Community Experience Distilled

Unveil the many hidden gems of programming functionally by taking the foundational steps with Elixir

Learning Elixir

Kenny Ballou

Elixir, based on Erlang’s virtual machine and ecosystem, makes it easier to achieve the scalability, concurrency, fault tolerance, and high availability goals that are pursued by developers using any programming language or programming paradigm. Elixir is a modern programming language that utilizes the benefits offered by Erlang VM without really incorporating the complex syntaxes of Erlang.

Learning Elixir will teach many things that are beneficial to programming as a craft, even if at the end of the day you aren’t using Elixir. This book will teach you concepts and principles important to any complex, scalable, and resilient application. Applications are historically difficult to reason about, but using the concepts in this book, they will become easy and enjoyable. It will show you the functional programming ropes to enable you to create better and more scalable applications, and you will explore how Elixir can help you reach new programming heights. Furthermore, you will learn the basics of metaprogramming: modifying and extending Elixir to suit your needs.

Who this book is written for

This book targets developers new to Elixir and Erlang, in order to make them feel comfortable in functional programming with Elixir. Although no knowledge of Erlang is assumed, some programming experience with mainstream object-oriented programming languages such as Ruby, Python, Java, and C# would be beneficial.

What you will learn from this book

- Explore Elixir to create resilient, scalable applications
- Create fault-tolerant applications
- Become better acquainted with Elixir code and see how it is structured to build and develop functional programs
- Gain an understanding of effective OTP principles
- Design and program distributed applications and systems
- Write and create branching statements in Elixir
- Learn to do more with less using Elixir’s metaprogramming
- Become familiar with the facilities Elixir provides for metaprogramming, macros, and extending the Elixir language

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 ‘Modules and Functions – Creating Functional Building Blocks’
- A synopsis of the book’s content
- More information on Learning Elixir
Kenny Ballou is a life-long learner, developer, mathematician, and overall thinker. He enjoys solving problems, learning about technologies, and discussing new and different ideas.

He graduated from the Boise State University, majoring in applied mathematics, and minoring in computer science. He has been programming professionally for nearly a decade, and is currently working as a software developer for an analytics company in the Boise, Idaho area.

Apart from developing professionally, he is active in the open source community, contributing where he can.

When he is not developing, he enjoys reading, learning, and shredding the local mountains.

You can read more from him on his blog at https://kennyballou.com, check out his code on GitHub at https://github.com/kennyballou/, and follow him on Twitter at @kennyballou.
Preface

This is an introduction to Elixir, a relatively new programming language. We will go through the basics of the language and teach functional programming and other paradigms as we progress. We will study the basics of concurrent and distributed programming through the lens of Elixir and OTP. We will also examine one of the more exciting features of Elixir: metaprogramming, or writing code that writes code.

This book doesn't teach the basics or programming in your first language, but it does teach the basics of Elixir under the assumption that this is a new language.

This book will also discuss a fair amount of Erlang, the predecessor language Elixir derives from, and the runtime Elixir compiles to, which is prevalent to the understanding of Elixir.

What this book covers

Chapter 1, Introducing Elixir – Thinking Functionally, introduces Elixir and functional programming, and provides some of the history and justification of Elixir. It also walks you through installing Elixir.

Chapter 2, Elixir Basics – Foundational Steps toward Functional Programming, introduces the basics of Elixir and its types, syntax, and semantics. This chapter lets you start reading and writing Elixir code.

Chapter 3, Modules and Functions – Creating Functional Building Blocks, lets us extend and expand on the previous chapter by introducing the basics of Elixir code organization into modules and functions. It makes us start our lengthy discussion on pattern matching, one of the coolest features of Elixir.

Chapter 4, Collections and Stream Processing, lets us examine collections and explains how to solve common problems using recursive algorithms. This chapter also lets us introduce Elixir's pipe operator and the basis of collection processing.
Chapter 5, Control Flow – Occasionally You Need to Branch, discusses how to do more traditional code branching, conditional statements using Elixir.

Chapter 6, Concurrent Programming – Using Processes to Conquer Concurrency, explains how to write concurrent code using Elixir. It introduces Elixir processes and the basics of message passing.

Chapter 7, OTP – A Poor Name for a Rich Framework, continues the discussion of concurrent programming with Elixir, more specifically in the context of OTP, which is the framework introduced in Erlang for building robust distributed applications.

Chapter 8, Distributed Elixir – Taking Concurrency to the Next Node, examines how to write Elixir that executes on multiple nodes, distributing the processing over possibly many computers.

Chapter 9, Metaprogramming – Doing More with Less, introduces Elixir behaviours, protocols, typespecs, and macros. Using Elixir macros, we examine how we can accomplish more with less code.
In the previous chapter, we introduced the basic, built-in types of Elixir.

In this chapter, we are going to dive further into functions and properly introduce modules. Furthermore, we are going to continue our discussion on pattern matching.

And, we will now get into some real, nonelementary examples that show how we can solve problems with what we have learned so far.

**Modules**

If you're familiar with other languages such as Python, modules aren't really a new concept. They define a set of functions and essentially namespace these functions from others. This avoids name conflicts and introduces a level of plugability and reusability throughout. Elixir modules, similarly, follow suit.

A module in Elixir defines a set of public and private functions that can either be used externally or internally. Modules in Elixir are defined with the `defmodule name do` block `end` construct. In fact, the simplest module we can define is the following:

```elixir
defmodule Foobar do end```

Of course, this is a highly uninteresting module, but we can define it. In fact, we can even define it in our interactive session:

```elixir
iex>(1) defmodule Foo do end
{:module, Foo,
 <<70, 79, 82, 49, 0, 0, 3, 136, 66, 69, 65, 77, 69, 120, 68, 99,
  0, 0, 0, 60, 131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105,
  114, 95, 100, 111, 99, 115, 95, 118, 49, 108, 0, 0, 0, 2, 104, 2,
  ...>>,
 nil}
```

Note that your output may vary slightly.

We’ve seen this output before, but we have never truly examined it. What is all this?

Well, as you might have already guessed reading the preceding output, it is the VM’s internal representation of the module we have just defined. Moreover, it’s defined in types we have already seen; we have a tuple whose first element is an atom, `:module`; second, the name of the module; third, a binary that defines the module; and fourth is `nil` since the module does not export any public functions.

Notably, this isn’t an entirely interesting or useful module though. For a more interesting module, we will need functions.

### Anonymous functions

In the previous chapter, we saw that functions are types and first-class citizens in Elixir. We even saw how to define some functions.

Anonymous functions are similar to regular functions except that they are not bound to an identifier.

Furthermore, there are actually two different syntaxes to define anonymous functions. This includes the syntax we saw earlier—`fn () -> block end`. But there is also a shorter syntax variant—`&{block}`. Let’s dive into some examples of both of these syntaxes and general function definitions.

As we saw in the previous chapter, we had the following:

```elixir
iex(1)> square = fn x -> x * x end
#Function<6.90072148/1 in :erl_eval.expr/5>
```
Then we could use it with the following:

```elixir
iex(2) square.(2)
4
```

So what are we doing here (at (1))? We are defining a function and binding it to the name `square`. The function takes a single variable (call it `x`), and we return the expression, `x * x`, or, after evaluation, the square of `x`.

The `#<Function<6.90072148/1 in :erl_eval.expr/5>>`, after we define the function, is the result of our expression evaluated by the Elixir compiler (where does the number come from?). This return value also tells us something about the function we have defined—it was in-lined by the compiler. Our function is actually translated to Erlang and then compiled by Erlang’s compiler, `erlc`, to generate the currently loaded bytecode for our simple `square` function.

Furthermore, we can have, arbitrarily, many arguments, or none at all, if we want:

```elixir
iex(1)> (fn x, y, z -> x + y * z end).(2, 6, 8)
50
```

We also did another trick here, we defined our function and then immediately executed it with the parameters 2, 6, and 8.

We can do the same thing too with the shorter syntax:

```elixir
iex(2)> (&(&1 + &2 * &3)).(2, 6, 8)
50
```

And just to prove that the two syntaxes are the same, we can do the following:

```elixir
iex(3)> square1 = fn x -> x * x end
#<Function<6.90072148/1 in :erl_eval.expr/5>>
iex(4)> square2 = &(&1 * &1)
#<Function<6.90072148/1 in :erl_eval.expr/5>>
```

With the shorter syntax, we gain brevity at the cost of named parameters. We can still have as many parameters as we want, we just have to know the order. For most small functions, this order is fine; however, if the anonymous function is long, you may want to opt for the former, longer syntax.
Pattern matching

Like many things in Elixir, anonymous functions support pattern matching! We can define quick functions with the same awesome power of pattern matching as regular functions and binding. For example, we may define a function like the following:

```elixir
iex(1)> area = fn {:circle, r} ->
...   (1)> 3.14159 * r * r
...   (1)> {:rect, w, h} ->
...   (1)> w * h
...   (1)> end
#Function<6.90072148/1 in :erl_eval.expr/5>
```

Here, we define an `area` function that can compute the area of either a circle or a rectangle by matching on a tuple with an atom of the shape type as the first element and either the radius or the width and height as the rest of the elements.

We could use this anonymous function like this:

```elixir
iex(2)> area.({:circle, 5})
78.53975
iex(3)> area.({:rect, 5, 5})
25
iex(4)> area.({:triangle, 5, 5})
** (FunctionClauseError) no function clause matching in :erl_eval."-inside-an-interpreted-fun-/1
```

Finally, since we didn't define a function for the area of a triangle, we get a match error when we attempt to compute the area for a triangle in (4).

Similarly, we would see a similar error if we attempted to compute the area of a rectangle when only passing the width:

```elixir
iex(5)> area.({:rect, 5})
** (FunctionClauseError) no function clause matching in :erl_eval."-inside-an-interpreted-fun-/1
```

Named functions

Named functions, unlike anonymous functions, require a module for definition. That is, to define a named function we must define the function inside a module.
Here, we combine what we learned about modules and anonymous functions a bit, and we define our square function again, though, this time, we define it inside a module named `MyMath`. Go ahead and create a file called `mymath.exs` and put the following code into it:

```elixir
defmodule MyMath do
  def square(x) do
    x * x
  end
end
```

Here, we are simply defining a function, `square`, which takes a single element, and returns the result of `x * x`. This really looks not much different from our previous versions except for being defined inside a module.

How do we run this module and see whether it works? Well, you might have tried `$ elixir mymath.exs` but that probably didn't do anything interesting...

The answer lies in importing the module in an interactive session. First, make sure your working directory is the same as the directory you saved `mymath.exs`. Then, from that directory, launch `iex`. From `iex`, we will use the `import_file/1`:

```elixir
iex(1)> import_file("mymath.exs")
{:module, MyMath,
 <<70, 79, 82, 49, 0, 0, 4, 160, 66, 69, 65, 77, 69, 120, 68, 99, 0,
  0, 0, 113, 131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95,
  100, 111, 99, 115, 95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
  {:square, 1}}
```

Here, we see again the returned tuple, similar to the last time. Here, however, instead of the last element being `nil`, we get a tuple with the elements `:square` and `1`. This is saying that the module exports a single public function, `square/1`, which is good because that's what we wanted!

Now, we can use our newly defined `square/1` function:

```elixir
iex(2)> MyMath.square(4)
16
```

This is not very different from other languages you may be used to. To invoke a function from a different namespace than the current, you prefix the call with the function's module name.
Private functions

All of the functions we have defined so far are public. That is, they are accessible from outside modules; there is nothing to stop other modules from calling these functions, which, for the most part, we want. But how do we define a function that is internal and should never be directly called from outside?

We can do it the same way we define public functions. We use the `defp` construct, though.

For a simplistic example, we could define a function in a module and then have a private function actually perform the named function's work:

```elixir
defmodule MyMath do
  def square(x) do
    do_square(x)
  end

  defp do_square(x), do: x * x
end
```

This isn't notably different to anything we've done so far. We define our function, `square/1`, which returns the result of calling the private member function, `do_square/1`.

Let's see what we get when we import this function into `iex`:

```elixir
iex(1)> import_file("mymath.exs")
{:module, MyMath, <<70, 79, 82, 49, 0, 0, 4, 224, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 113, 131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115, 95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>, {:do_square, 1}}
```

Everything looks correct. Wait, what about the function tuple? Why does it say `:do_square` and not `:square`? This is because our function is mostly uninteresting, and it is returning an in-lined version of our function. This does not mean that to use this module we need to know the private function name.

We still use the module and function the same way:

```elixir
iex(2)> MyMath.square(2)
4
```
Private functions are really no different from regular functions other than scoping. That is, in private functions we can use the same pattern matching techniques we have already used.

We will see some more examples of private functions soon when we go into some real code examples.

**Calling functions**

There's another issue I've so far been skirting over—the syntax of calling functions. Specifically, you may have seen some examples around in which calling a function does not use parentheses to denote the function call. For example, from our previous example, these are equivalent:

```
iex(3)> MyMath.square(4)
16
iex(4)> MyMath.square 4
16
```

Why, you ask, is this the case? Why do we have two different acceptable forms? Elixir has a lot of its roots in Erlang. But this is only, so far, as some basic syntax and the runtime. The syntax borrows fairly heavily from Ruby. Thus, there's some Ruby syntactical elements present throughout Elixir, and this happens to be one of the Ruby carry-overs.

As far as when to use one version of the syntax over another is concerned, it depends. I would argue it's mostly a preference of style. If you like to read functions with less braces running around, don't use parentheses to invoke functions. Or, you may like the explicit use as they denote calling a function. Whichever way you choose, you should tend toward consistency.

There are cases, however, where you will have to use parentheses. These cases are usually when the binding of the arguments needs to be explicit, you need to override the default operator precedence order, or you're calling an anonymous function.

For example, when a function has multiple arguments and/or this same function is invoked as an operand to another operator or function. Continuing with the `square/1` function from before, are these two expressions equivalent?

```
MyMath.square 2 * 5
```

```
MyMath.square(2) * 5
```
If you’re guessing they are not, you would be correct. Function invocation, typically, has less precedence than other operators. Parentheses can help make explicit what your intention is:

```elixir
iex(5)> MyMath.square(2 * 5)
100
iex(6)> MyMath.square 2 * 5
100
iex(7)> MyMath.square(2) * 5
20
```

My suggestion would be to use parentheses when the code may look ambiguous.

### When to use .

There’s another syntactical issue I’ve been neglecting to mention, the use of the `.()` syntax to invoke (some) functions. This, to me, is probably one of the oddest blemishes of Elixir’s syntax. It’s slightly confusing, and feels particularly awkward to use. However, it may never go away and we, as Elixir developers, will have to deal with it. The question is, then, when should we use it and when should we not? Fortunately, when to use it is fairly simple, but can often be forgotten when starting out—invoke anonymous functions or function handles grabbed with `&{foo}/arity`, otherwise, use the regular syntax (without the `.` syntax character).

For example, every time we define an anonymous function, we used the `.()` syntax to invoke it:

```elixir
iex(1)> f = fn x -> x * x end
...
iex(2)> f.(2)
4
```

However, when we define the function part of a module, we use the regular syntax:

```elixir
iex(1)> defmodule Foo do
...(1)> def f(x), do: x * x
...(1)> end
...
iex(2)> Foo.f(2)
4
```
Similarly, using our MyMap example from the previous chapter, notice how the function passed in is invoked with the . syntax, but calling the map function itself isn’t:

```iex(1)> defmodule MyMap do
    ...
    def map([], _), do: []
    def map([h|t], f) do
      [f.(h) | map(t, f)]
    end
    ...
end
```

```iex(2)> MyMap.map([1, 2, 3, 4, 5], &(&1 * &1))
[1, 4, 9, 16, 25]
```

**Grabbing functions**

Elixir supports passing defined functions as parameters. That is, Elixir's functions are first-class citizens of the type system. But then, how do we pass the existing functions around? We use the & operator or function capture operator. Going back to our MyMath.square/1 function, we could pass it to Enum.map/2 with the following:

```iex(1)> import_file("mymath.exs")
...
```

```iex(2)> Enum.map([1, 2, 3], &MyMath.square/1)
[1, 4, 9]
```

Here, we load the module again, for completeness, and then we invoke Enum.map/2 with the list [1, 2, 3] and pass our square/1 function from MyMath. You may wonder why we need to grab the function with the arity. This, if you recall, is because Elixir functions are defined by their name and arity or number of parameters. For example, say we define our square/1 function as pow/1 instead where, if pow is given one function, it assumes we want to raise the argument to the second power, otherwise, there is a pow/2 that takes the base and the power. It would look something like this:

```defmodule MyMath do
    def pow(x), do: pow(x, 2)

    def pow(x, p) do
        Enum.reduce(Enum.take(Stream.repeatedly(fn -> x end), p),
        &*/2)
        end
    end

```
Before we go on, let’s take a quick moment to discuss and break apart what this new `pow/2` function is doing.

The `pow/2` function takes the base, \( x \), and the power, \( p \), and we pass it to a particularly dense function chain. Reading inside out, we are using another internal Elixir module, `Stream`. Specifically, we call `Stream.repeatedly/1` with the argument, `fn -> x end`. This is, of course, an anonymous function that simply returns \( x \), our base. `Stream.repeatedly/1` will call this anonymous function as many times as it's required, possibly infinitely!

Moving a step out, we are using another function from the `Enum` module, `Enum.take/2`. This function takes a collection; in this case, the result of `Stream.repeatedly/1`, and a number, \( p \), and returns the first \( p \) elements of the collection.

Moving out again, we are using yet another function from `Enum`, `Enum.reduce/2`. The `reduce/2` function takes a collection, the result from `Enum.take/2`, and a function, `&*/2` (the multiplication operator), and reduces the collection into a single element.

Let's explore this more in an interactive session.

First, let's look at the result of `Stream.repeatedly/1`:

```iex(1)> repeatedly = Stream.repeatedly(fn -> 2 end)
#Function<24.29647706/2 in Stream.repeatedly/1>
```

Here, we see a function is returned. You might be thinking, "But you said it was a collection!". That is correct, I did say the result of `Stream.repeatedly/1` is a collection. However, this is a different type of collection than we've seen before, specifically, the `Stream` module produces lazy collections, which are essentially collections that are not realized until it's absolutely necessary.

Next, let's look at the result of `Enum.take/2`:

```iex(2)> taken = Enum.take(repeatedly, 5)
[2, 2, 2, 2, 2]
```

At this point, we have invoked the function returned into `repeatedly` five times and turned the result into a list, a list of the element 2 repeated five times.

Finally, we use the `Enum.reduce/2` function to collapse or fold the list into a single element:

```iex(3)> Enum.reduce(taken, &*/2)
32
```

Here, we are grabbing the multiplication operator as a function and using it to reduce our list of 2's into a single 32.
That was a long tangent on our new definition of pow/2; let's return to using our new version of the MyMath module.

Reloading the file into our interactive session, we have a choice to map our new function over some list using either pow/1 or pow/2:

```elixir
iex(1)> import_file("mymath.exs")
...
...
...
```

Oops! Since Enum.map/2 passes each element of our list to the function we pass (and nothing else), the function we pass must only accept a single argument. To use MyMath.pow/2, we must partially apply or curry (https://en.wikipedia.org/wiki/Currying) the function and pass that instead. Since Elixir doesn't have a native syntax for currying, we must resort to simply wrapping it in an anonymous function:

```elixir
iex(4)> Enum.map([1, 2, 3], &(MyMath.pow(&1, 3)))
[1, 8, 27]
```

When patterns aren't enough for matching

So far, we have seen pattern matching very basically in our functions and assignment (reading binding) expressions. However, there's more that can be done when doing pattern matches, particularly when defining functions. When a simple type decomposition pattern isn't enough, we can use guards to add an extra layer to our matches.

Guards are simply boolean expressions we can add to our function definitions to make the pattern matches that we define more strict or specific.

Here are some basic examples:

```elixir
iex(1)> defmodule MyMath do
...(1)> def sqrt(x) when x >= 0, do: #implement sqrt
...(1)> end
```
However, we are only allowed a limited set of expressions. The following is a pretty exhaustive list of the available expressions allowed in guard clauses:

- All comparison operators (==, !==, ===, !===, >, <, ==, >=)
- Boolean operators (and, or) and negation operators (not, !)
- <> and ++ as long as the left side is a literal
- The in operator
- All of the following type check functions:
  - is_atom/1
  - is_binary/1
  - is_bitstring/1
  - is_boolean/1
  - is_float/1
  - is_function/1 and 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)
  - bit_size(bitstring)
  - byte_size(bitstring)
  - div(integer, interger)
  - elem(tuple, n)
  - hd(list)
  - map_size(map)
  - node()
  - node(pid | ref | port)
  - rem(integer, integer)
° round(number)
° self()
° tl(list)
° trunc(number)
° tuple_size(tuple)

We won't do examples using all of the preceding listed guards, but this certainly gives you an idea of just how much you can accomplish with just guards.

In addition, users may also define their own guards, typically starting with is_.

**Functional algorithms for everyday problems**

Now that we have seen modules, functions, guards, the basic types of the previous chapter, and some basic examples of pattern matching, we essentially have everything we need to start solving problems. Let's take this further and actually write some code!

Let's solve some basic problems such as those you might find in your standard set of interview questions; however, instead of solving them with imperative code, we are going to see how we can solve them using only functional constructs and what we have covered so far.

**Iteration versus recursion**

Often in functional languages, we will use recursion instead of iteration since iteration, by its very nature, requires side-effects. That is, to use a for loop, most languages require that the loop modifies some state (usually, an integer) to keep track of where the loop is in execution.

Functional languages, contrastingly, opt for recursive strategies since these are (or, at the very least, can be) inherently non-destructive.

One may argue, though, that iteration doesn't have the same problems as recursion. Notably, iteration can potentially iterate infinitely, whereas recursion can be limited by the stack. Also, certain algorithms are less space and time complex when solved iteratively versus when solved recursively.
There are some benefits of recursion though too. Chiefly, describing computations with recursion can be simpler to reason about; when defining recursive functions, we are describing the more foundational, even mathematical, model of the concept. Iteration, in contrast, hides this conceptual insight from the programmer.

Additionally, functional recursion can typically overcome the mentioned limitations here. Coming from an imperative world, infinite recursion isn’t possible because the program would eventually run out of space on the stack. How would a language like Elixir or Erlang avoid this? The answer is in a concept you may have heard or called **tail recursion**. Tail recursion is, essentially, a way for the runtime to modify the call stack of a well-designed recursive function by collapsing the tailing frames:

In the preceding diagram, we are given the stack frame of recursively computing factorial of \( n \) where \( n = 5 \). We see that each frame is materialized and sticks around until the end when they start popping in rapid succession:
Chapter 3

Contrast the last diagram with the preceding one, where we materialize each frame but, instead of hanging onto each frame at each step, the runtime notices that the return value of each frame isn't used other than as a return value of its own. Therefore, we discard each frame and collapse the return values into a single frame or single expression.

To overcome the issue of time and space complexity, typically inherent with recursive algorithms, we usually must think only a little harder and define our recursive solution with an iterative pattern. That is, instead of doing what seems immediately obvious, recursively, we approach the problem with some thought, and what we usually come to is doing the recursion backward. This last issue may be better illustrated through an example.

Let's consider the Fibonacci sequence. The Fibonacci sequence is useful since it has an already recursive definition: \( f_n = f_{n-1} + f_{n-2} \) for \( n > 2 \) and \( f_0 = 0 \) and \( f_1 = 1 \).

This definition gives us the sequence we may expect: 0 1 2 3 5 8 13 21 ...

If we wanted to codify Fibonacci in Elixir, we may be tempted to define the following module:

```elixir
defmodule Fibonacci do
  def seq(0), do: 0
  def seq(1), do: 1
  def seq(n) when n > 1, do: seq(n-1) + seq(n-2)
end
```

That is, we define a module with a single function, `seq/1`. This function matches against integers; specifically, if the function is passed 0 or 1, it will return 0 or 1, respectively. If it's passed anything greater than 1, it will return, recursively, the addition of the two previous elements, exactly as the mathematical model defines it.

However, as you may have learned in your introductory algorithms class, this method is awfully slow; in fact, exponentially slow. Tail recursion can't even save us here. Although, it can help, it won't solve the massive duplication of work. How could we write this better still using recursion?

How would we do it sequentially?

Let's take a moment to think about this. If we had sequential constructs available, how could we use them to compute the \( n \)th number in the Fibonacci sequence?

Let's say, instead of using the definition as stated, let's flip it. Let's use it backward. Instead of starting at \( n \), let's start at 1 (or 2, really) and count up.
How would we do this using Elixir? Well, like I noted earlier, we are looking to do the recursion backward. Let's see what this might look similar to:

```elixir
defmodule Fibonacci do
def seq(0), do: 0
def seq(1), do: 1
def seq(n) when n > 1 do
  compute_seq(n, 1, [0, 1])
end

defp compute_seq(n, i, acc) when n == i, do: Enum.at(acc, length(acc) - 1)
defp compute_seq(n, i, acc) do
  len = length(acc)
compute_seq(n, i + 1, acc ++ [Enum.at(acc, len-1) + Enum.at(acc, len-2)])
end
end
```

Like earlier, we defined our Fibonacci module with the `seq/1` function. This function, returns 0 or 1 if n == 0 or n == 1, respectively. But then, instead of computing from the top of the sequence, we start from the bottom of the sequence and build to the result. This takes us from an exponential algorithm for computing Fibonacci to a linear one. Moreover, we haven't necessarily lost the expressiveness of the recursive function. Arguably, there's a little noise from the syntax, but the overall pattern of describing computation with expressions from the definition is still present.

Let's see how this second version compares in terms of performance. In `fibonacci_1.exs`, we will have the following script:

```elixir
defmodule Fibonacci do
def seq(0), do: 0
def seq(1), do: 1
def seq(n) when n > 1, do: seq(n-1) + seq(n-2)
end

IO.puts Fibonacci.seq(50)
```

And in `fibonacci_2.exs`, we will have our second, more iterative version of Fibonacci:

```elixir
defmodule Fibonacci do
def seq(0), do: 0
def seq(1), do: 1
def seq(n) when n > 1 do
  compute_seq(n, 1, [0, 1])
end
```
Save both of these files, and we will be able to use the time command to get an idea of the difference in running time between the two implementations. Here are the results from my machine:

$ time elixir fibonacci_1.exs
12586269025
elixir fibonacci_1.exs  753.42s user 0.07s system 100% cpu 12:32.98 total

$ time elixir fibonacci_2.exs
12586269025
elixir fibonacci_2.exs  0.29s user 0.08s system 115% cpu 0.314 total

That's fairly substantial! Notice that we do, in fact, get the same result, but doing it the first way is several orders of magnitude slower. This is a clear win for the second approach. Not only is it functional and recursive, we didn't sacrifice the performance to get it.

Performance considerations
There are a few performance considerations that must be handled when using tail recursion. I mentioned well-designed recursive functions earlier. What does that mean? What are the implications?

Here, well-designed means the runtime is actually capable of performing the tail call optimization. If we attempt to use the return of a recursive call in the wrong way, we completely lose the tail call optimization. For example, if we had defined our factorial function from the second diagram as \( \text{fact}(n-1) \times n \) instead of \( n \times \text{fact}(n-1) \), we would have lost the tail-recursion optimization completely.
To ensure that we can actually use the optimization, we must take care to define the tail of the function properly. That is, the return can be rolled up into the previous frame without breaking the closure that the frame would have created. Returning to the factorial example, using \( \text{fact}(n-1) \times n \) breaks the closure created by the current stack frame because we must know what \( n \) is when we return. Using \( n \times \text{fact}(n-1) \) doesn't break the closure because the result can be substituted into current expression.

Tail recursion is a really cool feature and enables some nice optimizations for the runtime. However, like all things, don't abuse it. There will be times when attempting tail recursion will be the wrong way to do something. Take some pragmatism when considering certain algorithms and approaches.

Reverse

Next, let's see how we can easily define a function to reverse a list. In fact, we've already seen the pattern we are going to use to do it too. In the previous chapter, we defined a map function that used \([h|t]\). This pattern is a way to deconstruct a list (or tuple) into its head component, \( h \), and its tail component, \( t \). We will use this again to construct a function that will reverse a list:

```elixir
defmodule Reverse do
  def reverse([]), do: []
  def reverse([h|t]), do: reverse(t) ++ [h]
end
```

This is fairly straightforward code. We define a pattern for our base case, the empty list, and simply return the empty list. We also define a pattern that allows us to decompose the head and tail of the list. The body of this pattern is then used to build the reverse by appending the head of the list to the recursive call of reverse on the tail.

Let's load this example and see it work:

```elixir
iex(1)> import_file "reverse.exs"
...
iex(2)> Reverse.reverse [1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
iex(3)> Reverse.reverse 'Hello, World!'
'!dlroW ,olleH'
```

Recall from the previous chapter how single-quoted strings are really simply a list of characters. That is, \( \text{is_list}('hello') \) returns true. Thus, we can use our reverse function on character lists as well.
Sorting

Another example of pattern matching is implementing sorting algorithms. Quicksort, it turns out, is a bit of a favorite for this since we have nice ways to deconstruct lists:

```elixir
defmodule Sort do
  def quicksort([], do: [])
  def quicksort([h|t]) do
    lower = Enum.filter(t, &(&1 <= h))
    upper = Enum.filter(t, &(&1 > h))
    quicksort(lower) ++ [h] ++ quicksort(upper)
  end
end
```

We define a module, Sort, with a single function quicksort/1. The empty list case simply returns the empty list as we expect. In the nonempty case, we again use the | operator to split the list into head and tail elements. We construct the lower and upper lists using the Enum.filter/2 function. This function takes a collection and an anonymous function and returns the elements that returned true from the anonymous function. In our case, we are using the function, &(&1 <= h), to find elements smaller or equal to h and similarly, &(&1 > h), to find elements strictly bigger than h. After constructing the two lists, we recursively call quicksort on the lower and upper lists, concatenating the results to get our result, a sorted list.

Let's see it in action:

```elixir
  iex(1)> import_file "sort.exs"

  iex(2)> Sort.quicksort Enum.shuffle 1..10
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  iex(3)> Sort.quicksort Enum.reverse 1..10
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Mix – the ladle of Elixir

So far, we have only interacted with simple modules defined in scripts or the interactive prompt. But this will only take us so far. Eventually, we will need more than just scripts. We will need a source tree that encloses our project's code. Moreover, we will need a tool to create the source tree, build the source, test, manage dependencies, and a number of other tasks. That tool is mix.
This tool handles everything we could need from a build tool. It creates projects, compiles code, runs tests, packages projects into distributable units, and even allows us to run our project, importing the necessary files into iex.

But enough about mix, let's use it and see it in action!

First, like any good command, mix comes with the trusty help command to give us a good list of what it can do for us:

```bash
$ mix help
mix                   # Run the default task (current: mix run)
mix app.start         # Start all registered apps
mix archive           # List all archives
mix archive.build     # Archive this project into a .ez file
mix archive.install   # Install an archive locally
mix archive.uninstall # Uninstall archives
mix clean             # Delete generated application files
mix cmd               # Executes the given command
mix compile           # Compile source files
mix deps              # List dependencies and their status
mix deps.clean        # Remove the given dependencies' files
mix deps.compile      # Compile dependencies
mix deps.get          # Get all out of date dependencies
mix deps.unlock       # Unlock the given dependencies
mix deps.update       # Update the given dependencies
mix do                # Executes the tasks separated by comma
mix escript.build     # Builds an escript for the project
mix help              # Print help information for tasks
mix hex               # Print hex help information
mix hex.config        # Read or update hex config
mix hex.docs          # Publish docs for package
mix hex.info          # Print hex information
mix hex.key           # Hex API key tasks
mix hex.outdated      # Shows outdated hex deps for the current project
mix hex.owner         # Hex package ownership tasks
mix hex.publish       # Publish a new package version
mix hex.search        # Search for package names
```
mix hex.user          # Hex user tasks
mix loadconfig        # Loads and persists the given configuration
mix local             # List local tasks
mix local.hex         # Install hex locally
mix local.rebar       # Install rebar locally
mix new               # Create a new Elixir project
mix run               # Run the given file or expression
mix test              # Run a project's tests
iex -S mix            # Start IEx and run the default task

Clearly, \texttt{mix} provides us with a lot of functionality. Furthermore, we can use \texttt{mix help} to get more about a specific task. Since we may want to know how to create projects, let's look at the \texttt{new} task:

\$ \texttt{mix help new}

```
mix new
```

Creates a new Elixir project. It expects the path of the project as argument.

```
mix new PATH [-sup] [--module MODULE] [--app APP] [--umbrella]
```

A project at the given PATH will be created. The application name and module name will be retrieved from the path, unless --module or --app is given.

A --sup option can be given to generate an OTP application skeleton including a supervision tree. Normally an app is generated without a supervisor and without the app callback.

An --umbrella option can be given to generate an umbrella project.

An --app option can be given in order to name the OTP application for the project.

A --module option can be given in order to name the modules in the generated code skeleton.
Examples

```
mix new hello_world
```

Is equivalent to:

```
mix new hello_world --module HelloWorld
```

To generate an app with supervisor and application callback:

```
mix new hello_world --sup
```

Location: `/usr/lib/elixir/lib/mix/ebin`

We see that we get the basic command example and some descriptions about the arguments we can specify:

- The `--module MODULE` argument allows us to create our first module with a different name than PATH. That is, the default will create a folder and module of the same name (with some modifications). However, if we want to change this behaviour, we can use this argument to do so.
- Similar to the `--module` argument, `--app APP` allows us to rename the Elixir application created by the project.

The `--app` argument and the rest of the arguments will make more sense in upcoming chapters and for now, we will mostly ignore them. For now, let's create our first project!

**Structure of Elixir projects**

Let's start by creating a small project called `hello_world`:

```
$ mix new hello_world
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/hello_world.ex
```
* creating test
* creating test/test_helper.exs
* creating test/hello_world_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

```
cd hello_world
mix test
```

Run `mix help` for more commands.

The output of `mix new` tells us of the files it has created, that it created it successfully, and where to go from here.

Let's navigate to our new project's directory and see what's there in it:

```
$ cd hello_world
$ find .
./mix.exs
./.gitignore
./config
./config/config.exs
./README.md
./lib
./lib/hello_world.ex
./test
./test/test_helper.exs
./test/hello_world_test.exs
```

This is essentially just another view into the list of files that `mix new` gives us, but it certainly confirms what it says. Let's break each of these files and folders down.

**mix.exs**

This is the file that describes our project. Unsurprisingly, it's actually Elixir code and defines a module. This is used by `mix` for compiling, testing dependencies, packaging our project, and running our project.
If we open it up, we will see something like the following code:

```elixir
$ cat mix.exs
defmodule HelloWorld.Mixfile do
  use Mix.Project

  def project do
    [app: :hello_world,
     version: "0.0.1",
     elixir: "-> 1.0",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps]
  end

  # Configuration for the OTP application
  #
  # Type `mix help compile.app` for more information
  def application do
    [applications: [:logger]]
  end

  # Dependencies can be Hex packages:
  #
  #   {:mydep, "-> 0.3.0"}
  #
  # Or git/path repositories:
  #
  #   {:mydep, git: "https://github.com/elixir-lang/mydep.git", tag: "0.1.0"}
  #
  # Type `mix help deps` for more examples and options
  defp deps do
    []
  end
end
```

It's not worth going into too much detail here, but we can see that it's really just an Elixir module and it defines a few functions that return information about the project.

**.gitignore**

There are several files that we will want to ignore when tracking our code with a VCS tool such as git. The mix tool creates this ignore file for us to give us a sane set of default exclusions for starting out, and we do not need to remember this set every time we create a project.
config

The config folder contains our project's configuration settings. These are our global, high-level options that our project can reference during execution.

By default, a single file is created for us here—config/config.exs. As the file extension might hint, this too is Elixir code. It defines a set of variables we can reference in our project.

README.md

Another default file mix new creates for us is a basic README file using the markdown markup syntax. This is a good file to have for all the projects as it's a good center place to shortly (or longly) describe your project, how to build and install it, how to use it, and where to go for more information.

Of course, your documentation is only as good as your effort to maintain it. Creating the file with a TODO tag isn't going to be enough; before you publish your project, be sure to make a pass through this file and update it.

lib

The lib directory, unlike other languages, is actually for your Elixir source. This will be where mix will look when building the project.

For this simple project, the lib directory only contains one file—lib/hello_world.ex. This file defines the so far uninteresting module named HelloWorld:

```
$ cat lib/hello_world.ex

defmodule HelloWorld do
end
```

This is what we get to start. Not terribly interesting, but it's something. It doesn't impose a lot, or assume too much. It's a plain module we get to shape as our project matures.

test

The test directory, as you may have suspected, is for test code. The mix tool will compile and load the modules defined in this folder, and run over the test cases defined therein.

There's initially two files created—test/test_helper.exs and test/hello_world_test.exs. The former sets up the test harness for the project:

```
$ cat test/test_helper.exs

ExUnit.start()
```
The latter file defines a simple, but passing, module and a test to give us a starting point for writing our tests:

```
$ cat test/hello_world_test.exs
defmodule HelloWorldTest do
  use ExUnit.Case

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

## Compiling a project

Now that we have done a high-level walk-through of the project we have just created, let's see how `mix` can compile this project and turn what we have so far into bytecode for the Erlang VM.

From the root of the project directory, go ahead and run `mix compile`:

```
$ mix compile
Compiled lib/hello_world.ex
Generated hello_world app
```

Elixir found our `lib/hello_world.ex` module, compiled it, and then generated our app. We won't concern ourselves with the concepts of Erlang applications for now, but we just acknowledge its existence.

If we list the contents of our directory now, we should see a `_build` directory:

```
$ ls
_build  config  lib  mix.exs  README.md  test
```

Under this `_build` directory will be the `.beam` files of our project and the app file.

## Testing a project

Simply to see what testing in Elixir looks similar to, we can go ahead and test our project as well:

```
$ mix test
Compiled lib/hello_world.ex
Generated hello_world app
```

```
This tells us a few things: it compiles our source, if it isn't already; runs the tests; times the loading time and computation time of our tests; gives us the result of our tests, pass or fail; and the random seed the test run used.

**Running interactively**

Finally, if we want to run our code, we can use a special option of `iex` to compile our project, if necessary, and include it into an interactive session:

```
$ iex -S mix
```

Erlang/OTP 17 [erts-6.4] [source] [64-bit] [smp:12:12] [async-threads:10] [hipe] [kernel-poll:false]

Compiled lib/hello_world.ex
Generated hello_world app

Interactive Elixir (1.0.4) - press Ctrl+C to exit (type h() ENTER for help)

```
iex(1)> 
```

This isn't entirely interesting yet. Certainly, we are in an interactive session, but we haven't defined anything in our `lib/hello_world.ex` module; thus, there isn't really anything exciting here for us. Let's correct that by way of revisiting "Hello, world!" quickly.

Open up the `lib/hello_world.ex` file in your favorite editor and add the following function:

```elixir
def hello(name \ "World") do
  IO.puts "Hello, #{name}!"
end
```

Now, let's return to the interactive session:

```
$ iex -S mix
```

Erlang/OTP 17 [erts-6.4] [source] [64-bit] [smp:12:12] [async-threads:10] [hipe] [kernel-poll:false]

Compiled lib/hello_world.ex
Generated hello_world app
Interactive Elixir (1.0.4) - press Ctrl+C to exit (type h() ENTER for help)

Now, we can invoke our function from the interactive prompt:

```
iex(1)> HelloWorld.hello
Hello, World!
:ok
iex(2)> HelloWorld.hello "Kenny"
Hello, Kenny!
:ok
```

**Files**

Let's take a quick moment to revisit Elixir files.

As we have seen again and again, Elixir has two different file types, `.ex` and `.exs`, but what are their differences?

The distinction is actually more conventional and signifies intention. Both files are compiled to bytecode and run by the VM. But the file extension informs the compiler and VM, and ourselves, that a file has a particular purpose.

That is, `.ex` files denote source or project files. The intention of the file is to run part of a project or application. Compared to `.exs` files, whose denoted intention is either scripting, configuration, or testing. These files are compiled the same as the `.ex` counterpart, but the resulting bytecode of these files is ephemeral and discarded after the file’s purpose is served.

**Mix and beyond**

Though we have only touched the surface of `mix` and its capabilities, we will be revisiting `mix` throughout, incrementally building a bigger image around `mix`. Till then, let's play with some more projects and revisit some examples from before.

**Building functional projects**

We have a lot in our proverbial tool belt so far and we are adding more. Let's take a look at some more examples where we solve some relatively simple problems using Elixir. This time, however, let's use `mix` as well.
Flatten

We will start small, by creating a function to flatten arbitrarily deep, nested lists.

Let’s create a project for it using mix new flatten:

```
$ mix new reverse
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/flatten.ex
* creating test
* creating test/test_helper.exs
* creating test/flatten_test.exs
```

Your mix project was created successfully.

You can use mix to compile it, test it, and more:

```
  cd flatten
  mix test
```

Run mix help for more commands.

Now, in the lib/flatten.ex file, let’s create the flatten function.

Open the file in your favorite editor and add the following flatten/1 function:

```
defmodule Flatten do
  def flatten([],), do: []
  def flatten([h|t]) when is_list(h), do: h ++ flatten(t)
end
```

In our Flatten module, we define our flatten/1 function with two patterns, the empty list base case, and the [h|t] pattern adding that h is a list. When we match against the second pattern, we return the list created by h and append the recursive flattening of t.
Of course, we can go ahead and launch it into iex and make sure it works:

```bash
$ iex -S mix
...

iex(1)> Flatten.flatten [[1, 2], [3], [4, 5]]
[1, 2, 3, 4, 5]
```

Awesome, it works. How could we more quickly try a bunch of different cases and ensure that our function to flatten arbitrarily nested elements does in fact work as we expect? We could write a test.

### A small introduction to testing

Testing is an inescapable aspect of software development. It is so essential, most languages now are building it in as part of the language.

Let's take a brief moment to introduce Elixir's testing functionality.

I've already shown you `mix test` and what it does. But let's write a test for our `flatten/1` function.

Writing tests in Elixir, besides syntax, is no different from writing functions. We define a module and a few tests (functions), and assert some results that we expect.

In our case, a third of this is already done! Since we created the project, `mix` helpfully created the `test` folder and stubbed out a module for us to use for testing. Let's open and examine this file.

In your favorite editor, open `test/flatten_test.exs`. You should see something like the following code:

```elixir
defmodule FlattenTest do
  use ExUnit.Case

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

It is a simple enough module. It defines the module and defines a test, `the truth`, that asserts that `1 + 1` does in fact equal 2.

The string, `the truth`, is the identifier of the test. We can use this to give a good description of the test and its assumptions and/or assertions.
Let's modify this quickly to demonstrate a failing test. Change the assertion statement to:

```elixir
assert 2 + 2 == 5
```

Then, let's run the test and see what we get:

```
$ mix test

1) test the truth (FlattenTest)
   test/flatten_test.exs:4
   Assertion with == failed
   code: 2 + 2 == 5
   lhs:  4
   rhs:  5
   stacktrace:
       test/flatten_test.exs:5
.

Finished in 0.03 seconds (0.03s on load, 0.00s on tests)
1 tests, 1 failures
```

Since the case fails, Elixir gives us a lot of information about the state and the values that caused the assertion to fail. We can use this information to begin looking at what is causing the failing test.

Moving back to our `flatten/1` function, let's remove this test case and define a new one.

We can go ahead and remove the simple test case. We will start with a simple case of our own:

```elixir
test "return flat list when given nested lists" do
  expected = [1, 2, 3, 4, 5]
  actual = Flatten.flatten [[1, 2], [3], [4, 5]]
  assert actual == expected
end
```
Save the file, and we can go ahead and run the test:

```bash
$ mix test
Compiled lib/flatten.ex
Generated flatten app
.

Finished in 0.02 seconds (0.02s on load, 0.00s on tests)
1 tests, 0 failures

Randomized with seed 138347

As we expect, it passes. However, something smells about this implementation. Let's write a test to sniff out this smell.

Let's add the following new test case:

```elixir
test "return flat list when no nesting" do
  expected = [1, 2, 3, 4, 5]
  actual = Flatten.flatten expected
  assert actual == expected
end
```

Save the test file again, and let's run this:

```bash
$ mix test
.

1) test return flat list when no nesting (FlattenTest)
test/flatten_test.exs:10
** (FunctionClauseError) no function clause matching in Flatten.flatten/1
stacktrace:
  (flatten) lib/flatten.ex:2: Flatten.flatten([1, 2, 3, 4, 5])
test/flatten_test.exs:12

Finished in 0.03 seconds (0.03s on load, 0.00s on tests)
2 tests, 1 failures

Randomized with seed 211452
```
This is interesting. We get a no match exception. Looking back to our `flatten/1` function, what could be causing this?

It turns out, we *always* assume that the head of the list will be another list. But we said the function should handle arbitrarily-deep nesting. Zero nesting is arbitrarily deep. We need another pattern.

Let's add the new pattern to our `flatten/1` function and try testing again.

Modify the `Flatten` module to look similar to the following:

```elixir
defmodule Flatten do
  def flatten([],), do: []
  def flatten([h|t]) when is_list(h), do: h ++ flatten(t)
  def flatten([h|t]), do: [h] ++ flatten(t)
end
```

Then, running the tests again, we should get a passing test:

```
$ mix test
Compiled lib/flatten.ex
Generated flatten app
..

Finished in 0.03 seconds (0.03s on load, 0.00s on tests)
2 tests, 0 failures

Randomized with seed 466648
```

We do get that.

The process we just went through isn't too different from TDD or other testing and test development strategies.

**More to do about modules**

Modules are the enclosing unit of our building blocks. They are the super block to which functions are the blocks. But occasionally, we wish to get data about our modules and metadata on our building blocks. We would probably like to document our code. We would like those following us to be able to read those comments and documentation. Better yet, we would like to have nice tooling around building rich documentation about our code while minimizing duplication, if not entirely eliminating it. Or, instead of documentation, we would like to tag our modules and functions with certain attributes that we could later use for any other number of reasons.
To support these goals, Elixir gives us the ability to give modules attributes, which we can use as developers, or users, or they can be used by the VM. Similarly, we can use attributes as constants.

Attributes are defined in Elixir as @name. For example, I could add the @vsn attribute to annotate a module named MyModule:

```elixir
defmodule MyModule do
  @vsn 1
end
```

That is, I’ve defined a simple module with a single annotation of vsn. Next, let's look at the two most used attributes—@moduledoc and @doc.

We could define a Math module, using the @moduledoc and @doc attributes appropriately:

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

  @doc ""
  Calculate factorial of a number.

  ## Example
  iex> Math.factorial(5)
  120
  """
  def factorial(n), do: do_factorial(n)
  defp do_factorial(0), do: 1
  defp do_factorial(n), do: n * do_factorial(n-1)

  @doc ""
  Compute the binomial coefficient of `n` and `k`

  ## Example
  iex> Math.binomial(4, 2)
  6
  """
  def binomial(n, k), do: div(factorial(n), factorial(k) * factorial(n-k))
end
```

```
The text between the @moduledoc and @doc attributes can be Markdown, and the documentation printed to the screen will use this for formatting and printing properly.

Save this module to a file named math.ex and let's compile it:

```bash
echo $?
```

If all is well, we should see no output and no exit code.

Next, we can launch iex and get the documentation about our new Math module:

```iex
iex(1)> h Math
```

```
Math

Provides math-related functions

```

```iex
iex(2)> h Math.factorial
```

```
def factorial(n)

Calculate factorial of a number, n.

Example

```
```

```iex
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
We can add some @doc comments to the function, and add what would look similar to an iex session of us manually testing the function, but in fact, would be tests run by the mix test.

Open the flatten.ex file from earlier and add the @doc attribute:

```elixir
defmodule Flatten do
  @doc ""
  Flatten an arbitrarily nested lists

  ## Examples
  iex> Flatten.flatten [[1, 2], [3], [4, 5]]
  [1, 2, 3, 4, 5]
  iex> Flatten.flatten [1, 2, 3, 4, 5]
  [1, 2, 3, 4, 5]
  ""
  def flatten([], ), do: []
  def flatten([h|t]) when is_list(h), do: h ++ flatten(t)
  def flatten([h|t]) do: [h] ++ flatten(t)
end
```

Next, open the flatten_test.exs file as well and add the following line to the top:

```elixir
doctest(Flatten)
```

Then, when running the mix test, we should see that more (of the same) tests are run:

```
$ mix test
Compiled lib/flatten.ex
Generated flatten app
...

Finished in 0.05 seconds (0.05s on load, 0.00s on tests)
3 tests, 0 failures
```

Tests in the @doc attributes will be combined into a single test when tested this way. So, although we defined two different inputs and outputs, we are only adding a single new test.
Exercises
Do:

- Write tests for `Reverse.reverse` and `Sort.quicksort`.
- Write a function to determine whether the given string is palindrome.
  - Write some tests for this function, considering the length of the string, is it even or odd, capitalization, punctuation, and space.
  - See whether you get an implementation that is just $O(n)$.

Summary
This chapter was also full of material, so let's take a quick moment to recap some of the material we covered.

We discussed how modules are defined, their purpose, and a little about their representation.

We went over functions, more about them as types, more explicitly into how they are defined, and the difference between anonymous (unnamed) functions versus named functions.

Along the lines of functions, we discussed pattern matching more and how it can be used to solve problems.

Using patterns, we also discussed guard statements and what functions or checks are available to us when using guards.

We also introduced Elixir’s build tool, `mix`, and how we can use it to create projects, and we started our first few projects.

Along with projects and `mix`, we introduced a few cool features with respect to testing available to us via `mix` and Elixir.

So far, we introduced the functional building blocks required to create programs and applications in Elixir. Next, we will want to see how we can improve some of our building blocks with the new processing techniques.
Where to buy this book
You can buy Learning Elixir from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.