

15:22:33.046 [error] Task #PID<0.55.0> started from #PID<0.53.0> terminating
** (RuntimeError) oops
    (elixir) lib/task/supervised.ex:74: Task.Supervised.do_apply/2
    (stdlib) proc_lib.erl:239: :proc_lib.init_p_do_apply/3
Function: #Function<20.90072148/0 in :erl_eval.expr/5>
    Args: []
```

Instead of `spawn/1` and `spawn_link/1`, we use `Task.start/1` and `Task.start_link/1` to return `{:ok, pid}` rather than just the PID. This is what enables Tasks to be used in supervision trees. Furthermore, `Task` provides convenience functions, like `Task.async/1` and `Task.await/1`, and functionality to ease distribution.

We will explore those functionalities in the ***Mix and OTP guide***, for now it is enough to remember to use Task to get better error reports.

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 loop infinitely, maintain state, and send and receive 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`:

```
defmodule KV do
  def start_link do
    Task.start_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
end
```

Note that the `start_link` function starts a new process that runs the `loop/1` function, starting with an empty map. The `loop/1` function then waits for messages and performs the appropriate action for each message. In the 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 invokes `loop/1` 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`:

```
iex> {:ok, pid} = KV.start_link
#PID<0.62.0>
iex> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
nil
:ok
```

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:

```
iex> send pid, {:put, :hello, :world}
{:put, :hello, :world}
iex> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
:world
:ok
```

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:

```
iex> Process.register(pid, :kv)
true
iex> send :kv, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
:world
:ok
```

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

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

A `:name` option could also be given to `Agent.start_link/2` and it would be automatically registered. Besides agents, Elixir provides an API for building generic servers (called GenServer), 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.

IO and the file system

This chapter is a quick introduction to input/output mechanisms and file-system-related tasks, as well as to 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*.

The `IO` module

The `IO` module is the main mechanism in Elixir for reading and writing to standard input/output (`:stdio`), standard error (`:stderr`), files and other IO devices. Usage of the module is pretty straightforward:

```
iex> IO.puts "hello world"
hello world
:ok
iex> IO.gets "yes or no? "
yes or no? yes
"yes\n"
```

By default, functions in the IO module read from the standard input and write to the standard output. We can change that by passing, for example, `:stderr` as an argument (in order to write to the standard error device):

```
iex> IO.puts :stderr, "hello world"
hello world
:ok
```

The `File` module

The `File` module contains functions that allow 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:

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

A file can also be opened with `:utf8` encoding, which tells the `File` module to interpret the bytes read from the file as UTF-8-encoded bytes.

Besides functions for opening, reading and writing files, the `File` module has many functions to work with 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` to create directories and all their parent chain. There are even `File.cp_r/2` and `File.rm_rf/1` to respectively copy and remove files and directories recursively (i.e., copying and removing the contents of the directories too).

You will also notice that functions in the `File` module have two variants: one "regular" variant and another variant with a trailing bang (`!`). For example, when we read the `&quot;hello&quot;` file in the example above, we use `File.read/1` . Alternatively, we can use `File.read!/1` :

```
iex> File.read "hello"
{:ok, "world"}
iex> File.read! "hello"
"world"
iex> File.read "unknown"
{:error, :enoent}
iex> 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. The version without `!` is preferred when you want to handle different outcomes using pattern matching:

```
case File.read(file) do
  {:ok, body}      -> # do something with the `body`
  {:error, reason} -> # handle the error caused by `reason`
end
```

However, if you expect the file to be there, the bang variation is more useful as it raises a meaningful error message. Avoid writing:

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

as, in case of an error, `File.read/1` will return `{:error, reason}` and the pattern matching will fail. You will still get the desired result (a raised error), but the message will be about the pattern which doesn't match (thus being cryptic in respect to what the error actually is about).

Therefore, if you don't want to handle the error outcomes, prefer using `File.read!/1`.

The `Path` module

The majority of the functions in the `File` module expect paths as arguments. Most commonly, those paths will be regular binaries. The `Path` module provides facilities for working with such paths:

```
iex> Path.join("foo", "bar")
"foo/bar"
iex> Path.expand("~/hello")
"/Users/jose/hello"
```

Using functions from the `Path` module as opposed to just manipulating binaries is preferred since the `Path` module takes care of different operating systems transparently. Finally, keep in mind that Elixir will automatically convert slashes (`/`) into backslashes (`\`) on Windows when performing file operations.

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

Processes and group leaders

You may have noticed that `File.open/2` returns a tuple like `{:ok, pid}`:

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

That happens because the `IO` module actually works with processes (see chapter 11). When you write `IO.write(pid, binary)`, the `IO` module will send a message to the process identified by `pid` with the desired operation. Let's see what happens if we use our own process:

```
iex> pid = spawn fn ->
...>  receive do: (msg -> IO.inspect msg)
...> end
#PID<0.57.0>
iex> IO.write(pid, "hello")
{:io_request, #PID<0.41.0>, #Reference<0.0.8.91>, {:put_chars, :unicode, "hello"}}
** (ErlangError) erlang error: :terminated
```

After `IO.write/2`, we can see the request sent by the `IO` module (a four-elements tuple) printed out. Soon after that, we see that it 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 strings:

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

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

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

```
iex> IO.puts :stdio, "hello"
hello
:ok
iex> IO.puts Process.group_leader, "hello"
hello
:ok
```

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.

`iodata` and `chardata`

In all of the examples above, we used binaries 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:

```
iex> IO.puts 'hello world'
hello world
:ok
iex> IO.puts ['hello', ?\s, "world"]
hello world
:ok
```

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., they expect 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. Those functions expect a `char_data` as an argument, that is, a list of characters or strings.

Although this is a subtle difference, you only need to worry about these 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 of 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 `chardata` and `iodata` for IO operations.

alias, require and import

In order to facilitate software reuse, Elixir provides three directives (`alias` , `require` and `import`) plus a macro called `use` summarized below:

```
# Alias the module so it can be called as Bar instead of Foo.Bar
alias Foo.Bar, as: Bar

# Ensure the module is compiled and available (usually for macros)
require Foo

# Import functions from Foo so they can be called without the `Foo.` prefix
import Foo

# Invokes the custom code defined in Foo as an extension point
use Foo
```

We are going to explore them in detail now. Keep in mind the first three are called directives because they have **lexical scope**, while `use` is a common extension point.

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:

```
defmodule Math do
  alias Math.List, as: List
end
```

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 prefixing the module name with `Elixir.`:

```
List.flatten             #=> uses Math.List.flatten
Elixir.List.flatten      #=> uses List.flatten
Elixir.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:

```
alias Math.List
```

Is the same as:

```
alias Math.List, as: List
```

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

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

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

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.

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:

```
iex> Integer.is_odd(3)
** (CompileError) iex:1: you must require Integer before invoking the macro Integer.is
_odd/1
iex> require Integer
Integer
iex> Integer.is_odd(3)
true
```

In Elixir, `Integer.is_odd/1` is defined as a macro so that it can be used as a guard. This means that, in order to invoke `Integer.is_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.

import

We use `import` whenever we want to easily access functions or macros from other modules without using the fully-qualified name. For instance, if we want to use the `duplicate/2` function from the `List` module several times, we can simply import it:

```
iex> import List, only: [duplicate: 2]
List
iex> 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 in order to avoid importing all the functions of a given module inside the namespace. `:except` could also be given as an option in order to import everything in a module *except* a list of functions.

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

```
import Integer, only: :macros
```

Or to import all functions, you could write:

```
import Integer, only: :functions
```

Note that `import` is **lexically scoped** too. This means that we can import specific macros or functions inside function definitions:

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    duplicate(:ok, 10)
  end
end
```

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 `import`ing a module automatically `require`s it.

use

Although not a directive, `use` is a macro tightly related to `require` that allows you to use a module in the current context. The `use` macro is frequently used by developers to bring external functionality into the current lexical scope, often modules.

For example, in order to write tests using the ExUnit framework, a developer should use the `ExUnit.Case` module:

```
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    assert true
  end
end
```

Behind the scenes, `use` requires the given module and then calls the `__using__/1` callback on it allowing the module to inject some code into the current context. Generally speaking, the following module:

```
defmodule Example do
  use Feature, option: :value
end
```

is compiled into

```
defmodule Example do
  require Feature
  Feature.__using__(option: :value)
end
```

With this we have almost finished our tour of Elixir modules. The last topic to cover is module attributes.

Understanding Aliases

At this point, you may be wondering: what exactly is an Elixir alias and how is it 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 `:&quot;Elixir.String&quot;`:

```
iex> is_atom(String)
true
iex> to_string(String)
"Elixir.String"
iex> :"Elixir.String" == String
true
```

By using the `alias/2` directive, we are simply changing the atom the alias expands to.

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

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

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

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

We are simply calling the function `flatten` on the atom `:lists`.

Module nesting

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

```
defmodule Foo do
  defmodule Bar do
  end
end
```

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. The code above is exactly the same as:

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

If, later, the `Bar` module is moved outside the `Foo` module definition, it must be referenced by its full name (`Foo.Bar`) or an alias must be set using the `alias` directive discussed above.

Note: in Elixir, you don't have to define the `Foo` module before being able to define the `Foo.Bar` module, as the language translates all module names to atoms. You can define arbitrarily-nested modules without defining any module in the chain (e.g., `Foo.Bar.Baz` without defining `Foo` or `Foo.Bar` first).

As we will see in later chapters, aliases also play a crucial role in macros, to guarantee they are hygienic.

Multi alias/import/require/use

From Elixir v1.2, it is possible to alias, import or require multiple modules at once. This is particularly useful once we start nesting modules, which is very common when building Elixir applications. For example, imagine you have an application where all modules are nested under `MyApp`, you can alias the modules `MyApp.Foo`, `MyApp.Bar` and `MyApp.Baz` at once as follows:

```
alias MyApp.{Foo, Bar, Baz}
```

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.

As annotations

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

```
defmodule MyServer do
  @vsn 2
end
```

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. You can read more about writing documentation in Elixir in our official documentation.

Let's go back to the `Math` module defined in the previous chapters, add some documentation and save it to the `math.ex` file:

```
defmodule Math do
  @moduledoc """
  Provides math-related functions.

  ## Examples

      iex> Math.sum(1, 2)
      3

"""

  @doc """
  Calculates the sum of two numbers.
"""
  def sum(a, b), do: a + b
end
```

Elixir promotes the use of markdown with heredocs to write readable documentation. Heredocs are multiline strings, they start and end with triple double-quotes, keeping the formatting of the inner text. We can access the documentation of any compiled module directly from IEx:

```
$ elixirc math.ex
$ iex
```

```
iex> h Math # Access the docs for the module Math
...
iex> h Math.sum # Access the docs for the sum function
...
```

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.

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

As constants

Elixir developers will often use module attributes as constants:

```
defmodule MyServer do
  @initial_state %{host: "147.0.0.1", port: 3456}
  IO.inspect @initial_state
end
```

> 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:

```
defmodule MyServer do
  @unknown
end
warning: undefined module attribute @unknown, please remove access to @unknown or expl
icitly set it before access
```

Finally, attributes can also be read inside functions:

```
defmodule MyServer do
  @my_data 14
  def first_data, do: @my_data
  @my_data 13
  def second_data, do: @my_data
end

MyServer.first_data #=> 14
MyServer.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.

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:

```
defmodule 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
end

IO.puts "Running MyPlug with Cowboy on http://localhost:4000"
Plug.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 function (`call/2`) which handles http requests. This function 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 on how using module attributes as storage allows developers to create DSLs.

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

```
defmodule MyTest do
  use ExUnit.Case

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

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.

Structs

In chapter 7 we learned about maps:

```
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
iex> map[:a]
1
iex> %{map | a: 3}
%{a: 3, b: 2}
```

Structs are extensions built on top of maps that provide compile-time checks and default values.

Defining structs

To define a struct, the `defstruct` construct is used:

```
iex> defmodule User do
...>   defstruct name: "John", age: 27
...> end
```

The keyword list used with `defstruct` defines what fields the struct will have along with their default values.

Structs take the name of the module they're defined in. In the example above, we defined a struct named `User`.

We can now create `User` structs by using a syntax similar to the one used to create maps:

```
iex> %User{}
%User{age: 27, name: "John"}
iex> %User{name: "Meg"}
%User{age: 27, name: "Meg"}
```

Structs provide *compile-time* guarantees that only the fields (and *all* of them) defined through `defstruct` will be allowed to exist in a struct:

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

Accessing and updating structs

When we discussed maps, we showed how we can access and update the fields of a map. The same techniques (and the same syntax) apply to structs as well:

```
iex> john = %User{}
%User{age: 27, name: "John"}
iex> john.name
"John"
iex> meg = %{john | name: "Meg"}
%User{age: 27, name: "Meg"}
iex> %{meg | oops: :field}
** (KeyError) key :oops not found in: %User{age: 27, name: "Meg"}
```

When using the update syntax (`|`), the *VM* is aware that no new keys will be added to the struct, allowing the maps underneath to share their structure in memory. In the example above, both `john` and `meg` share the same key structure in memory.

Structs can also be used in pattern matching, both for matching on the value of specific keys as well as for ensuring that the matching value is a struct of the same type as the matched value.

```
iex> %User{name: name} = john
%User{age: 27, name: "John"}
iex> name
"John"
iex> %User{} = %{}
** (MatchError) no match of right hand side value: %{}
```

Structs are bare maps underneath

In the example above, pattern matching works because underneath structs are just bare maps with a fixed set of fields. As maps, structs store a "special" field named `__struct__` that holds the name of the struct:

```
iex> is_map(john)
true
iex> john.__struct__
User
```

Notice that we referred to structs as **bare** maps because none of the protocols implemented for maps are available for structs. For example, you can neither enumerate nor access a struct:

```
iex> john = %User{}
%User{age: 27, name: "John"}
iex> john[:name]
** (UndefinedFunctionError) undefined function: User.fetch/2
iex> Enum.each john, fn({field, value}) -> IO.puts(value) end
** (Protocol.UndefinedError) protocol Enumerable not implemented for %User{age: 27, na
me: "John"}
```

However, since structs are just maps, they work with the functions from the `Map` module:

```
iex> kurt = Map.put(%User{}, :name, "Kurt")
%User{age: 27, name: "Kurt"}
iex> Map.merge(kurt, %User{name: "Takashi"})
%User{age: 27, name: "Takashi"}
iex> Map.keys(john)
[:__struct__, :age, :name]
```

Structs alongside protocols provide one of the most important features for Elixir developers: data polymorphism. That's what we will explore in the next chapter.

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:

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

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
defimpl Blank, for: Integer do
  def blank?(_), do: false
end

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

# Just empty map is blank
defimpl 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
end

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

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:

```
iex> Blank.blank?(0)
false
iex> Blank.blank?([])
true
iex> Blank.blank?([1, 2, 3])
false
```

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

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

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 that chapter:

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

And then check:

```
iex> Blank.blank?(%{})
true
iex> Blank.blank?(%User{})
** (Protocol.UndefinedError) protocol Blank not implemented for %User{age: 27, name: "
john"}
```

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

```
defimpl Blank, for: User do
  def blank?(_), do: false
end
```

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 implement all relevant protocols, such as `Enumerable` and possibly `Blank`, for this data type.

Implementing `Any`

Manually implementing protocols for all types can quickly become repetitive and tedious. In such cases, Elixir provides two options: we can explicitly derive the protocol implementation for our types or automatically implement the protocol for all types. In both cases, we need to implement the protocol for `Any`.

Deriving

Elixir allows us to derive a protocol implementation based on the `Any` implementation. Let's first implement `Any` as follows:

```
defimpl Blank, for: Any do
  def blank?(_), do: false
end
```

Now, when defining the struct, we can explicitly derive the implementation for the `Blank` protocol. Let's create another struct, this one called `DeriveUser`:

```
defmodule DeriveUser do
  @derive Blank
  defstruct name: "john", age: 27
end
```

When deriving, Elixir will implement the `Blank` protocol for `DeriveUser` based on the implementation provided for `Any`. Note this behaviour is opt-in: structs will only work with the protocol as long as they explicitly implement or derive it.

Fallback to `Any`

Another alternative to `@derive` is to explicitly tell the protocol to fallback to `Any` when an implementation cannot be found. This can be achieved by setting `@fallback_to_any` to `true` in the protocol definition:

```
defprotocol Blank do
  @fallback_to_any true
  def blank?(data)
end
```

Assuming we have implemented `Any` as in the previous section:

```
defimpl Blank, for: Any do
  def blank?(_), do: false
end
```

Now all data types (including structs) that have not implemented the `Blank` protocol will be considered non-blank. In contrast to `@derive`, falling back to `Any` is opt-out: all data types get a pre-defined behaviour unless they provide their own implementation of the protocol. Which technique is best depends on the use case but, given Elixir developers prefer explicit over implicit, you may see many libraries pushing towards the `@derive` approach.

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:

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

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:

```
iex> to_string :hello
"hello"
```

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

```
iex> "age: #{25}"
"age: 25"
```

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

```
iex> tuple = {1, 2, 3}
{1, 2, 3}
iex> "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:

```
iex> "tuple: #{inspect tuple}"
"tuple: {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:

```
iex> {1, 2, 3}
{1, 2, 3}
iex> %User{}
%User{name: "john", 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:

```
iex> 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.

Comprehensions

In Elixir, it is common to loop over an Enumerable, often filtering out some results and mapping values into another list. Comprehensions are syntactic sugar for such constructs: they group those common tasks into the `for` special form.

For example, we can map a list of integers into their squared values:

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

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

Protocol consolidation

When working with Elixir projects, using the Mix build tool, you may see output as follows:

```
Consolidated String.Chars
Consolidated Collectable
Consolidated List.Chars
Consolidated IEx.Info
Consolidated Enumerable
Consolidated Inspect
```

Those are all protocols that ship with Elixir and they are being consolidated. Because a protocol can dispatch to any data type, the protocol must check on every call if an implementation for the given type exists. This may be expensive.

However, after our project is compiled using a tool like Mix, we know all modules that have been defined, including protocols and their implementations. This way, the protocol can be consolidated into a very simple and fast dispatch module.

From Elixir v1.2, protocol consolidation happens automatically for all projects. We will build our own project in the ***Mix and OTP guide***.

Generators and filters

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

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

Generator expressions also support pattern matching on their left-hand side; all non-matching patterns are *ignored*. 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 compute the square of the `:good` values:

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

Alternatively to pattern matching, filters can be used to select some particular elements. For example, we can select the multiples of 3 and discard all others:

```
iex> multiple_of_3? = fn(n) -> rem(n, 3) == 0 end
iex> for n <- 0..5, multiple_of_3?.(n), do: n * n
[0, 9]
```

Comprehensions discard all elements for which the filter expression returns `false` or `nil`; all other values are selected.

Comprehensions generally 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 gets the size of each file in those directories:

```
for dir  <- dirs,
    file <- File.ls!(dir),
    path = Path.join(dir, file),
    File.regular?(path) do
  File.stat!(path).size
end
```

Multiple generators can also be used to calculate the cartesian product of two lists:

```
iex> for i <- [:a, :b, :c], j <- [1, 2], do:  {i, j}
[a: 1, a: 2, b: 1, b: 2, c: 1, c: 2]
```

A more advanced example of multiple generators and filters is Pythagorean triples. A Pythagorean triple is a set of positive integers such that `a*a + b*b = c*c`, let's write a comprehension in a file named `triple.exs`:

```
defmodule Triple do
  def pythagorean(n) when n > 0 do
    for a <- 1..n,
        b <- 1..n,
        c <- 1..n,
        a + b + c <= n,
        a*a + b*b == c*c,
        do: {a, b, c}
  end
end
```

Now on terminal:

```
iex triple.exs
```

```
iex> Triple.pythagorean(5)
[]
iex> Triple.pythagorean(12)
[{3, 4, 5}, {4, 3, 5}]
iex> Triple.pythagorean(48)
[{3, 4, 5}, {4, 3, 5}, {5, 12, 13}, {6, 8, 10}, {8, 6, 10}, {8, 15, 17},
 {9, 12, 15}, {12, 5, 13}, {12, 9, 15}, {12, 16, 20}, {15, 8, 17}, {16, 12, 20}]
```

The code above is quite expensive when the range of search is a large number. Additionally, since the tuple `{b, a, c}` represents the same Pythagorean triple as `{a, b, c}`, our function yields duplicate triples. We can optimize the comprehension and eliminate the duplicate results by referencing the variables from previous generators in the following ones, for example:

```
defmodule Triple do
  def pythagorean(n) when n > 0 do
    for a <- 1..n-2,
        b <- a+1..n-1,
        c <- b+1..n,
        a + b + c <= n,
        a*a + b*b == c*c,
        do: {a, b, c}
  end
end
```

Finally, 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.

Bitstring generators

Bitstring generators are also supported and are very useful when you need to comprehend over bitstring streams. The example below receives a list of pixels from a binary with their respective red, green and blue values and converts them into tuples of three elements each:

```
iex> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
iex> 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 "regular" enumerable generators, and supports filters as well.

The `:into` option

In the examples above, all the comprehensions returned lists as their result. However, the result of a comprehension can be inserted into different data structures by passing the `:into` option to the comprehension.

For example, a bitstring generator can be used with the `:into` option in order to easily remove all spaces in a string:

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

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

A common use case of `:into` can be transforming values in a map, without touching the keys:

```
iex> for {key, val} <- %{"a" => 1, "b" => 2}, into: %{}, do: {key, val * val}
%{"a" => 1, "b" => 4}
```

Let's make another example using streams. Since the `IO` module provides streams (that are both `Enumerable`s and `Collectable`s), an echo terminal that echoes back the upcased version of whatever is typed can be implemented using comprehensions:

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

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

Sigils

We have already learned that 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, for example, are mostly created via the `:atom` representation.

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

In this chapter, we are going to explore sigils, which are one of the mechanisms provided by the language for working with textual representations. Sigils start with the tilde (`~`) character which is followed by a letter (which identifies the sigil) and then a delimiter; optionally, modifiers can be added after the final delimiter.

Strings, char lists and words sigils

Besides regular expressions, Elixir ships with three other sigils.

Strings

The `~s` sigil is used to generate strings, like double quotes are. The `~s` sigil is useful, for example, when a string contains both double and single quotes:

```
iex> ~s(this is a string with "double" quotes, not 'single' ones)
"this is a string with \"double\" quotes, not 'single' ones"
```

Char lists

The `~c` sigil is used to generate char lists:

```
iex> ~c(this is a char list containing 'single quotes')
'this is a char list containing \'single quotes\''
```

Word lists

The `~w` sigil is used to generate lists of words (*words* are just regular strings). Inside the `~w` sigil, words are separated by whitespace.

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

The `~w` sigil also accepts the `c`, `s` and `a` modifiers (for char lists, strings and atoms, respectively), which specify the data type of the elements of the resulting list:

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

Regular expressions

The most common sigil in Elixir is `~r`, which is used to create regular expressions:

```
# A regular expression that matches strings which contain "foo" or "bar":
iex> regex = ~r/foo|bar/
~r/foo|bar/
iex> "foo" =~ regex
true
iex> "bat" =~ regex
false
```

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:

```
iex> "HELLO" =~ ~r/hello/
false
iex> "HELLO" =~ ~r/hello/i
true
```

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 delimiters:

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

The reason behind supporting different delimiters is that different delimiters can be more suited for 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.

Interpolation and escaping in sigils

Besides lowercase sigils, Elixir supports uppercase sigils to deal with escaping characters and interpolation. While both `~s` and `~S` will return strings, the former allows escape codes and interpolation while the latter does not:

```
iex> ~s(String with escape codes \x26 #{"inter" <> "polation"})
"String with escape codes & interpolation"
iex> ~S(String without escape codes \x26 without #{interpolation})
"String without escape codes \\x26 without \#{interpolation}"
```

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

- `\&quot;` – double quote
- `\&#039;` – 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
- `\0` - null byte
- `\xDD` - represents a single byte in hexadecimal (such as `\x13`)
- `\uDDDD` and `\u{D...}` - represents a Unicode codepoint in hexadecimal (such as `\u{1F600}`)

Sigils also support heredocs, that is, triple double- or single-quotes as separators:

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

The most common use case for heredoc sigils is when writing documentation. For example, writing escape characters in documentation would soon become error prone because of the need to double-escape some characters:

```
@doc """
Converts double-quotes to single-quotes.

## Examples

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

"""
def convert(...)
```

By using `~S`, this problem can be avoided altogether:

```
@doc ~S"""
Converts double-quotes to single-quotes.

## Examples

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

"""
def convert(...)
```

Custom sigils

As hinted at the beginning of this chapter, sigils in Elixir are extensible. In fact, using the sigil `~r/foo/i` is equivalent to calling `sigil_r` with a binary and a char list as argument:

```
iex> sigil_r(<<"foo">>, 'i')
~r"foo"i
```

We can access the documentation for the `~r` sigil via `sigil_r`:

```
iex> h sigil_r
...
```

We can also provide our own sigils by simply implementing functions that follow the `sigil_{identifier}` pattern. For example, let's implement the `~i` sigil that returns an integer (with the optional `n` modifier to make it negative):

```
iex> defmodule MySigils do
...>   def sigil_i(string, []), do: String.to_integer(string)
...>   def sigil_i(string, [?n]), do: -String.to_integer(string)
...> end
iex> import MySigils
iex> ~i(13)
13
iex> ~i(42)n
-42
```

Sigils can also be used to do compile-time work with the help of macros. For example, regular expressions in Elixir are compiled into an efficient representation during compilation of the source code, therefore skipping this step at runtime. If you're interested in the subject, we recommend you learn more about macros and check out how sigils are implemented in the `Kernel` module (where the `sigil_*` functions are defined).

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.

Errors

Errors (or *exceptions*) are used when exceptional things happen in the code. A sample error can be retrieved by trying to add a number into an atom:

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

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

```
iex> raise "oops"
** (RuntimeError) oops
```

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

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

You can also define your own errors by creating a module and using the `defexception` construct inside it; this way, you'll create an error with the same name as the module it's defined in. The most common case is to define a custom exception with a message field:

```
iex> defmodule MyError do
iex>   defexception message: "default message"
iex> end
iex> raise MyError
** (MyError) default message
iex> raise MyError, message: "custom message"
** (MyError) custom message
```

Errors can be **rescued** using the `try/rescue` construct:

```
iex> try do
...>   raise "oops"
...> rescue
...>   e in RuntimeError -> e
...> end
%RuntimeError{message: "oops"}
```

The example above rescues the runtime error and returns the error itself which is then printed in the `iex` session.

If you don't have any use for the error, you don't have to provide it:

```
iex> try do
...>   raise "oops"
...> rescue
...>   RuntimeError -> "Error!"
...> end
"Error!"
```

In practice, however, Elixir developers rarely use the `try/rescue` construct. For example, many languages would force you to rescue an error when a file cannot be opened successfully. Elixir instead provides a `File.read/1` function which returns a tuple containing information about whether the file was opened successfully:

```
iex> File.read "hello"
{:error, :enoent}
iex> File.write "hello", "world"
:ok
iex> File.read "hello"
{:ok, "world"}
```

There is no `try/rescue` here. In case you want to handle multiple outcomes of opening a file, you can simply use pattern matching with the `case` construct:

```
iex> case File.read "hello" do
...>   {:ok, body}      -> IO.puts "Success: #{body}"
...>   {:error, reason} -> IO.puts "Error: #{reason}"
...> end
```

At the end of the day, it's 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, it leaves 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 that file is truly an *error*) you can simply use `File.read!/1`:

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

Many functions in the standard library follow the pattern of having a counterpart that raises an exception instead of returning tuples to match against. The convention is to create a function (`foo`) which returns `{:ok, result}` or `{:error, reason}` tuples and another function (`foo!` , same name but with a trailing `!`) that takes the same arguments as `foo` but which raises an exception if there's an error. `foo!` should return the result (not wrapped in a tuple) if everything goes fine. The `File` module is a good example of this convention.

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

Throws

In Elixir, a value can be thrown and later be caught. `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 except when interfacing with libraries that do not provide a proper API. For example, let's imagine the `Enum` module did not provide any API for finding a value and that we needed to find the first multiple of 13 in a list of numbers:

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

Since `Enum` *does* provide a proper API, in practice `Enum.find/2` is the way to go:

```
iex> Enum.find -50..50, &(rem(&1, 13) == 0)
-39
```

Exits

All Elixir code runs inside processes that communicate with each other. When a process dies of "natural causes" (e.g., unhandled exceptions), it sends an `exit` signal. A process can also die by explicitly sending an exit signal:

```
iex> spawn_link fn -> exit(1) end
#PID<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` :

```
iex> try do
...>   exit "I am exiting"
...> catch
...>   :exit, _ -> "not really"
...> end
"not 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 from 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 an 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.

After

Sometimes it's necessary to ensure that a resource is cleaned up after some action that could potentially raise an error. The `try/after` construct allows you to do that. For example, we can open a file and use an `after` clause to close it--even if something goes wrong:

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

The `after` clause will be executed regardless of whether or not the tried block succeeds. Note, however, that if a linked process exits, this process will exit and the `after` clause will not get run. Thus `after` provides only a soft guarantee. Luckily, files in Elixir are also linked to the current processes and therefore they will always get closed if the current process crashes, independent of the `after` clause. You will find the same to be true for other resources like ETS tables, sockets, ports and more.

Sometimes you may want to wrap the entire body of a function in a `try` construct, often to guarantee some code will be executed afterwards. In such cases, Elixir allows you to omit the `try` line:

```
iex> defmodule RunAfter do
...>   def without_even_trying do
...>     raise "oops"
...>   after
...>     IO.puts "cleaning up!"
...>   end
...> end
iex> RunAfter.without_even_trying
cleaning up!
** (RuntimeError) oops
```

Elixir will automatically wrap the function body in a `try` whenever one of `after` , `rescue` or `catch` is specified.

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:

```
iex> try do
...>   raise "fail"
...>   what_happened = :did_not_raise
...> rescue
...>   _ -> what_happened = :rescued
...> end
iex> what_happened
** (RuntimeError) undefined function: what_happened/0
```

Instead, you can store the value of the `try` expression:

```
iex> what_happened =
...>   try do
...>     raise "fail"
...>     :did_not_raise
...>   rescue
...>     _ -> :rescued
...>   end
iex> what_happened
:rescued
```

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".

Typespecs and behaviours

Types and specs

Elixir is a dynamically typed language, so all types in Elixir are inferred by the runtime. Nonetheless, Elixir comes with **typespecs**, which are a notation used for:

1. declaring custom data types;
2. declaring typed function signatures (specifications).

Function specifications

By default, Elixir provides some basic types, such as `integer` or `pid`, as well as more complex types: for example, the `round/1` function, which rounds a float to its nearest integer, takes a `number` as an argument (an `integer` or a `float`) and returns an `integer`. As you can see in its documentation, `round/1`'s typed signature is written as:

```
round(number) :: integer
```

`::` means that the function on the left side *returns* a value whose type is what's on the right side. Function specs are written with the `@spec` directive, placed right before the function definition. The `round/1` function could be written as:

```
@spec round(number) :: integer
def round(number), do: # implementation...
```

Elixir supports compound types as well. For example, a list of integers has type `[integer]`. You can see all the built-in types provided by Elixir in the typespecs docs.

Defining custom types

While Elixir provides a lot of useful built-in types, it's convenient to define custom types when appropriate. This can be done when defining modules through the `@type` directive.

Say we have a `LousyCalculator` module, which performs the usual arithmetic operations (sum, product and so on) but, instead of returning numbers, it returns tuples with the result of an operation as the first element and a random remark as the second element.

```
defmodule LousyCalculator do
  @spec add(number, number) :: {number, String.t}
  def add(x, y), do: {x + y, "You need a calculator to do that?!"}

  @spec multiply(number, number) :: {number, String.t}
  def multiply(x, y), do: {x * y, "Jeez, come on!"}
end
```

As you can see in the example, tuples are a compound type and each tuple is identified by the types inside it. To understand why `String.t` is not written as `string`, have another look at the notes in the typespecs docs.

Defining function specs this way works, but it quickly becomes annoying since we're repeating the type `{number, String.t}` over and over. We can use the `@type` directive in order to declare our own custom type.

```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_remark :: {number, String.t}

  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to do that?"}

  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition on steroids."}
end
```

The `@typedoc` directive, similarly to the `@doc` and `@moduledoc` directives, is used to document custom types.

Custom types defined through `@type` are exported and available outside the module they're defined in:

```
defmodule QuietCalculator do
  @spec add(number, number) :: number
  def add(x, y), do: make_quiet(LousyCalculator.add(x, y))

  @spec make_quiet(LousyCalculator.number_with_remark) :: number
  defp make_quiet({num, _remark}), do: num
end
```

If you want to keep a custom type private, you can use the `@typep` directive instead of `@type`.

Static code analysis

Typespecs are not only useful to developers and as additional documentation. The Erlang tool Dialyzer, for example, uses typespecs in order to perform static analysis of code. That's why, in the `QuietCalculator` example, we wrote a spec for the `make_quiet/1` function even if it was defined as a private function.

Behaviours

Many modules share the same public API. Take a look at Plug, which, as its description states, is a **specification** for composable modules in web applications. Each *plug* is a module which **has to** implement at least two public functions: `init/1` and `call/2`.

Behaviours provide a way to:

- define a set of functions that have to be implemented by a module;
- ensure that a module implements all the functions in that set.

If you have to, you can think of behaviours like interfaces in object oriented languages like Java: a set of function signatures that a module has to implement.

Defining behaviours

Say we want to implement a bunch of parsers, each parsing structured data: for example, a JSON parser and a YAML parser. Each of these two parsers will *behave* the same way: both will provide a `parse/1` function and an `extensions/0` function. The `parse/1` function will return an Elixir representation of the structured data, while the `extensions/0` function will return a list of file extensions that can be used for each type of data (e.g., `.json` for JSON files).

We can create a `Parser` behaviour:

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

Modules adopting the `Parser` behaviour will have to implement all the functions defined with the `@callback` directive. As you can see, `@callback` expects a function name but also a function specification like the ones used with the `@spec` directive we saw above.

Adopting behaviours

Adopting a behaviour is straightforward:

```
defmodule JSONParser do
  @behaviour Parser

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

```
defmodule YAMLParser do
  @behaviour Parser

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

If a module adopting a given behaviour doesn't implement one of the callbacks required by that behaviour, a compile-time warning will be generated.

Erlang libraries

Elixir provides excellent interoperability with Erlang libraries. In fact, Elixir discourages simply wrapping Erlang libraries in favor of directly interfacing with Erlang code. In this section we will present some of the most common and useful Erlang functionality that is not found in Elixir.

As you grow more proficient in Elixir, you may want to explore the Erlang STDLIB Reference Manual in more detail.

The binary module

The built-in Elixir String module handles binaries that are UTF-8 encoded. The binary module is useful when you are dealing with binary data that is not necessarily UTF-8 encoded.

```
iex> String.to_char_list "Ø"
[216]
iex> :binary.bin_to_list "Ø"
[195, 152]
```

The above example shows the difference; the `String` module returns UTF-8 codepoints, while `:binary` deals with raw data bytes.

Formatted text output

Elixir does not contain a function similar to `printf` found in C and other languages. Luckily, the Erlang standard library functions `:io.format/2` and `:io_lib.format/2` may be used. The first formats to terminal output, while the second formats to an iolist. The format specifiers differ from `printf`, refer to the Erlang documentation for details.

```
iex> :io.format("Pi is approximately given by:~10.3f~n", [:math.pi])
Pi is approximately given by:     3.142
:ok
iex> to_string :io_lib.format("Pi is approximately given by:~10.3f~n", [:math.pi])
"Pi is approximately given by:     3.142\n"
```

Also note that Erlang's formatting functions require special attention to Unicode handling.

The crypto module

The crypto module contains hashing functions, digital signatures, encryption and more:

```
iex> Base.encode16(:crypto.hash(:sha256, "Elixir"))
"3315715A7A3AD57428298676C5AE465DADA38D951BDFAC9348A8A31E9C7401CB"
```

The `:crypto` module is not part of the Erlang standard library, but is included with the Erlang distribution. This means you must list `:crypto` in your project's applications list whenever you use it. To do this, edit your `mix.exs` file to include:

```
  def application do
    [applications: [:crypto]]
  end
```

The digraph module

The digraph module (as well as digraph_utils) contains functions for dealing with directed graphs built of vertices and edges. After constructing the graph, the algorithms in there will help finding for instance the shortest path between two vertices, or loops in the graph.

Note that the functions in `:digraph` alter the graph structure indirectly as a side effect, while returning the added vertices or edges.

Given three vertices, find the shortest path from the first to the last.

```
iex> digraph = :digraph.new()
iex> coords = [{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}]
iex> [v0, v1, v2] = (for c <- coords, do: :digraph.add_vertex(digraph, c))
iex> :digraph.add_edge(digraph, v0, v1)
iex> :digraph.add_edge(digraph, v1, v2)
iex> :digraph.get_short_path(digraph, v0, v2)
[{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}]
```

Erlang Term Storage

The modules `ets` and `dets` handle storage of large data structures in memory or on disk respectively.

ETS lets you create a table containing tuples. By default, ETS tables are protected, which means only the owner process may write to the table but any other process can read. ETS has some functionality to be used as a simple database, a key-value store or as a cache mechanism.

The functions in the `ets` module will modify the state of the table as a side-effect.

```
iex> table = :ets.new(:ets_test, [])
# Store as tuples with {name, population}
iex> :ets.insert(table, {"China", 1_374_000_000})
iex> :ets.insert(table, {"India", 1_284_000_000})
iex> :ets.insert(table, {"USA", 322_000_000})
iex> :ets.i(table)
<1   > {"USA", 322000000}
<2   > {"China", 1_374_000_000}
<3   > {"India", 1_284_000_000}
```

The math module

The `math` module contains common mathematical operations covering trigonometry, exponential and logarithmic functions.

```
iex> angle_45_deg = :math.pi() * 45.0 / 180.0
iex> :math.sin(angle_45_deg)
0.7071067811865475
iex> :math.exp(55.0)
7.694785265142018e23
iex> :math.log(7.694785265142018e23)
55.0
```

The queue module

The `queue` is a data structure that implements (double-ended) FIFO (first-in first-out) queues efficiently:

```
iex> q = :queue.new
iex> q = :queue.in("A", q)
iex> q = :queue.in("B", q)
iex> {value, q} = :queue.out(q)
iex> value
{:value, "A"}
iex> {value, q} = :queue.out(q)
iex> value
{:value, "B"}
iex> {value, q} = :queue.out(q)
iex> value
:empty
```

The rand module

`rand` has functions for returning random values and setting the random seed.

```
iex> :rand.uniform()
0.8175669086010815
iex> _ = :rand.seed(:exs1024, {123, 123534, 345345})
iex> :rand.uniform()
0.5820506340260994
iex> :rand.uniform(6)
6
```

The zip and zlib modules

The `zip` module lets you read and write zip files to and from disk or memory, as well as extracting file information.

This code counts the number of files in a zip file:

```
iex> :zip.foldl(fn _, _, _, acc -> acc + 1 end, 0, :binary.bin_to_list("file.zip"))
{:ok, 633}
```

The `zlib` module deals with data compression in zlib format, as found in the `gzip` command.

```
iex> song = "
...> Mary had a little lamb,
...> His fleece was white as snow,
...> And everywhere that Mary went,
...> The lamb was sure to go."
iex> compressed = :zlib.compress(song)
iex> byte_size song
110
iex> byte_size compressed
99
iex> :zlib.uncompress(compressed)
"\nMary had a little lamb,\nHis fleece was white as snow,\nAnd everywhere that Mary we
nt,\nThe lamb was sure to go."
```

Where to go next

Eager to learn more? Keep reading!

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

Meta-programming

Elixir is an extensible and very customizable programming language thanks to its meta-programming support. Most meta-programming in Elixir is done through macros, which are very useful in several situations, especially for writing DSLs. We have written a short guide that explains the basic mechanisms behind macros, shows how to write macros, and how to use macros to create DSLs:

- Meta-programming in Elixir

A byte of Erlang

Elixir runs on the Erlang Virtual Machine and, sooner or 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 Erlang's syntax but also review some of the things you have learned in this 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. 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.

Community and other resources

We have a Learning section that suggests books, screencasts and other resources for learning Elixir and exploring the ecosystem. There are also 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. To keep posted on the latest news and announcements, follow the blog and follow the language development on the elixir-core mailing list.

Don't forget that 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.