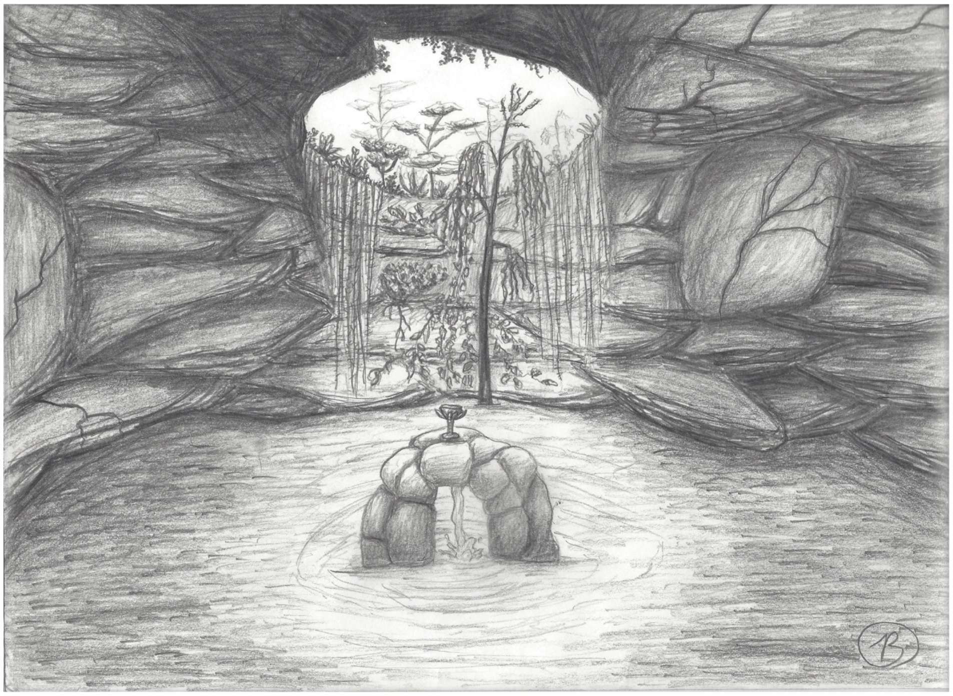
The Pragmatic Programmers

grox.io Learning

Programmer Passport

Elixir



Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: Elixir

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: Corina Lebegioara
Founders: Andy Hunt and Dave Thomas

ISBN-13: 978-1-68050-962-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2022

Contents

Preface

Since its release in 2011, Elixir has grown to be one of the leading functional programming languages in the world. Many industry trends have contributed to this, especially in areas where this budding language is strong. Insatiable demand for computers with more cores is pushing programming languages toward better concurrency support, and Elixir has a particularly good story here. Explosive growth in the Internet of Things has created a demand for Elixir's many frameworks for managing, networking, and measuring hardware. A push for more interactive web systems is driving demand for web programming tools like Elixir's Phoenix. These developments led Groxio to publish a series of Elixir videos, projects, and this book.

You might wonder whether the world needs yet another Elixir book. It's a good question. *Programming Elixir 1.6 [Tho18]* by Dave Thomas provides a great introduction to Elixir for intermediate programmers. *Learn Functional Programming with Elixir [Alm18]* by Ulisses Almeida offers a good Elixir overview for those learning functional programming. This book is neither as comprehensive as Dave's book, nor as focused as Ulisses's. We think those books are better places to learn Elixir.

Still, we think there's a place for this book.

If you think of a book as a travel guide, this book provides quick day trips that many travelers miss. We'll focus on several blind spots that beginning and intermediate Elixir developers encounter. We'll walk you through how to explore types in IEx and when to use Elixir's primitive data types. We'll unlock sigils and show you how macros work. Together, we'll build a `mix` task.

If this doesn't sound like what you're looking for, that's OK. Pick up another book. If it sounds interesting to you, read on. If you find it particularly useful, you might like it well enough to take the plunge and become a full Groxio subscriber.[1] Regardless, enjoy this quick guide through this fascinating language!

Bruce Tate
May 2022

1. https://grox.io

CHAPTER 1

Sweet Tooling

Elixir is one of the most important languages created in the past decade. It's a functional language, meaning the underlying concepts deal with mathematical functions. It's an immutable language, meaning Elixir programs won't *mutate* or *change* values, opting instead to have functions that *transform* values. Its smooth, friendly, Ruby-inspired syntax makes it understandable by a generation of object-oriented programmers. Its Erlang foundations make Elixir massively scalable with excellent features for reliability.

More than that, Elixir has libraries and tooling for solving some of the most important problems of our day. Several growing communities exist under the overall Elixir umbrella, and each community has impressive libraries and infrastructures.

Phoenix is a community that's growing rapidly. It has a web server that's stunningly reliable and concurrent. Though Elixir isn't fast when measured on a single core, it's incredibly concurrent, leading Phoenix to accumulate staggering statistics at scale. For some use cases a single Phoenix box can serve hundreds of thousands of concurrent users. The OTP foundation leads to great uptime numbers for Elixir applications. Phoenix offers these advantages along with a development model that's rich enough for experts but simple enough for intermediate developers. A new library called Phoenix LiveView leverages these strengths to build highly interactive web applications without involving custom JavaScript, leading to excellent productivity.

The Nerves project is another Elixir community that's growing hand over fist. Most developers of embedded devices use C, and a few are starting to use Python. Nerves offers better tooling along with the reliability features of Elixir. As embedded chip designers begin to embrace multiple cores, the concurrency advantages of Elixir will begin to tell. Like Phoenix, Nerves is experiencing explosive growth.

The Elixir tooling stack is rich for such a young language. It has a package manager called Hex, one that's open to both Elixir and Erlang projects. Hex provides a place to share a common infrastructure and also a way to resolve dependencies. Another tool, called mix, provides a way to wrap up development tasks such as running tests, compiling projects, and the like. Elixir also provides a command-line shell, debugging services that work both locally and remotely, and an inspector that makes it easy to see all of an application's processes.

Elixir is a language based on Joe Armstrong's creation, Erlang. Before we dive into Elixir, we're going to devote a few paragraphs to exploring why Erlang remains so important, even thirty years after its creation.

Based on Erlang

A team at Erickson, including Joe Armstrong, was building applications to work with phone switches. Erlang, the movie,[1] tells this story. These telecom programs had to be extremely reliable, with real-time performance, and excellent concurrency. Because of Joe's experience with Prolog, they wanted a language that would work in a declarative style. They quickly ruled out languages like C and Smalltalk because they weren't declarative or expressive enough for the problems Erickson was solving. They also ruled out existing functional languages like Prolog and ML because those languages didn't have the support for concurrency or low-level constructs for dealing with the bits that often showed up in hardware problems. Reluctantly, they decided to build a new language from scratch.

Erlang Escapes the Lab

Over time, the team built Erlang, and the new language quickly accumulated an impressive list of successes in Erickson. They established a way of building generic services called GenServers, and wrapped that up into a library called OTP (the Open Telephony Protocol). This library established a strategy for dealing with concurrency and failure in a uniform way. Eventually, Erlang was released beyond Erickson and established a growing following as a language for building reliable infrastructure.

OTP and the BEAM: Erlang's Crown Jewels

The centerpieces of the Erlang language are the virtual machine it runs on, called the BEAM, and the OTP framework for running scalable, reliable ser-

1. https://www.youtube.com/watch?v=xrIjfIjssLE

vices. As it grows, the BEAM is becoming known for high concurrency, responsive performance, high reliability, and excellent support for processes. OTP runs on the BEAM, and it describes an application's processes and life-cycles so that individual pieces can be easily shut down and restarted upon failure.

These changes make Erlang one of the most reliable programming languages in existence. When any part of a distributed Erlang system crashes, the infrastructure can simply shut it down and restart it.

These advantages have gathered a fierce following among a small set of infrastructure and application developers. So far, Erlang has yet to break through into the mainstream. Most people believe that when a BEAM-based language does break through, it'll be Elixir. Let's find out why José Valim developed Elixir.

Beyond Erlang

José was a leader in the community supporting Ruby on Rails, one of the most successful web development frameworks ever built. José was an author, core team member, and framework developer in that Ruby space with a large following, but he was growing increasingly frustrated with some of the problems he encountered related to scalability and programming abstractions. Ruby was an excellent language, but not appropriate for the types of problems José was solving most frequently:

- The Rails framework wasn't explicit, so it was sometimes difficult to debug and extend.
- The Ruby language didn't support concurrency well, an increasingly important language feature.
- Ruby applications didn't scale particularly well for many users.

In 2011, Elixir emerged and began to solve some of these problems. With its Erlang foundations, Elixir began to grow.

Let's clarify one misconception right now. Elixir isn't simply Erlang with a different syntax. It's a modern language with important new features and a wide suite of tools that Erlang has traditionally lacked. In this section, we'll highlight some of those differences.

Mass Appeal

To grow rapidly, a language must be easy to learn, so it needs convenient and popular syntax, strong documentation, and approachable tooling. Elixir has all of these things in spades. Based on Ruby, Elixir's syntax is far more approachable to a large audience than Erlang. It's not that Elixir's syntax is inherently better. Rather, Elixir's syntax is based on Ruby, a far more popular language than the strongest influence on Erlang's syntax, Prolog.

Having effective and consistent documentation is also critical. From the very beginning, Elixir's core team built on the successful documentation strategies from Ruby. Elixir also has means for publishing documentation on Hex. The result isn't only a tool set, but also a culture around good documentation.

Elixir also provides tools that are critical to those who would build early libraries: a good build tool called `mix` and a package manager to store and share dependencies called Hex. These tools ensured that when others were ready to contribute to Elixir, they could immediately build and share tools.

New Abstractions

In the thirty years since Erlang's creation, the state of the art for what makes an effective language has advanced. Elixir built in several critical advancements that Erlang supports only poorly, or not at all. Protocols provide a way to safely extend types to support new functions. Streams are an abstraction for long or infinite data sets. Structs allow the rapid creation of structured data types. Macros allow the rapid creation of advanced functions in Elixir itself. Elixir pipes make Elixir easier to learn by letting beginners write programs as a series of transformations. We'll look at each of these features throughout the rest of this book.

Taken separately, these features are interesting. Taken together, they elevate Elixir programming as a whole. More higher-level abstractions in the hands of a good programmer improve productivity, allow better reliability, and make language learning easier.

These features help Elixir extend the reach of the BEAM to more developers, and potentially make each developer more effective. Effectively, Elixir is drawing a whole new user base into the Erlang ecosystem.

Elixir is also having an unexpected impact on the BEAM. Elixir's existence is making a better Erlang. The Hex package manager is becoming increasingly valuable to Erlang developers, and many Erlang libraries are consumed directly by Elixir programmers.

Now that you have an appreciation of where Elixir came from, let's start to put some of those ideas into use. In this chapter, we're going to focus on Elixir's tooling, the primary data types, and the functions that operate on those data types.

We'll start by installing Elixir and then working with Elixir's tooling, including mix.

Tools

It's time to take Elixir for a spin. If you haven't already done so, install Elixir.[2] Those instructions will also install Erlang. We're going to use version 1.10, though earlier versions should work fine, and later versions absolutely will work. When you've done so, verify your installation by requesting a version number, like this:

```
[elixir] ➔ elixir -v
Erlang/OTP 21 [erts-10.2] [source] [64-bit] [smp:12:12]
[ds:12:12:10] [async-threads:1] [hipe]

Elixir 1.10.1 (compiled with Erlang/OTP 21)
[elixir] ➔
```

We're up! We're using Elixir version 1.10, on Erlang/OTP release 21. Let's build our first Elixir app.

Mix Manages Development Tasks

One of the tools you installed is mix. This build tool is like rake for Ruby, ant for Java, or make for C. It's the tool we'll use to compile programs, run tests, fetch dependencies, and create projects. When you use Elixir, knowing mix is a must. Let's create our first project:

```
[elixir] ➔ mix new hello
* creating README.md
* creating mix.exs
* creating lib
* creating lib/hello.ex
* creating test
* creating test/hello_test.exs
...

Run "mix help" for more commands.

[elixir] ➔ cd hello
[hello] ➔
```

We created a project, and Mix generated several files. I've shortened that list here a little bit, but you get the idea. Elixir will create the same project structure each time, with a few additions. That means you'll have a pretty good idea where files go in any Elixir project that's created with mix.

2. https://elixir-lang.org/install.html

Mix Runs Tasks

While we're at it, let's look at all of the things you can do with mix:

```
mix                     # Runs the default task (current: "mix run")
...
mix clean               # Deletes generated application files
mix cmd                 # Executes the given command
mix compile             # Compiles source files
mix deps                # Lists dependencies and their status
mix deps.clean          # Deletes the given dependencies' files
mix deps.compile        # Compiles dependencies
mix deps.get            # Gets all out-of-date dependencies
...
mix do                  # Executes the tasks separated by comma
mix format              # Formats the given files/patterns
mix help                # Prints help information for tasks
mix new                 # Creates a new Elixir project
mix release             # Assembles a self-contained release
mix release.init        # Generates sample files for releases
mix run                 # Starts and runs the current application
mix test                # Runs a project's tests

mix xref                # Prints cross-reference information
iex -S mix              # Starts IEx and runs the default task
```

I've shortened this list a lot, but you can see it's a pretty healthy tool with plenty of options. Mostly, we'll use it to *compile* and *deploy* our program, work with *dependencies*, *run tests*, and *run other custom commands*.

Mix Projects Have a Strict Structure

We'll get to more mix commands in a minute. For now, let's go back to the Elixir project we created. You can see that Elixir created a main /hello directory with the two subdirectories test and lib. We'll put application files in lib and tests in test.

```
hello
  ├── lib
  └── test
```

These directories will hold our configuration, application source files, and test files, respectively. Elixir programs have two types of extensions. Interpreted scripts have the extension exs and compiled source files have the extension ex.

Mix generated three *program* files. The first is mix.exs which describes our project. Next is hello.ex, an example program. Finally, hello_test.exs is a test for our code. Here is where each of the files lives:

```
hello
  mix.exs
  ├── lib
  │     hello.ex
  └── test
        hello_test.exs
```

Let's look at those files one by one, and along the way, put the Mix features through their paces. We'll start with the program, hello.ex:

```
defmodule Hello do
  @moduledoc """
  Documentation for `Hello`.
  """

  @doc """
  Hello world.

  ## Examples

      iex> Hello.hello()
      :world

  """
  def hello do
    :world
  end
end
```

That's a typical program. We declare a module to hold our code, present some documentation, and declare a function. You can see the emphasis the Elixir community puts on strong documentation. The `@doc` lines signify that each Elixir project should have documentation, and you can base your documentation on the examples in your first generated Mix application.

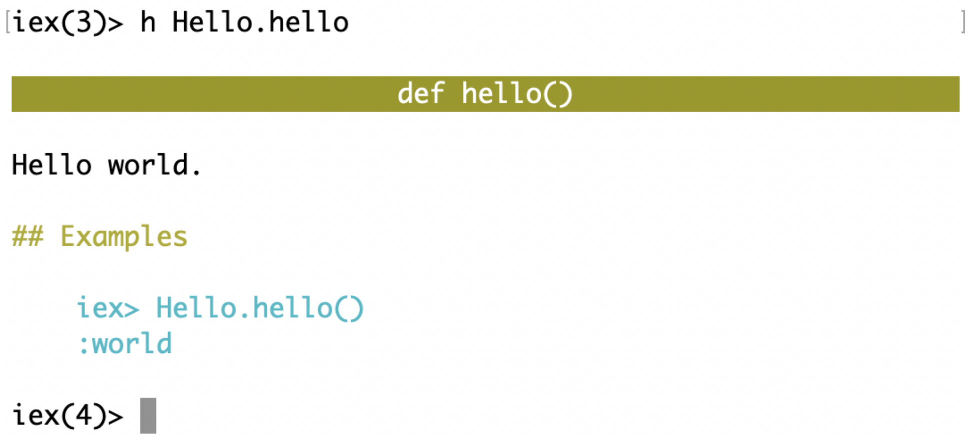
To interact with our program, we're going to need an interactive shell. Let's explore that now.

IEx: Interactive Elixir

Let's fire up an interactive console from the `/hello` directory:

```
iex -S mix
```

IEx is interactive Elixir, and the `-S` flag starts Elixir with everything our project needs so we can call functions within our project. We can see the documentation for our function and call it:



```
iex(3)> h Hello.hello

                      def hello()

Hello world.

## Examples

    iex> Hello.hello()
    :world

iex(4)>
```

You can also get the Hello module information with h Hello as well:

```
iex(1)> h Hello

                                Hello

Documentation for Hello.
iex(2)> Hello.hello
Hello
```

We get help for the module, and then we call the Hello.hello function. To execute any function, you'll chase the module name with a period and then the function name.

Let's look into some more obscure Mix features.

Get a Past Value

For the most part, IEx has line numbers, as in the previous listing. We'll normally remove them because it's tough for readers to try to perfectly synchronize their output with ours. A nice feature of IEx is the ability to get the previous value of a line. In the previous listing, we executed the command Hello.hello(). Let's say we're lazy and want to retrieve that value without waiting the few microseconds for the code to complete. We can use the v() command, which stands for value. Pair that command with one of the line numbers in IEx, like this:

```
iex> v(2)
:world
```

Brilliant! In fact, we can retrieve any previous value.

IEx is a greatly underutilized feature for Elixir. It has many features we can use to make programming easier. In particular, Elixir is great for accessing Elixir documentation.

Explore Programs and Documentation

If we want to know the public functions we can call from a module (these functions are called *exported* functions), we can simply ask:

```
iex> exports Hello
hello/0
```

Elixir correctly shows us there's only one function in our module, hello/0 meaning the hello function with zero arguments.

We can get information about the last result in the console:

```
iex> v(2)
:world
iex> i
Term
  :world
Data type
  Atom
Reference modules
  Atom
Implemented protocols
  IEx.Info, Inspect, List.Chars, String.Chars
```

The last value was an atom, with the value :world, and we can see the module that deals with other Atom modules. We can also see the protocols, and we'll get to those details later.

IEx also lets you build projects without exiting. Let's find out how.

Recompile Programs

We can also recompile our program:

```
iex> recompile
:noop
```

We get a :noop, meaning no-operation, because the file doesn't need compilation. We're going to be spending a ton of time in the console because it's an excellent way for you to follow along as we work through the chapter.

We've looked at one of our three generated program files. Let's move on to another, the test. Leave IEx behind for now. Unix and OSx developers will hit command-c twice to exit the IEx shell. Since Windows installations diverge, you'll need to follow the directions for your platform to exit IEx, perhaps Control-g followed by q.

Let's move on to tests.

Built-in Testing

Often, new programming languages lack the robust tools that mature languages have. Almost from the beginning, Elixir was created with a full set of tools. This tooling has accelerated Elixir's adoption. Two of the most critical parts of that infrastructure are tests and tasks.

The Elixir tooling includes a built-in test environment called ExUnit. Believe it or not, you've already created your first test. mix new created it for you. Let's run that test now:

```
[hello] ➔ mix test
Compiling 1 file (.ex)
Generated hello app
..

Finished in 0.02 seconds
1 doctest, 1 test, 0 failures
```

We ran our test, and got two dots. Let's see what's making all of the noise:

```
defmodule HelloTest do
  use ExUnit.Case
  doctest Hello

  test "greets the world" do
    assert Hello.hello() == :world
  end
end
```

We have a module, a new directive called Use, a doctest[3] directive which we won't cover here, and a test. Interestingly, without the use ExUnit.case, the test "greets the world" wouldn't be valid Elixir. The use directive is inviting Elixir to import some *macros*. Think of macros as code that writes code through the use of a template. We'll cover macros a little bit later in this book.

You can see exactly what's happening in the test. The test line defines a test, and assert compares the :world result we expect against what actually happens when we call Hello.hello(). If they match, the test succeeds.

Let's break a test to see what happens when a test fails. Open up your file test/hello_test.exs, and drop this new test in:

3. https://elixir-lang.org/getting-started/mix-otp/docs-tests-and-with.html

```
test "fails" do
  assert Hello.hello() == :World
end
```

```
[hello] ➔ mix test
..

  1) test fails (HelloTest)
     test/hello_test.exs:9
     Assertion with == failed
     code:  assert Hello.hello() == :World
     left:  :world
     right: :World
     stacktrace:
       test/hello_test.exs:10: (test)


Finished in 0.02 seconds
1 doctest, 2 tests, 1 failure
```

And we get a failure! Rejoice in failures because they are bugs in your code that you've found. In functional languages, they're usually repeatable so you can debug and fix them.

Since this failure is an artificial one, delete the broken test and let's move on. We'll circle back to tests when it's time to run some exercises.

We've used mix to build our project, start a console, and run tests. Let's see what else it can do.

Custom Mix Tasks

mix test is a custom task, one that the Elixir team has written for us. mix is a tool that's built in Elixir, and it's an open one. From the beginning, the Elixir team concentrated on tools that the earliest developers could use to build a growing community library. We can write our own tasks as easily as we write any Elixir function. Let's build a trivial remind task to help us remember how to do some simple tasks.

As you might expect, we're going to create a new file. Because it's part of our application, we'll put it in the lib directory. We'll create one subdirectory called mix within lib, and another subdirectory called tasks within mix. Then, we'll create a new task called remind to hold all of the details about running mix that we can't remember.

The Mix.Task Contract

To get started, we're going to have to fulfill a contract. We need to name our task the right way and write a function called run. We'll explore how to specify such contracts later when we cover Elixir behaviours and protocols. (Note that Elixir uses the British spelling of behavior.) For now, simply create a file called lib/mix/tasks/remind.ex and key this much in:

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

  @shortdoc "Return a reminder for how to use our project."
  def run(_) do
    IO.puts(
      """
      iex -S mix
      -> Start a console with our project

      mix test
      -> Run tests

      mix deps.get
      -> Fetch dependencies.
      """
    )
  end
end
```

We define a new module. The periods are actually namespaces, meaning a module called Mix.Helper is different from one called Hex.Helper, and all of the functions inside them are different too. It's good for each organization to use a different namespace so two organizations won't accidentally create the same functions in the same modules.

We want our module to be part of the Mix module, and we want it to be a Task. Then, we create a unique module name that describes what we want to do, Remind.

Next, we have the use directive. Remember, the use directive tells Elixir we're going to use macros that provide the features we need. In this case, the features make creating mix tasks easy.

Then we have some documentation specific to mix tasks called the @shortdoc.

Finally, we use the function IO.puts to print out a multi-line string. When we're done, we have a simple little mix task.

Run Your Custom Task

We can run our custom task like any other mix task. Let's call mix remind like this:

```
[hello] ➔ mix remind
Compiling 1 file (.ex)
```

```
iex -S mix
-> Start a console with our project

mix test
-> Run tests

mix deps.get
-> Fetch dependencies.
```

Notice that Mix knows it must build the program first. Then, it dutifully prints out our instructions.

Access Task Shortdoc Documentation

Because we provided some short documentation, we can see our task show up in help. Type:

```
[hello] ➔ mix help
mix                    # Runs the default task (current: "mix run").
...
mix remind             # Returns a reminder for how to use our project.
...
```

Nice. We can see our task is a full citizen. We also get spelling correction by default:

```
[hello] ➔ mix rem
** (Mix) The task "rem" could not be found. Did you mean "remind"?
```

That's a lot of extra benefit for such a short program. We can quickly add our own custom tasks to our own projects.

We've barely scratched the surface of what you can do with mix. Your task might depend on other tasks; you can take advantage of other parts of your program or the environment. The point is not to give you a comprehensive guide for building mix tasks but to underscore the importance of this kind of tooling on the community. By making mix so open and extensible, the Elixir team has encouraged others to make convenient tasks available to their users.

Let's look at one final part of our ecosystem, working with dependencies.

Sound Dependency Management Fuels Adoption

As you begin to build projects in Elixir, you'll begin to rely more and more on the work of others. One of the first working tools in Elixir was a good package manager. Let's see how that works.

Each Elixir project has a configuration in a file called mix.exs. Open up mix.exs so we can poke around. You'll see three functions. Start with the def project(...) function:

```
def project do
  [
    app: :hello,
    version: "0.1.0",
    elixir: "~> 1.10",
    start_permanent: Mix.env() == :prod,
    deps: deps()
  ]
end
```

This function is the description of your project. It's stored in a list called a Keyword list, made up of keys like :app and values like :hello. It has the name, version number, and other details that make up your program. Notice the line that says deps: deps(). That line calls a function called deps() and places the result in the key :deps. We'll skip the application function for now, noting only that it defines the other full application ours depends on.

Now, let's look at that deps() function:

```
defp deps do
  [
    # {:dep_from_hexpm, "~> 0.3.0"},
    # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git",
    # tag: "0.1.0"}
  ]
end
```

This list will eventually hold the dependencies in our application. So far, there aren't any. The two lines are simply comments that tell us the types of dependencies our applications can use.

Let's say we want our application to be able to greet the world in the JSON format. In case you've not seen it before, JSON stands for JavaScript Object Notation and is a file format that's often used on the web. Let's say we don't want to write our own JSON parser. We want to use the Elixir package manager called Hex.

So, we point our browser to Hex[4] and search for json. The first result has been downloaded 3,000,000 times, so we'll pick that one. After a couple of clicks, we find the documentation on github[5] and read it.

Per the instructions there, we plug in our dependency on mix.exs, deleting the comments as we go, like this:

```
defp deps do
```

4. hex.pm
5. https://hexdocs.pm/jason/readme.html

```
  [
    {:jason, "~> 1.1"}
  ]
end
```

Then, we fetch dependencies with mix, like this:

```
[hello] ➔ mix deps.get
Could not find Hex, which is needed to build dependency :jason
Shall I install Hex?
(if running non-interactively, use "mix local.hex --force") [Yn] Y
* creating /Users/batate/.asdf/installs/elixir/1.10.1/.mix/archives/hex-0.20.5
Resolving Hex dependencies...
Dependency resolution completed:
New:
  jason 1.1.2
* Getting jason (Hex package)
```

We say Y to the prompt when mix asks us if we want to install Hex since we've not yet done that, and then we let mix do the work. It installs the dependency and we're good to go…

but what happened here?

It turns out that we fetched the dependency json, together with any dependencies it needs to run. Like a cargo ship, we can see a manifest of all the goods that we moved in a file called mix.lock:

```
%{
  "jason":
    {:hex, :jason, "1.1.2", "b03ded...64afe7", [:mix],
    [{:decimal, "~> 1.0", [hex: :decimal, repo: "hexpm", optional: true]}],
    "hexpm", "fdf8...c3afe"},
}
```

I've shortened some of the longer tokens and reformatted things a bit, but otherwise, the file is untouched. It turns out that mix only had to fetch a single dependency. This dependency is an Elixir project, and our project fetched it. The next time we build our project, Elixir will automatically build version 1.1.2 of jason right along with it.

You can even see the source code. It's in a directory predictably called deps/jason:

```
[hello] ➔ ls deps/
jason
```

We can open that project and edit the code as if it were our own. (Beyond editing the code for a little debugging, that's not a good idea because the dependencies will often need to be fetched and rebuilt several times through the course of a busy project.)

Let's use the dependency in our project. Open up lib/hello.ex and add this new function:

```
def hello_json do
  Jason.encode!( %{ hello: :world } )
end
```

Now, try it out:

```
[hello] ➔ iex -S mix
...

==> jason
Compiling 8 files (.ex)
Generated jason app
==> hello
Compiling 2 files (.ex)
Generated hello app

...
iex> Hello.hello_json
"{\"hello\":\"world\"}"

iex> IO.puts Hello.hello_json
{"hello":"world"}
:ok
```

And it works! In a few short minutes, we've built a completely useless translation of an Elixir map to JSON. Along the way, we got to poke into the Elixir world of dependencies!

Your Turn

We've spent time with the tools and structures that make Elixir such a compelling language for functional programming beginners and library creators alike. Let's sum it all up.

It's All About the Alchemy Lab

The Ruby programming language is perhaps unique in how effectively it embraced early adopters. Elixir developers, including the creator himself, came from this community and provided excellent tooling from the beginning. The mix tool provides the toolbox for managing development projects. We used mix tasks to build our project, run tests, and get a list of other tasks. We then built our own task.

We spent a good deal of time in IEx. It provided an excellent tool for running and debugging bits of code. We spent time trying small examples and even compiling programs.

We then spent some time integrating a dependency into our program. We included a Hex dependency to add a JSON representation for our short program.

Now, it's your turn to use some of these tools.

Try It Yourself

In this section, we're going to focus on getting most of the tooling off of the ground. We'll list only the easiest Elixir track problem, the `Hello` problem, so you can get the infrastructure working. We have a single Exercism problem. To install an exercise, you'll first install Exercism,[6] and then join the Elixir track. Then, you'll download the problem. For example, to download a problem called `hello`, you'd do this:

```
`exercism download --exercise=hello --track=elixir`
```

These *questions* will help you understand where to go for more information.

- How would you enable command history in IEx?
- How do you start a remote shell in IEx?
- Can you find a Hex package to run performance benchmarks in Elixir apps?
- Can you find an Erlang package on Hex to process web HTTP commands?

These *easy* problems deal with establishing infrastructure.

- Install Elixir. If you've not done so yet, get busy! Install Elixir.
- Exercism `hello-world` problem. This will get you started with Exercism exercises.

These *medium* problems deal with dependencies.

- Find out what's required to package your own dependency on Hex.
- How would you create a `mix` project that requires a supervisor? (Check the documentation.)

This *hard* problem will help you with building `mix` tasks.

- Build your own `mix` task to count the number of files in the project.
- Build an IEx representation of a module. Hint: use the `Inspect` protocol.

6. exercism.io

Next Time

In the next chapter, we're going to focus on code organization and data. We'll work through the standard data types and establish how to explore them within IEx.

CHAPTER 2

Data and Code Organization

In the first chapter, we looked at how to build and navigate your Elixir project. In this chapter, you'll learn how to organize your project's code around elements of data called *data types.* Programming languages each have different patterns for building things, and Elixir is no different. Elixir will reward you if you organize your code in a certain way. In the next few chapters, we'll introduce the major Elixir data types and some ideas for building up the modules, data, and functions that make up your applications, layer by layer.

Rather than focus entirely on data types and operators right out of the gate, we'll do a quick presentation of four concepts and explore them in greater detail in the context of each of the major Elixir data types. The concepts are these:

Structure
: The primary structure for the data type, and what it looks like under the hood.

Operators and functions
: The main tools we'll use to manipulate a data type.

Pattern matching
: The Elixir tool to access and compare across data types.

Writing code
: We'll use most data types in a bit of code.

These concepts will be similar for each major data type, but we'll introduce more nuanced techniques as we go. We'll focus on some core Elixir data types including atoms, Booleans, numbers, and characters. Since the data types are simple, we'll get to introduce some IEx features that will aid us as we explore. Then, we'll layer on some simple Elixir code constructs and pattern matching. Finally, we'll write a little bit of code.

Let's get right to it!

Atoms, Pattern Matching, and Erlang Access

We're going to start with atoms, a data type that might be confusing to Java, C, or C# developers. If you're a Ruby developer, atoms are like symbols. If you come from Erlang, the data type is the atom.

Exploring a Type in IEx

Open up an IEx console and we'll get right to work. So, what's an atom?

```
iex> :atom
:atom
```

Atoms are names for concepts. Some good examples are :north, :string, and :yellow. Preface an atom with a colon, followed by a word starting with an alphabetic character.

You're not limited to atoms with simple characters. By placing double quotes around the text to the right of the colon, you can make more complex atoms like :"blue-green" that would otherwise be invalid. In Elixir, we won't do too much of that. We'll focus on atoms that start with a lower-case character and contain alphanumeric characters, plus possibly some additional punctuation. Let's go back to the console and explore a bit.

Once you've typed :atom, you can use the console to get more information about it. Type i for info, like this:

```
iex> i
Term
  :atom
Data type
  Atom
Reference modules
  Atom
Implemented protocols
  IEx.Info, Inspect, List.Chars, String.Chars
```

We can see the term we entered is :atom, its type is Atom, the main module for the data type is Atom, and it supports some interfaces called *protocols*. These include the info that we used to get this information, inspection for debugging, and translations to two common types, Charlist and String.

Atom is a module, and modules contain functions. In Elixir, the central theme of a module is often a data type. Modules have public functions and private functions. In Erlang, public functions are ones we explicitly *export*. Instead, Elixir exports all functions that we don't declare as *private*. Let's find the exported functions for Atom, like this:

```
iex> exports Atom
to_char_list/1     to_charlist/1      to_string/1
```

That's simple. We see the names of three functions. Each has a /1 after it. The /1 is the *arity*, meaning the number of arguments the function takes. In Elixir, it's customary to make the first argument the best data type for the underlying module. In our case, that's an atom.

Let's get a little more information about that last function:

```
iex> h Atom.to_string

                            def to_string(atom)

  @spec to_string(atom()) :: String.t()

Converts an atom to a string.

Inlined by the compiler.

## Examples

    iex> Atom.to_string(:foo)
    "foo"
```

Now that we know enough to use the function, let's do so:

```
iex> Atom.to_string(:atom)
"atom"
```

This pattern of Elixir exploration is a common one. We'll get a term in the console, obtain more information, examine the module through help or by getting exports, and finally explore a function. Let's look at some of the other things we can do with atoms.

Atoms, Equality, and Matching

Atoms and equality work as you might expect:

```
iex(6)> :atom == :eve
false
```

The == operator checks for equality, and only identical atoms are equal. That's not too interesting, but you might run into a surprise if you use the = operator:

```
iex(7)> :atom = :eve
** (MatchError) no match of right hand side value: :eve
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    ...
```

If you're not familiar with a pattern matching language, this example might surprise you. The = operator means match, which is used to make the thing on the left match the thing on the right. Let's say we want to bind a variable to the value of the atom. We do it like this:

```
iex(7)> a = :atom
:atom
iex(8)> a
:atom
iex(9)> a = :eve
:eve
iex(10)> a
:eve
```

We bind the variable a to the atom called :atom. Then we rebind the variable a to the atom :eve. Elixir is an immutable language, meaning values can't change. You're actually getting an *entirely new variable that's also called a*.

If you wanted to use = to match a variable on the left side of an expression rather than rebind, you'd use the ^ operator, called the *pin* operator, like this:

```
iex(10)> a
:eve
iex(11)> ^a = :atom
** (MatchError) no match of right hand side value: :atom
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
```

That's more like what we expected. We tried to match the old binding of the variable a (:eve) against :atom, and it failed.

Let's create a new application called graphics. Create a new mix project with mix new graphics, and change into the directory. We'll use that project to build a few rough features dealing with the data types we encounter.

You can use atoms and pattern matches when you write functions. Let's add a file called lib/color.ex to the hello application we built in the previous chapter.

Now let's make a short program that represents colors in lib/color.ex:

```
defmodule Color do
  def hex_code(:white) do
    "#ffffff"
  end
  def hex_code(:black) do
    "#000000"
  end
end
```

Cool. We create a module called Color. As is customary in many Elixir modules, our module will contain functions that deal with one central idea, colors. We'll represent the colors as atoms. As often as we can, within this module, the first argument of each function *will be the data type for our module*, an atom that represents a color.

We can try out the program. Open up the console for your project with iex -S mix and use your program:

```
iex> Color.hex_code(:white)
"#ffffff"
```

Our two function heads have the same name, hex_code. When we call the function, Elixir will return the first function that matches the argument we provide.

Let's see what our function exports:

```
iex(1)> exports Color
hex_code/1
```

The first hint that something interesting is up is that we have two function heads but one export. It turns out that two functions with the same arity are the same.

Now, we can try our functions with the values we expect:

```
iex(2)> Color.hex_code(:white)
"#ffffff"
iex(3)> Color.hex_code(:black)
"#000000"
```

Through pattern matching, Elixir picks the right function clause. Now, let's try a color we don't expect:

```
iex(4)> Color.hex_code(:blue)
** (FunctionClauseError) no function clause matching in Color.hex_code/1

    The following arguments were given to Color.hex_code/1:

        # 1
        :blue

    Attempted function clauses (showing 2 out of 2):

        def hex_code(:white)
        def hex_code(:black)

    (hello 0.1.0) lib/color.ex:2: Color.hex_code/1
```

Elixir tells us the color and the two function heads it tried.

We can tighten up our code to provide a better error. Add this function head to our program:

```
def hex_code(_other) do
```

```
  raise "unsupported color"
end
```

And try it out:

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> Color.hex_code(:yellow)
** (RuntimeError) unsupported color
    (graphics 0.1.0) lib/color.ex:9: Color.hex_code/1
```

We recompile our code and then provide an unsupported color. This time, our function raises an exception with more information than a match error.

Now, you've seen three ways to use a pattern match: to *bind* a variable, to *compare* values, and to *select* the code we want to execute. As we get further, you'll see still more ways to use pattern matching.

For now, we'll put patterns aside and move on to some other uses for atoms within Elixir.

Erlang Modules Are Atoms

Another way Elixir uses atoms is to represent modules in Erlang. This behavior is a little jarring at first, but let's explore a little bit. To see how this works, enter this strange line of code:

```
iex(11)> exports :erlang
bitsize/1
check_process_code/2
check_process_code/3
...
```

Yes, those are functions exported by the :erlang module! You can access all Erlang modules in this way. As you might expect, there's a special function to find the total number of atoms your application can support:

```
iex(12)> :erlang.system_info(:atom_limit)
1048576
iex(13)> :erlang.system_info(:atom_count)
10862
```

That's cool! We can access the root Erlang system_info function. In this example, there's also a pretty significant nugget of information.

Atoms are not garbage collected. If you do things like use atoms for identifiers or create them from user input, *you can run out of space for atoms*, and if you do, you'll crash your virtual machine! Use atoms, but use them correctly, for naming finite concepts.

Atoms as Booleans

As you might expect, Elixir uses atoms for several of the key concepts of the language, Booleans included. Check out both the term we type and the term that's returned:

```
iex(14)> :true
true
iex(17)> true == :true
true
iex(18)> false == :false
true
```

The Booleans in the language are represented as atoms, and you can freely substitute :true for true in Elixir. We've barely scratched the surface in Elixir data types, but we've only covered atoms! Take heart, though, we're about to pick up speed dramatically.

We're not going to spend a ton of time going over primitive data types and operators in excruciating detail. Instead, we'll let you cover the details of the dozens of operators on your own. We'll give you the highlights.

First, let's look at Booleans.

Booleans and Truthy Expressions

Elixir has two Boolean values, and you've already seen they're expressed as atoms. Here's a quick overview of Booleans in Elixir:

- Elixir allows pure Boolean with and, or, and not.
- Elixir allows *truthy* Boolean operations à la C and Ruby with && and ||.
- Elixir also allows bitwise Boolean operators, but you need a library.
- There's excellent documentation[1] for the rest.

This tiny error provides some quick proof of these concepts in IEx:

```
iex(1)> nil && true
nil
iex(2)> nil and true
** (BadBooleanError) expected a boolean on left-side of "and", got: nil

iex(2)> nil || true
true
iex(3)> !nil
```

1. https://elixir-lang.org/getting-started/basic-operators.html

```
true
iex(4)> not nil
** (ArgumentError) argument error
    :erlang.not(nil)
```

In Elixir's truthy expressions, nil, :false, and false are false. Everything else is true. A commonly used trick in Elixir takes advantage of the || operator and true.

Now that we've seen them in action, let's look at numerics and comparisons.

Numerics Favor Utility over Performance

Elixir supports a couple of important numeric types, most prominently floats and integers. Elixir doesn't support small and big ints out of the box, opting to convert integers to use more bytes as needed:

```
iex(7)> big = 256 * 256 * 256 * 256
4294967296
iex(9)> huge = big * big * big * big
340282366920938463463374607431768211456
iex(10)> huge * huge * huge * huge
1340780792994259709957402499820584612747936582059239337772356144372176
4030073546976801874298166903427690031858186486050853753882811946569946
433649006084096
iex(11)> i
Term
  1340780792994259709957402499820584612747936582059239337772356144372176
  4030073546976801874298166903427690031858186486050853753882811946569946
  433649006084096
Data type
  Integer
Reference modules
  Integer
Implemented protocols
  IEx.Info, Inspect, List.Chars, String.Chars
```

The biggest one is huge, but it's still only an integer. *This numeric representation is a compromise of utility over performance.* If you'd like you can dig into the Integer module to find some of the functions you can use with integers, but remember Erlang has many more.

Note that Elixir focuses on floats, but decimals of fixed precisions are available as a library. Elixir has a bunch of operators for dealing with numerics, and you can look them up. As a language that's been around for 40 years, Erlang has even more. We'll call out a few highlights.

Some of those operators are comparisons. Elixir actually has two checks for equality, == and ===. The second is stricter:

```
iex(11)> 1 == 1.0
true
iex(12)> 1 === 1.0
false
```

These numbers are mathematically equivalent so the first comparison matches. They're not structurally identical, so the second one doesn't match.

Before we move on, there's one usage of an integer that might seem strange to you. Let's look at characters.

Characters Are Code Points

The process of making code safe for multiple languages at once is *internationalization*. If you're familiar with it, you know that every single character in a string is called a code point. In Elixir, a code point is represented with a number. Here's how it works:

```
iex(18)> ?a
97
```

If this were the only surprising behavior, I'd probably let you read about it yourself, but there's a strange remnant from Elixir's dependence on Erlang. Namely, Elixir can't always tell the difference between lists of numbers and character lists!

Here's what I mean:

```
iex(22)> [99, 97, 116]
'cat'
```

This happens because Erlang can't tell the difference between a list of integers and a list of characters, so it guesses. Sometimes these lists of characters can be a bit disconcerting. If it ever happens to you, tack on a zero, which isn't a printable character:

```
iex(22)> [99, 97, 116]
'cat'
iex(23)> list = [99, 97, 116]
'cat'
iex(24)> list ++ [0]
[99, 97, 116, 0]
```

Problem solved. The `Charlist` is surrounded by single quotes. Under the hood, linked lists are made up of a slot for data and another for the pointer to the next bit of data. You should know that this type of text data isn't an efficient format for long bits of data like text. Elixir does have a more efficient representation of strings, called the Binary type. We'll cover it in more depth later. For now, we'll touch on it briefly.

Say we wanted to represent three numbers in the most efficient way. We put them in memory, back-to-back. We'd use a binary, surrounded by double angle brackets, like this:

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

These numbers are in one big block of memory, and that's a pretty efficient way to store data in Elixir as long as you don't need to modify it. You can probably guess what would happen if we put characters in those angle brackets:

```
iex> <<?c, ?a, ?t>>
"cat"
```

Rather than putting characters into a linked list like Charlists do, Elixir puts each character in a Binary back to back in memory. Strings are *binaries*, and charlists are *lists*. In Elixir, since lists have overhead, we'll choose to deal with text data using strings.

We're ready to move on. Now, let's compare some numeric values to look into Elixir's main control structures.

Elixir Deemphasizes Control Structures

For the most part, Elixir doesn't have as many data structures as typical procedural or object-oriented languages. Instead, programmers will use other features such as functions or pattern matching. We'll look at three now: if/unless, case, and cond. We'll save with for later.

Compare with if and unless

You've already seen one way to do comparisons in Elixir, and it's often a good way. In the Color module, we used pattern matching within our function head to compare values. The performance of this kind of comparison is very good.

Another way to do such comparisons is with the if statement. Go to your IEx session. Let's assign a variable to compare against, like this:

```
iex(16)> x = 10
10
```

Now, let's use x for some comparisons. There's a one-line form that looks like this:

```
iex(17)> if x == 11, do: :tis_true, else: :tis_false
:tis_false
```

That's a strange syntax for a function! What's going on?

It turns out that if is actually a macro, with syntax like a function call. A couple of things are happening all at once to give us this syntax at the end of the statement:

```
, do: :tis_true, else: :tis_false
```

First, all of that code is actually the second argument to the if. That's why we need the first comma.

Second, the rest of the statement is a keyword list. In Elixir, if the last argument of a function is a key-value list, you can eliminate the square brackets that would normally surround it. That's why we don't have to type , [do: :tis_true, else: :tis_false].

Finally, when the first elements of a keyword list or map are atoms, you can reverse the atom and eliminate some syntax. Here's what a keyword list looks like without that trick:

```
[{:do, :tis_true}, {:else, :tis_false}]
```

Usually, you won't use this shortened version. Elixir has some sugar that lets you use this form instead:

```
if x == 10 do
  :tis_true
else
  :tis_false
end
```

Whew. That's prettier. You can try it in the console as well. I've broken it out to show the syntax without the extra line numbers and other noise.

Elixir supports an unless that works exactly like an if, but in reverse. These two functions aren't the only ways you can compare data.

Cond

Sometimes, you need to make several comparisons at once. Elixir lets you do so with the cond statement. Cond statements are like if statements with several different conditions. Elixir will take the first one that's true.

For example, we might have implemented our Color.hex_code function like this:

```
def hex_code(color) do
  cond do
```

```
    color == :white ->
      "#ffffff"
    color == :black ->
      "#000000"
    true ->
      :error
  end
end
```

We have three conditions, but we could have had more or less. The only rule is that at least one of the expressions on the left must evaluate to a truthy value. The last true serves as our default. It'll catch the execution if none of the other clauses match. Elixir doesn't require it, but it's a good habit to have at least one catch-all value like this one.

This code works, but it's an awkward control structure to use because it often forces us to repeat small bits of code, such as the color == bit. Elixir is a language loosely based on Prolog, and if you went through our Prolog language, you understand comparisons reward matching. The cond statement takes the first statement that evaluates to true. Let's see what case does.

Case

While the if, unless, and cond statements use Booleans to control flow, the case statement works with pattern matching, as our Color.hex_codes did. Instead of matching function heads, we'll move the matching conditions inside of our function, like this:

```
def hex_code(color) do
  case color do
    :white ->
      "#ffffff"
    :black ->
      "#000000"
    _ ->
      :error
  end
end
```

That code is much better because it eliminates the repetitive comparison against color. Elixir will test color, and stop at the first condition that matches. Notice that instead of true, our catch-all clause uses an underscore, which in Elixir matches anything.

These constructs may seem like a thin foundation for programming, but that's not all we have to work with. Elixir is a functional language, and you'll primarily rely on functions you'll compose in various ways to do the bulk of your work.

This is a great place to stop and take stock. It's time to put some of what you've learned to work.

Your Turn

This chapter introduced the primary primitive data types in Elixir. As you work through this section, you'll get a chance to build your own projects and put Elixir through its paces.

Elixir Organizes Code Around Data

Elixir's modules are all organized around data. The standard data types include atoms, Booleans, and numerics. Where possible, the first argument in a function is determined by the containing module. For example, `Atom.to_string/1` takes an atom first.

Atoms represent concepts, and pattern matches are expressions that attempt to match the structure and content of data. Pattern matches work as individual expressions and also in function heads.

IEx is a great tool for exploring pieces of data and drilling into the underlying data types. With the `i` command, you can get more information about the underlying data type. Each main data type has an associated module, and you can explore that module in IEx.

Boolean values are atoms under the hood. They work much like they do in other languages. Elixir has a few types of numerics, including integers and floats. We didn't cover the huge number of expressions, leaving that exploration to the reader. You'll get plenty of practice with primitives in the following exercises.

Try It Yourself

In this section, we're going to focus on exercises related to primitive data and Elixir control structures. We'll mostly use Exercism.io. To install an exercise, if you haven't done so, you'll first install Exercism, and then join the Elixir track. Then, you'll download the problem. For example, to download a problem called `hello`, you'd do this:

```
`exercism download --exercise=hello --track=elixir`
```

Let's look at a few problems related to data types and flow.

These *easy* problems deal with establishing infrastructure.

- leap: Implement the rules for a leap year.
- bob: We look ahead to strings. This problem uses Elixir cond statements.
- rna-transcription: We look ahead to breaking down strings, and working with lists.

These *medium* problems deal with dependencies.

- roman: This problem looks ahead to joining strings, but the bulk of the work is done with control structures.
- hexidecimal: You'll need a few concepts we haven't covered, including strings, Enum.map, Enum.sum, and the function String.graphemes.
- sieve: Find prime numbers. You'll need the Enum module, possibly with map or filter.
- Represent an integer as a hex, octal, and binary number.

Next Time

In the next chapter, we're going to ramp up the complexity of the data we can handle. We'll first address tuples, and then we'll move on to lists.

CHAPTER 3

Tuples and Functions

In our journey so far, we've learned to build and manipulate Elixir projects and use primitive data types. The next few data types are for sequential data. Languages like Ruby have one dominant way to represent sequential data, the array. Elixir has multiple ways to represent sequential data. Tuples express fixed-length sequential data, and lists represent variable-length sequential data.

In this chapter, we'll begin our exploration of tuples. Along the way, we'll use tools and techniques to create, inspect, and use them. As you might imagine, central to those techniques will be functions and pattern matching. As usual, we'll explore each data structure in the console and we'll also begin to roll them up into advanced constructs.

Tuples, Deconstruction, and Pattern Matching

In Elixir, you use tuples to create lists of things with a fixed size. You represent a tuple with curly braces surrounding elements with commas between:

```
iex(1)> place = {:stockholm, :sweden}
{:stockholm, :sweden}
iex(2)> origin = {0, 0}
{0, 0}
iex(3)> white = {0xff, 0xff, 0xff}
{255, 255, 255}
iex(4)> success = {:ok, "result"}
{:ok, "result"}
iex(5)> failure = {:error, 401}
{:error, 401}
```

Each of these examples is a tuple. A point is an iconic example of a tuple. Erlang developers frequently use tuples to pair return codes with results. Notice that the elements of a tuple aren't necessarily the same type. For example, in a result tuple from a function, the first element is usually an atom describing the result, and the second element is the type the function returns.

The most important part of representing tuples is that *the position of an element in a tuple determines its meaning*. The parts of a place are city and country, points are expressed as x and y, and so on.

Another ergonomic consideration for tuples has more to do with the computer between your ears than the one running your code. Since we can't label tuple elements in any way, it's hard to read code with tuples longer than two or three elements, so keep them short!

Let's look at tuples in more detail.

Exploring Tuples

Staying with IEx for a minute, let's do our customary dive. Enter a line that returns a tuple, and then get its info:

```
iex(6)> i
Term
  {:error, 401}
Data type
  Tuple
Reference modules
  Tuple
Implemented protocols
  IEx.Info, Inspect
```

As expected, the module for working with tuples is, well, Tuple. Let's dive deeper:

```
iex(7)> exports Tuple
append/2
delete_at/2
duplicate/2
insert_at/3
to_list/1
```

Usually, you can get a better sense for working with a data type by looking at the exports of its primary module. Be careful, though. In this case, the help will lead you astray. You might think that tuples are variable-length constructs that you should transform with abandon. That's a dangerous assumption! Let's look at what might happen if you did so.

Best Uses for Tuples

The implementation of a tuple in Elixir is one slice of memory of a fixed size with no room for expansion. There are two significant ramifications of this implementation:

- Longer short-lived tuples are tough on garbage collection. Creating and freeing larger constructs breaks up memory.
- To change a tuple in any way, Elixir must create a whole new copy.

So, you should understand that changing or adding to tuples *is not idiomatic Elixir* because it's expensive to return a new, modified tuple. Tuples should be created once in their final form and then left alone! Elixir will reward you with better performance and friendlier code if you use tuples for structures that are more permanent and shorter. You'll be able to take better advantage of pattern matching and your code will run more efficiently.

Let's see a few ways to use pattern matching with tuples.

Pattern Matching

So far, we've used pattern matching with a whole atom or integer. We're going to broaden your repertoire a bit. Sometimes, you can use pattern matching to *deconstruct* a complex data type. Let's say you have a place you've chosen to represent with a two-tuple, like this:

```
iex(1)> place = {:austin, :tx}
{:austin, :tx}
```

In Elixir, you can access the various elements in a tuple with pattern matching, like this:

```
iex(2)> {city, state} = place
{:austin, :tx}
iex(3)> city
:austin
iex(4)> state
:tx
```

That's nice! We access the city and state from within our place tuple. We can also ignore either city or state, or even ignore both to match only tuples with two elements, like this:

```
iex(5)> {city_name, _state_name} = place
{:austin, :tx}
iex(6)> city_name
:austin
iex(7)> {_, _} = place
{:austin, :tx}
```

```
iex(8)> {_, _} = {:some, :thing, :else}
** (MatchError) no match of right hand side value: {:some, :thing, :else}
...
```

This code works exactly as you'd expect. We don't have to access the elements of a tuple in this way. We can use the function called elem/2 to return a tuple element with a zero-based index, like this:

```
iex(8)> elem(place, 0)
:austin
iex(9)> elem(place, 1)
:tx
```

That lays out the foundation. Let's see how we might use pattern matching in the context of a greater application.

Functions and Code Organization

One of the central themes of our programs so far is that we package functions that operate on like data together in a module. Let's create a file called point.ex. We'll have functions on points in this module. We'll strive to form functions that take points as the *first argument* of our functions, and where possible, our functions will return points as well.

Deconstruction in Function Heads

Let's say we wanted to take a point in the form {x, y} and move it one unit to the right. Knowing that elem(tuple, index) gives us an element of the tuple, we might decide to write this bit of tedious code:

```
defmodule Point do
  def right(point) do
    x = elem(point, 0)
    y = elem(point, 1)
    {x + 1, y}
  end
  ...
end
```

That's a typical program that we might see in Java or Ruby. We can do better in Elixir. We can deconstruct tuples in function heads, case statements, and other Elixir constructs, like this:

```
defmodule Point do
  def right({x, y}), do: {x+1, y}
  def left({x, y}), do: {x-1, y}
  def up({x, y}), do: {x, y-1}
  def down({x, y}), do: {x, y+1}
  def move({x1, y1}, {x2, y2}), do: {x1 + x2, y1 + y2}
end
```

Nice. We use the one-line function syntax that works well when we are expressing a single thought. In the function head, we deconstruct the tuple, picking off the x and y variables. Then, we return the updated point.

In move, we match on *both* arguments: an initial point and a vector defining the difference in x and the difference in y.

These functions are no longer tedious because we can let the function head do most of the work. Our code expresses code with a single thought on a single line.

Along the way, I've said if you build modules with functions returning the same kind of data first, you'll be rewarded. Here's some of the candy.

Using Pipes

So far, we've leaned away from operators and into code organization. We're going to briefly break with that approach to introduce the pipe operator. Open up your application in IEx with IEx -S mix, or issue a recompile command if it's already loaded.

Then, let's create a point and work with it a bit. The first thing we'll do is to import the Point module, like this:

```
iex(1)> import Po
Point    Port
iex(1)> import Point
Point
iex(2)>
```

You can see that I pressed the tab key after I typed import Po, and Elixir showed me the modules that were available for importing. Then, I imported Point, meaning I can fully access the module features as if they were functions in IEx.

Specifically, importing functions *adds them to your current module*. Create a point, and then call your function on it, like this:

```
iex(2)> origin = {0, 0}
{0, 0}
iex(3)> left(origin)
{-1, 0}
iex(4)> origin
{0, 0}
```

We create a point called origin, and then we create a new point that's moved left one place. Notice that when we check the value of origin, it hasn't moved! That's because Elixir is an immutable language. Our function called left creates a new point that's to the left of origin.

Now, what happens if we want to start at the origin, and then move the point like a knight on a chess board by moving right two and down one? The code is surprisingly awkward:

```
iex(5)> down(right(right(origin)))
{2, 1}
```

That code is…

disappointing. We are taking the origin, moving the result right one, moving that result right one more, and then moving that result down. Here's what pseudocode that describes that problem might look like:

```
origin
   right
   right
   down
```

That's much more readable. Interestingly, that's what the code looks like when we express the code as a pipe:

```
origin
|> right
|> right
|> down
```

In the console, it looks like this:

```
iex(7)> origin |> right |> right |> down
{2, 1}
```

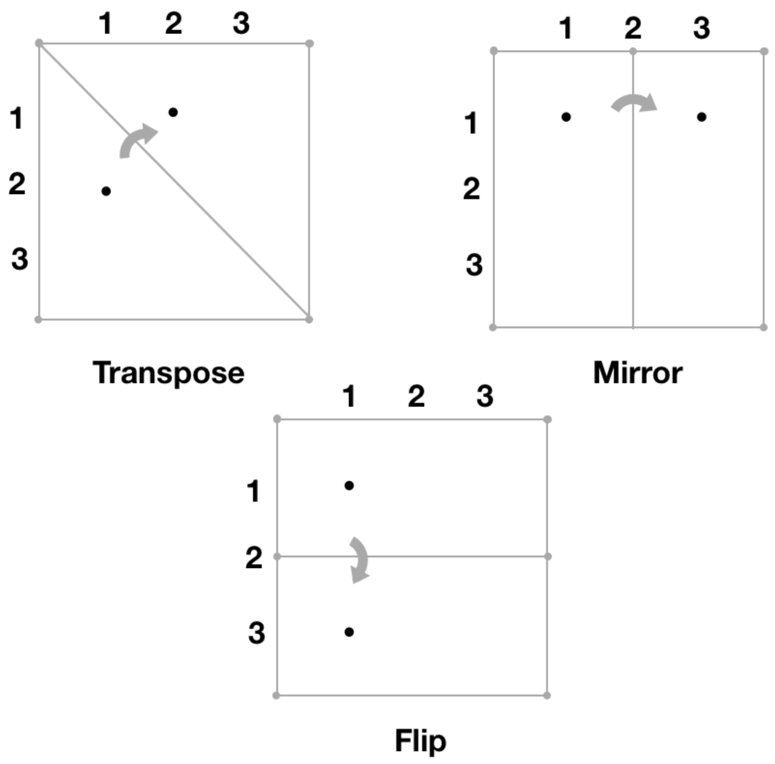
That code is much more satisfying. The |> operator uses the code *before* the pipe as the *first argument* for the function immediately *after* the pipe. We can write this string of code explicitly because we structured our code to:

- Take a point as the first argument
- Return a point

We structured our code in that way, and Elixir rewarded us. Let's build a couple of quick functions to mirror and flip a point on a canvas. We're going to do a little bit of math within our point that'll pay off down the line.

Reflect a Point on a Canvas

Reflecting a point on a canvas in various ways will give us some pretty cool capabilities down the line. The following figure shows the types of features we want to build:



All three of these functions reflect a point over a line. Transpose reflects the point over a diagonal line down the middle of the canvas, mirrors across a vertical line, and flips across a horizontal line.

Here's what it looks like to mirror a point horizontally. Add this function to point.ex:

```
...
def mirror({x, y}, w), do: {w - x, y}
...
```

Using tuple deconstruction, the function becomes trivial. We leave `y` intact, and mirror `x` by subtracting it from the width of the canvas.

Flipping a point vertically looks nearly identical. Add this function to point.ex:

```
...
```

```
def flip({x, y}, h), do: {x, h - y}
...
```

Perfect. We use a similar technique, but we manipulate y based on the canvas height instead.

With these two features, we'll later be able to flip and mirror polygons on a canvas.

Let's build in a few more tweaks that'll give us the ability to rotate polygons in 90-degree increments. Using a quick function called transpose which mirrors our shape along the line {0, 0} to {i, i}, we'll have all of the building blocks we need. Tack this function onto point.ex:

```
def transpose({x, y}), do: {y, x}
```

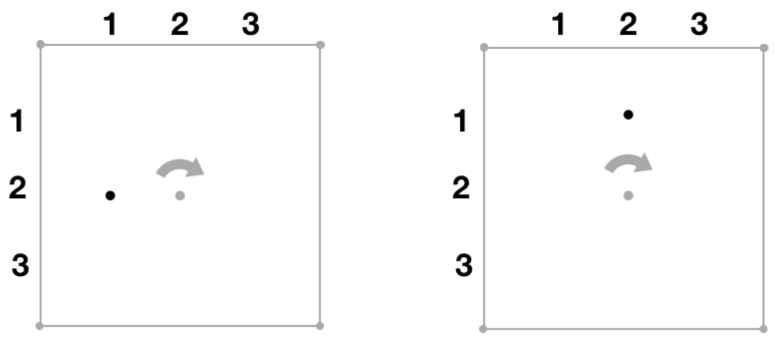
We reflect a point across the northwest to southeast diagonal with a transpose by merely reversing the x and y coordinates. Next, let's move a point:

```
def move({x, y}, {cx, cy}), do: {x + cx, y + cy}
```

We accept two tuples, one representing a point and one representing a vector with the changes in x and y. We'll use this later to move a bunch of points together. For example, we'll move a square by moving all of the corners at the same time.

Rotate Around the Canvas Center

There's an obscure geometry trick that lets us rotate shapes using the functions we've built so far. Don't worry too much about the technical details for now. We're rotating a point clockwise 90 degrees around the center of a canvas as in the following figure.



To rotate a point, we'll sequentially apply various transpose, flip, and mirror functions. There are similar formulas for rotating the point 180 and 270 degrees. With pipes, those rotations are surprisingly clear and beautiful. In point.ex:

```
def rotate(point, 0, _w, _h) do
  point
end
def rotate(point, 90, w, _h) do
  point
  |> transpose
  |> mirror(w)
end
def rotate(point, 180, w, h) do
  point
  |> flip(h)
  |> mirror(w)
end
def rotate(point, 270, _w, h) do
  point
  |> flip(h)
  |> transpose
end
```

These functions combine flip, mirror, and transpose in various ways to rotate a point. For example, transposing a point and then mirroring that result rotates a point 90 degrees around the center point of a canvas.

Now, you can try it out:

```
iex(4)> Point.origin |> Point.rotate(270, 5, 4)
{4, 0}
iex(5)> Point.origin |> Point.rotate(180, 5, 4)
{5, 4}
iex(6)> Point.origin |> Point.rotate(90, 5, 4)
{5, 0}
```

If you think about it, these points are exactly right! Imagine the point as the upper right corner of a rectangle, and you'll know the points are right.

Let's look at a few more examples of code organization before we move on.

Advanced Pattern Matching and Constructors

Most pattern matching uses simple deconstruction, but not all. Sometimes, Elixir code needs to make allowances for special cases. One special case is comparing elements within a function head. Let's take a few examples.

Comparison Across Tuples

Let's look at some special cases of pattern matching. The first is representing exact matches of primitive data within a data structure like a tuple. When you use primitive data in a function head, Elixir will try to match it exactly. An Elixir function to check and see if a point is the origin would look like this:

```
def origin?({0,0}), do: true
def origin?(_point), do: false
```

The function has two heads. The first matches the exact {0, 0} tuple, returning true. The second starts with an underscore. The compiler ignores arguments beginning with an underscore. We label our argument _point to document the interface, returning false. Let's look at another example that does a more advanced match. Let's say we wanted to match all elements where x and y are the same. We might decide to do a guard, like this:

```
def identity?({x, y}) when x == y, do: true
def identity?(_point), do: false
```

The function has one new concept, the guard. A guard in a function head will only match invocations where the values satisfy the guard. Guards can only work with macros, not arbitrary functions. We'll talk a bit about macros later. Note that some things are possible in guards, and others are not, because of the way Elixir compiles code.

The first function in this example deconstructs the point and uses the guard to check if x and y are equal. If so, the first clause matches, returning true. Otherwise, our catch-all guard matches, returning false.

There's a better way, though. We can use a variable more than once in a function head, and Elixir will make sure they're the same value, like this:

```
def identity?({i, i}), do: true
def identity?(_point), do: false
```

When Elixir is doing a match with two identical variable names within a function head, the data *must match exactly*. We use this quirk to match only tuples with the same element in the first and second positions. This function will match {5, 5}, but not {5, 4}. Can you explain why?

As we continue within Elixir, we'll use tuples often to handle result tuples. These return some version of {:ok, result} for success and {:error, code} or {:error, code, message} for failure.

Constructors

Before we move on, let's look at one final code organizational idea, the constructor. As you might expect, constructors are functions. They take standard building blocks as arguments and return the modules' data type. Let's take a look at a few constructors for our Point module. First, we can create a special function returning an origin. Add this function to your Point module:

```
def origin, do: {0, 0}
```

Constructors like this have the advantage of naming a concept. You can see how much more expressive the origin function makes the next example:

```
iex(9)> Point.origin |> Point.right |> Point.right |> Point.down
{2, 1}
```

Nice! Our intention is marvelously clear. The pipes represent *transformations*, and the *reducers* that do transformations between the same type are especially powerful. Our constructor is better than using a simple {0, 0} because it names the geometry concept: the point {0,0} is called the origin.

We can also use constructors to keep invalid data out of our functions:

```
def new(x, y) when is_integer(x) and is_integer(y) do
  {x, y}
end
```

If the user invokes new with x and y values that are both integers, the clause will match. Otherwise, it'll fail, like this:

```
iex(11)> recompile
Compiling 1 file (.ex)
:ok
iex(12)> Point.new(1, 2)
{1, 2}
iex(13)> Point.new(1, :two)
** (FunctionClauseError) no function clause matching in Point.new/2

    The following arguments were given to Point.new/2:

        # 1
        1

        # 2
        :two

    Attempted function clauses (showing 1 out of 1):

        def new(x, y) when is_integer(x) and is_integer(y)

    (hello 0.1.0) lib/point.ex:3: Point.new/2
```

We attempt to create a point, but it fails. Elixir tells us exactly what's wrong. That means as long as our code uses constructors that enforce valid data, the values we work with will be valid and won't have to include an oppressive amount of error data.

For now, let's move on. It's time to wrap up.

Your Turn

Tuples represent sequential data in Elixir. While tuples can be arbitrarily long, we'll tend to keep tuples short to make them easy to read. Also, while full APIs exist to transform tuples, for efficiency reasons, we'll create them once in their final form.

The functions we build work best if the functions all take data of the same shape, meaning tuples that all represent the same type. We built a function to work with points. Our points were tuples with an `x` and `y` coordinate.

We built two special types of functions. We used constructors to name concepts and keep data pure. We used reducers to perform operations that also returned points. Rather than mutating a point, our reducers created whole new points we transformed in some way.

Try It Yourself

In this section, we're going to focus on exercises related to tuples. The Exercism track doesn't have too many examples, so we'll make up a few problems for you to try.

Let's look at a few problems related to tuples. These *easy* questions deal with pattern matching and code organization.

- How would you represent a three-dimensional point on a tuple?
- Write a function to determine if a three-dimensional point `{x, y, z}` is on a plane identified by a `z` coordinate. For example, `plane({4, 5, 0}, 0)` would return `true` but `plane({1, 1, 1}, 0)` would return false.
- How would you represent a shopping cart with a tuple?

This *medium* problem starts with an Exercism problem and refactors it to use tuples.

- `triangle`: Solve this problem, then transform the program to take a three-tuple instead of three different sides.

Next Time

In the next chapter, we're going to address one of Elixir's most important data structures, the list.

We'll see you on the next page!

CHAPTER 4

Lists and Algorithms

In the previous chapter, we represented fixed-length sequences of elements, holding potentially different types, in *tuples*. In this chapter, we'll represent variable-length data, usually holding similar or identical types, with lists.

We'll add a few examples to our mini graphics application by incorporating multiple points to represent polygons.

Lists

When you want to model sequential data in Elixir, you have two choices, tuples and lists. We've already covered tuples. Tuples are great for short sequences of data with a fixed length, while lists can hold potentially longer sequences of data with variable lengths. While the same tuple has data that might vary in type, lists tend to hold items of the same type, but not always.

Along with maps, lists are among the most important Elixir data structures. Let's do our usual exploration in the console:

```
iex(1)> [1, 2, 3]
[1, 2, 3]
```

Elixir syntax represents lists in square brackets. Let's inspect the type. We'll look at the results in pieces:

```
iex(2)> i
Term
  [1, 2, 3]
Data type
  List
```

The term we typed is a list, and the module associated with that data type is the List module. Let's find out more:

```
Reference modules
```

```
  List
```

There are a couple of dozen important functions on List, most notably flatten, first, delete, and some conversions. When you get a chance, either run exports List or type List and press tab to see the functions you can call from the List module.

Let's dive deeper into a list's implementation.

List Construction

Elixir lists are linked lists, built head first. Take this list:

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

That's not simply one list, but four. Here's how we might represent it piece by piece:

```
iex> list1 = []
[]
iex> list2 = [3|list1]
[3]
iex> list3 = [2|list2]
[2, 3]
iex> list4 = [1|list3]
[1, 2, 3]
```

Interesting. The | operator in functional programming is sometimes called the cons operator, for list construction. [item|list] means *add item to the head of list*. Strangely, every list in Elixir is made up of *cons cells*. The [1, 2, 3] list shown here is made up of the three cons cells you see in the previous listing.

Looking at it another way, we can build a function called append, like this:

```
iex> append = fn lst, item -> [item|lst] end
#Function<12.128620087/2 in :erl_eval.expr/5>
```

We'll provide more information about anonymous functions in a bit. For now, know that we're building a function that appends an item to a list. Remember, each item is added to the head of list.

Then we start with an empty list and append each item using a pipe:

```
iex> []  |> append.(3) |> append.(2) |> append.(1)
[1, 2, 3]
```

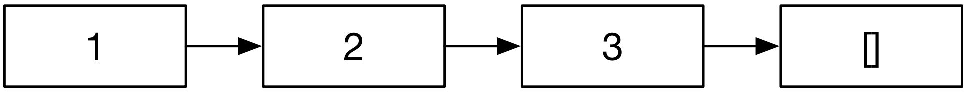
When the full pipeline runs, we're left with the final list. Note that we don't actually change a list. Instead, each call to append returns a new list.

That's not just the way we've built the list [1, 2, 3]. It's the way *every* list of [1, 2, 3] is represented, internally.

You might be wondering why you should care what happens under the hood. The reason is that since Elixir is immutable, when you do anything to an element of a list, each transformation gets progressively more expensive as you navigate away from the head of a list.

When you're searching for an element, accessing the head of a list is extremely fast. Accessing the last element of a list n elements long will take n steps. For a list n items long, we say algorithms like this one have an order of n complexity using a system called big-O[1] notation, while accessing an item at the head is on the order of 1, or O(1).

When you're updating a list, the same ideas apply. Changing the first element of a list *changes only the first element of a list,* leaving the rest of the list intact as you can see in the following image:



Each list item has a pointer to the rest of the list, but no pointer back to the previous element. That means changing the last element is much worse than changing the first. Since Elixir doesn't allow value updates, changing one element at the tail of a list forces your program to make a full copy of the list to preserve the pointers. Tail updates are expensive for long lists.

Let's dig a little deeper and explore how lists work with pattern matching.

The Enumerable Abstraction

We're not going to stop and loiter here because many of the most important functions you need for List are actually in other modules. Let's do a little detective work to find out why.

Let's look at the description of the List module:

```
Implemented protocols
  Collectable, Enumerable, IEx.Info, Inspect, List.Chars, String.Chars
```

That's the last bit of our inspection, and there's some interesting data in there. Lists are part of several protocols. Remember, the protocols listed here are essentially contracts that have functions you can use with List. Some, like Inspect and the IEx.Info we're using right now, are for debugging and information.

1. https://en.wikipedia.org/wiki/Big_O_notation

Others, like List.Chars, Collectable, and Enumerable, are important classes that work on collections of things in various contexts.

By far the most important of these is the Enumerable protocol. Elixir has two major ways for working with lists. Both work with the Enumerable protocol. Let's take a look at it now:

```
iex(6)> h Enumerable

                                Enumerable

Enumerable protocol used by Enum and Stream modules.
```

There we go.

The *eager* data structures are in the Enum module and are immediately allocated. The *lazy* ones are called Streams. These potentially infinite lists work with functions in the Stream module and are only allocated as we need them. In this chapter, we're going to focus on the eager data structures in Enum.

Enum has many of the functions you need for accessing lists of things:

```
iex(7)> h Enum

Enum

Provides a set of algorithms to work with enumerables.

In Elixir, an enumerable is any data type that implements the Enumerable
protocol. Lists ([1, 2, 3]), Maps (%{foo: 1, bar: 2}) and Ranges (1..3) are
common data types used as enumerables:

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

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

The Enumerable protocol defines things that can be enumerated. The bulk of the implementations are in Enum for dealing with eager lists and Stream for dealing with lazy lists. You may notice the Enum functions all take an enumerable as the first argument. Let's look at a few examples:

```
iex> Enum.sum([1, 2, 3])
6
iex> Enum.count([1, 2, 3])
3
```

We get the total for a list, or count the elements in a list. We'll look at more enumerable functions soon. For now, let's look at the next building block for Elixir list code, the pattern match.

Pattern Matching and Lists

Matching tuples is easy in Elixir because tuples have fixed sizes. For example, our points all have two elements. Lists are a bit trickier to match because they can be any length. Let's match a few lists.

```
ie> list
[1, 2, 3]
ie> [head|tail] = [1, 2,  3]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
iex> [] = [1, 2,  3]
** (MatchError) no match of right hand side value: [1, 2, 3]
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    (iex) lib/iex/evaluator.ex:257: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:237: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:215: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:103: IEx.Evaluator.loop/1
    (iex) lib/iex/evaluator.ex:27: IEx.Evaluator.init/4
iex> [first, second|rest] = list
[1, 2, 3]
iex> second
2
iex> [first, second, third] = list
[1, 2, 3]
```

These pattern matches should give you a pretty good idea of what's happening. They use the [|] list construction operator, but in reverse.

Everything on the left of the | matches the head of the list exactly. Let's try some trickier pattern matches.

Advanced Patterns

We're not limited to one element. We can try to match as many elements in the head as we want, like this:

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> [first, second|rest] = list
[1, 2, 3]
iex> first
1
iex> second
2
iex> rest
[3]
```

```
iex> [_, second, third|rest] = list
[1, 2, 3]
iex> second
2
iex> third
3
iex> rest
[]
```

Matching more than one element in the head is OK and works similarly to matching the first elements of a tuple. You can ignore elements, and if you use the |, Elixir will match the remaining items as a list.

Can you guess what this code does? Try not to peek past the following listing:

```
iex(10)> [1|[_|rest]] = list
[1, 2, 3]
```

Try not to peek!

```
iex(11)> rest
[3]
```

Did you get it? Here's what happened. The left-hand side is the head of our list, a 1. That leaves the right-hand side to match the rest of our list, [2, 3].

We match the rest of the list with this code: [_|rest]. The underscore ignores the head of the list, a 2. The rest of the list is [3].

The | is optional, but if you omit it, Elixir will treat the list like a tuple. It won't match lists where the number of elements on the left and right sides are different:

```
iex(15)> [first, second] = list
** (MatchError) no match of right hand side value: [1, 2, 3]
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    (iex) lib/iex/evaluator.ex:257: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:237: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:215: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:103: IEx.Evaluator.loop/1
    (iex) lib/iex/evaluator.ex:27: IEx.Evaluator.init/4
```

As expected, Elixir complains. The number of elements on the left and right don't match. The same holds true for too many elements on the left-hand side.

If you're new to functional programming, you might wonder how to take advantage of this kind of pattern matching. Let's find out.

Recursion over Lists

Elixir is immutable, meaning our algorithms won't iterate the way you might see them do in a procedural language like C or an object-oriented language like Python. Instead of iteration, functional languages use pattern matching and functions.

The Simplest Recursive Algorithms

Recursion picks off the head from a list, deals with it directly, and then makes a recursive call to deal with the rest. Here's an example of using a list with recursion over the tail.

```
def total([]), do: 0
def total([head|tail]), do: head + total(tail)
```

If you've not seen code like this before, take a little time to understand what's going on. The first clause is easy to understand. The total of an empty list is zero. The second feels a bit like we're picking ourselves up by our own shoe laces.

Let's dig a little deeper. Say we're calling this function with a list of [1, 2, 3]. Here's what the execution would look like step by step.

```
total([1, 2, 3])
```

We start by invoking total with the whole list. Elixir won't match the first clause, because there's something in the list. So, we take the second clause, with head getting 1 and tail getting [2, 3]. Let's plug that into the return value, like this:

```
1 + total([2, 3])
```

You can start to see what's happening. Now, let's run the function again, but with [2, 3]:

```
1 + 2 total([3])
```

And so on:

```
1 + 2 + 3 + total([])
```

This time we match the final clause:

```
1 + 2 + 3 + 0
```

At least, that's the idea. If you're having trouble with this concept, push it aside for a little while and come back to it. You won't need to write much recursive code in Elixir because many Elixir libraries do that work for you.

There's one more detail we need to cover. It's a concept called tail recursive optimization. Functional languages use tail recursive optimization to convert inefficient recursive calls to efficient loops. The key concept to understand is that if the last function the compiler must execute is a recursive call, Elixir can optimize that code. Let's see how that concept might work.

Tail Recursion with Accumulators

You might have noticed that the previous example started with a large list and made it smaller and smaller with every invocation. The code was easy to understand, but not necessarily efficient. Let's make a *tail recursive* version. Using this strategy, we'll have one function argument holding a list that gets smaller and smaller and another argument that accumulates the answer we've accumulated so far. As you might have guessed, we'll call this second argument the *accumulator*. The program looks like this:

```
def total_with_accumulator([], partial_total) do
  partial_total
end

def total_with_accumulator([first|rest], partial_total) do
  total_with_accumulator(rest, partial_total + first)
end
```

That code might look a little confusing, but it's not too bad if we break it down step by step. Let's call it with our usual list of [1, 2, 3]. We'll start the ball rolling with a partial_total of 0, like this:

```
total_with_accumulator([1, 2, 3], 0)
```

This code won't match the first head, because the list isn't empty yet. Let's take the second clause:

```
total_with_accumulator([2, 3], 0 + 1)
```

And again:

```
total_with_accumulator([3], 1 + 2)
```

And once again, with feeling:

```
total_with_accumulator([], 3 + 3)
```

Which brings us to the last clause, so we return 6!

We can make one tiny improvement. We know the initial accumulator will always be zero, so we don't have to make the user specify it:

```
def total_with_accumulator_api(list) do
  total_with_accumulator(list, 0)
```

```
end
```

Nice. Now the user can simply invoke our total with only a list. We can improve on this code, but to do so, we'll need a concept called higher order functions.

Reduce and Anonymous Functions

Anonymous functions, also called higher order functions, allow developers to treat functions like data. You can pass anonymous functions as arguments, or hold them in variables.

You can then use anonymous functions to automate the code we've written so far. Sometimes, you might want to build your own function to combine elements in a list. Take another look at the critical line of code that uses an accumulator:

```
total_with_accumulator(rest, partial_total + first)
```

Now that you know how that works, let's organize the code with a slight difference:

```
process_list(rest, arbitrary_function)
```

This pattern is common enough that there's a function to do this work called *reduce*. To use it, we need to be able to specify our own functions or use ones we've coded elsewhere.

Let's start with creating our own functions on the fly. That's the role of anonymous functions. They take this form:

```
fn arguments -> body end
```

Notice there's no way to specify a name. Instead of naming them, we'll pass them as arguments to other functions or bind them to variables. In short, *we treat them like data.*

The arguments will use the same form they do in the named functions we've already created. The body of the function is also the same. This syntax is useful when we need to create tiny, shorthand one-shot functions to solve a focused problem.

```
iex(8)> add = fn x, y -> x + y end
#Function<12.128620087/2 in :erl_eval.expr/5>
iex(9)> i
Term
  #Function<12.128620087/2 in :erl_eval.expr/5>
Data type
  Function
Type
```

```
  local
Arity
  2
Description
  This is an anonymous function.
Implemented protocols
  Enumerable, IEx.Info, Inspect

iex(10)> add.(3, 4)
7
```

We create a function that adds `x` and `y` together. Then, we inspect it. The function is as much a data type as any other piece of data in Elixir. The term is a function, the type is `Function`, and the `Info` protocol correctly recognizes it as an anonymous function.

You might have noticed that we need an additional `.` when we invoke it, like this:

```
add.(3, 4)
```

Let's mark that extra period a little better, like this:

```
add<<<.>>>(3, 4)
```

Any time you want to invoke an anonymous function, you need to add the `.` operator and then supply the arguments. This syntax becomes a little awkward when you want to use the function in a pipe, like this:

```
iex(11)> inc = fn x -> add.(x, 1) end
#Function<6.128620087/1 in :erl_eval.expr/5>
iex(12)> inc.(4)
5
iex(13)> 4 |> inc.() |> inc.()
6
```

Normally, when we use a pipe and we don't have to specify any arguments, we don't need any parentheses at all. Since the `.` is not optional, when we pipe with anonymous functions, we need to specify both the dot and the parentheses.

We're not limited to simple single-headed functions.

Multiple Anonymous Function Heads

We have a surprisingly full bag of tricks when we're dealing with anonymous functions in Elixir. We can use pattern matching, multiple function heads, and even guards, like this:

```
iex(17)> fn x when x > 0 -> :positive
```

```
...(17)>   0 -> :zero
...(17)>   x when x < 0 -> :negative
...(17)> end
```

This function specifies multiple heads and guards. It'll work fine. Recursion isn't possible because recursive functions call themselves and anonymous functions don't have names. Still, take heart. You can make your anonymous functions as long, complex, and unreadable as their named counterparts!

Pitfalls of Anonymous Functions

When you're marveling over all of the things you can do as you build your anonymous functions, sometimes it's good to answer the question "Yeah, but should I?" Here are a couple of questions you can ask yourself.

- Would giving my function a name make my code easier to understand?
- Does creating my beautiful anonymous function make the function it's embedded in less readable?
- Can I reorganize my code, perhaps by extracting a bit from an existing function, to obviate the need for the anonymous function altogether?

Anonymous functions are powerful and important. Still, they have their limitations. The main rule when using them is simply to think.

Higher Order Functions

So, now we have another building block, a function we can use as data. We'll call this a higher order function. This concept isn't unique to Elixir. It's the foundation for all functional languages. Anonymous functions are playing an increasingly important role in object-oriented languages as well. Let's see how we might use this idea in the total_with_accumulator we were building earlier.

Elixir's Enum module lets us take a list and a two-argument function to reduce a list to a single value, step by step. When you think about it, that's exactly what our total_with_accumulator function was doing! Here's how it works.

First, create a function with arguments for a list item and an accumulator, like this:

```
iex(18)> add = fn list_item, acc -> list_item + acc end
#Function<12.128620087/2 in :erl_eval.expr/5>
```

Now, you can call reduce, like this:

```
iex> Enum.reduce(list, 0, add)
6
```

Notice we need to specify the initial accumulator, and Elixir does the rest of the work for us. Sometimes, the first element of your list can actually serve as the accumulator, and let Elixir run Enum.reduce/2. Remember, the /2 suffix means the function needs two arguments:

```
iex> Enum.reduce(list, add)
6
```

And it's the same. If this is a bit confusing, this piped version of the same idea may help you understand what's happening:

```
0
|> add.(1)
|> add.(2)
|> add.(3)
```

Marvelous! We're simply using the add function over and over to combine the accumulated total so far with an item from the list. I cheated, but only a little. The pipe expects the accumulator to be *first*, while the Enum.reduce function expects the reducer to be the *second* argument. Still, this little snippet may help you understand exactly what's happening.

Now, we're ready to pull together some of these ideas into another example. We'll implement a polygon using lists.

Implement a Polygon

Roughly stated, a polygon is a shape made up of the area between connected lines. For example, triangles are polygons with three sides. Let's create a module for working with polygons.

We'll start with a few constructors. Here are a couple of constructors for building squares and rectangles:

```
defmodule Polygon do
  def rectangle({x, y}, height, width) do
    [{x, y}, {x+width, y}, {x+width, y+height}, {x, y+height}]
  end

  def square(point, length) do
    rectangle(point, length, length)
  end
end
```

Both of these functions take the building blocks of the shape and return a polygon, a list of points. Now, let's build some reducers. We've already done the bulk of the work in the Point module. The functions in Point work with a single point. All we have to do is call those functions for every point in a polygon, like this:

```
def mirror(polygon, w), do: Enum.map(polygon, &Point.mirror(&1, w))
def flip(polygon, h), do: Enum.map(polygon, &Point.flip(&1, h))
def transpose(polygon), do: Enum.map(polygon, &Point.transpose/1)

def rotate(polygon, degrees, w, h) do
  Enum.map(polygon, &Point.rotate(&1, degrees, w, h))
end

def move(polygon, vector) do
  Enum.map(
    polygon,
    fn point ->
      Point.move(point, vector)
    end
  )
end
```

Each `polygon` is a list of points. To move, flip, or rotate a polygon, we simply call the same function on all of the points in our `polygon`, passing the height or width through as needed. The notation `&function(&1, arg)` builds a new function that looks like this:

```
second_argument = ...some_value...
fn first_argument -> function(first_argument, second_argument) end
```

`Enum.map` combines the concepts of enumerables and higher order functions in a different way from `reduce`. `Enum.map` takes an enumerable and makes a new one by applying a function to each of the arguments in the first one. We're effectively building a mathematical mapping between elements, using a function.

Let's take it for a spin.

```
iex> r = Polygon.rectangle(Point.origin, 4, 2)
[{0, 0}, {2, 0}, {2, 4}, {0, 4}]
iex> r |> Polygon.rotate(270, 2, 4)
[{4, 0}, {4, 2}, {0, 2}, {0, 0}]
```

It works! We create a new polygon and then move it around a bit. We've covered plenty of ground in this example. It's time for you to explore concepts on your own.

Your Turn

This entire chapter was focused on lists, the algorithms we can build with them, the libraries that work on them, and a quick program.

Let's review.

Lists Hold Sequential Variable-Length Data

Lists represent sequences of variable lengths, and they usually have elements of the same type. Pattern matching in lists breaks the list into the head and the rest of the elements.

Under the hood, lists are linked lists, and that means each list is made up of its links, sometimes called cons cells. Like links in a chain, these cons cells make list access at the front of the list relatively fast, but they also make access at the end of the list much slower.

Because Elixir is a functional language, our programs don't iterate over lists. Instead, they use a combination of functions and pattern patching to traverse lists.

We used lists to build out a polygon, a quick program that used our underlying points in a list.

Try It Yourself

We finally have enough tools in the box to solve a good number of the Exercism problems, so we're going to focus on that area for a bit.

These *easy* problems work with lists and enumerables.

- `beer-song`: List the verses of a famous beer song. It'll use enumerables and the concept of a range.
- `rna-transcription`: Work with lists, characters, charlists, and strings to transcribe RNA codes.
- `twelve-days`: Print out the days to the twelve days of Christmas.

These *medium* problems work with lists and recursion.

- `list-ops`: Use recursive algorithms to build list operations like `length` and the like.
- `nucleotide-count`: Work with lists, strings, and characters to count nucleotides within a string.
- `strain`: Use primitive functions to build a filter.

This is a good set of problems for lists and strings. We'll fold in more complex problems as we go on.

Next Time

In the next chapter, we're going to begin to address more advanced programming concepts. We'll also spend a good amount of time with maps and structs. These data structures will give us plenty of firepower to work on more complex data structures. Then, we'll move on to more advanced programming techniques like streams and protocols.

CHAPTER 5

Key-Value Data

The previous two chapters were about linear data. This chapter will focus on key-value data. Let's set the stage, right from the beginning.

Key-value data is everywhere in functional programming because key-value data is everywhere in the world. All of these examples represent key-value pairs:

- We look up definitions in a dictionary by a word.
- Programs refer to people informally by name, or formally by a social security number or email address.
- Function or program options have names and values.
- Database rows have columns.
- Data structures often have fields.

These examples all *associate a key with a value*. To put it another way, these associations are *mappings*. In fact, way down at its foundation, functional programming is also about associations because a *function* is also an association.

Since associations are everywhere in functional programming, it shouldn't surprise you that Elixir has several different ways to refer to key-value data.

But it might surprise you that Elixir had to wait on Erlang before adding maps to the language! That's right. Erlang existed more than 20 years before adding maps, and today, Elixir uses maps everywhere.

In this chapter, you'll see the old way that Elixir used to represent key-value data, called the `Keyword` dictionary, or `Keyword` for short. Then, we'll move on to Elixir maps, which are more efficient representations of key-value data. Finally, we'll use structs, which are maps with restrictions.

Along the way, we'll show you how to use a single map, but also how to work with maps en masse to build our software in layers. We'll also look at some strategies for building APIs that use associations, and where each key-value Elixir data type might fit.

Let's get to it.

Keyword Dictionaries

The first way to represent key-value data in Elixir is to use a combination of the data types we've seen so far. To do so, let's think about the data structures we know about and how to use them.

Remember, tuples hold fixed-length data, where the position of the tuple determines how we treat it. Let's represent a single key-value pair with a tuple. We'll create atoms with a number's *name* as the key, and the number itself as the *value*, like this:

```
iex> one = {:one, 1}
{:one, 1}
iex> two = {:two, 2}
{:two, 2}
iex> three = {:three, 3}
{:three, 3}
```

That's a good start. We have a few key-value pairs. That's not going to get us too far, though. We need to group the data together. Let's use the only data structure we've studied so far that can take variable-sized data, a list:

```
iex> numbers = [one, two, three]
[one: 1, two: 2, three: 3]
```

That was unexpected! We gave Elixir a list of tuples, and it presented the result as [one: 1, two: 2, three: 3].

What you're actually seeing is some syntactic sugar for a keyword list. Let's do our usual inspection of the data type:

```
iex> i
Term
  [one: 1, two: 2, three: 3]
Data type
  List
Description
  This is what is referred to as a "keyword list". A keyword list is a list
  of two-element tuples where the first element of each tuple is an atom.
Reference modules
  --> Keyword <--, List
Implemented protocols
```

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

Look at the emphasized Keyword module under Reference modules. Elixir is recognizing that our structure is a keyword list. Keyword lists are lists of two tuples. These represent key-value pairs. All Erlang associations in the past used this format.

We used atoms to represent keys, and that's an important Elixir idiom. Because all of our tuples start with atoms, we have access to some sugar. First, we can represent our keyword lists the same as in the previous listing. Let's give it a try:

```
iex> bigger = [four: 4, five: 5, six: 6]
[four: 4, five: 5, six: 6]
iex> [first|rest] = bigger
[four: 4, five: 5, six: 6]
iex> first
{:four, 4}
```

We represent a list using the sugar and pick off the first item. You can see the first element is simply a two-tuple.

Though we'll usually prefer maps to keywords, we'll still use keywords for representing things like Elixir configuration in places and function options.

Keywords as Options

In fact, keywords are so important as options that you can omit the square brackets if the last argument of a function expects a keyword. Let's say that you have the following code:

```
def optional(required, opt) do
  one  = Keyword.get(opt, :option1) || "default value"
  two =  Keyword.get(opt, :option2) || "default value"
  do_something_with(required, one, two)
end
```

You could call it like this:

```
optional(:required, option1: "first option", option2: "second option")
```

You'll see syntax sprinkled around Elixir that looks like this, including the one-line function syntax.

Since keywords are lists, it's efficient to access keywords near the front of a list, but doing so becomes increasingly expensive as they hold more tuples for keys that aren't at the front of the list.

That's all we need to address with keywords. Let's move on to a workhorse data structure for Elixir, the map.

Maps

Maps are key-value pairs, but they're built for fast random access. That means you can access any key quickly. In big O notation,[1] the operation is `O(Log n)` for an `n`-item map.

Map Syntax, Sugar, and Access

Let's build a map from a list of tuples. In your console, make sure you still have the `Keyword` called `numbers` defined. Then, create a new map with `Map.new`, like this:

```
iex> numbers = [one: 1, two: 2, three: 3]
[one: 1, two: 2, three: 3]
iex> numbers_map = Map.new(numbers)
%{one: 1, three: 3, two: 2}
```

That's straightforward. The syntax of a map is the same as when the keys are atoms. Maps have a leading `%` and curly braces surrounding the keys and values. If keys are first, you can use the shorthand atom syntax.

If you don't have a key in the first position, you need to use the `=>` operator. We'll call it the hash rocket:

```
iex> names = %{1 => :one, 2 => :two}
%{1 => :one, 2 => :two}
```

Like a `Keyword`, a `Map` holds key value pairs. There are a few key differences. The first, which we've already mentioned, is speed, especially for larger collections. The second is uniqueness. Map keys must be unique.

To access an element from a map, you can use one of three common techniques:

```
iex> numbers_map.one
1
iex> names[1]
:one
iex> Map.get(names, 2)
:two
```

1. https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

We accessed a map three ways. First, if the keys are atoms, you can use the dot syntax. Second, you can use the square brackets, sometimes called the access protocol. Finally, you can use a library function.

If a key isn't there, Map.get/2 returns a nil, and Map.get/3 allows you to specify a default value, like this:

```
iex> Map.get(names, 4)
nil
iex> Map.get(names, 4, 0)
0
```

That behavior may make you nervous because it can cause silent failures. You actually have two more options.

Safe Access with Map.fetch

You can use the different versions of Map.fetch to return an error or raise an exception, like this:

```
iex> Map.fetch(names, 4)
:error
iex> Map.fetch!(names, 4)
** (KeyError) key 4 not found in: %{1 => :one, 2 => :two}
    (stdlib) :maps.get(4, %{1 => :one, 2 => :two})
```

In general, it's better to fail as soon and as hard as you can within application code. The worst behavior is code that silently does something wrong.

As you might expect, you can use pattern matching with maps. Here's how it works:

```
iex> numbers_map
%{one: 1, three: 3, two: 2}
iex> %{one: value1, two: value2, three: value3} = numbers_map
%{one: 1, three: 3, two: 2}
iex> value3
3
iex> %{two: two} = numbers_map
%{one: 1, three: 3, two: 2}
iex> two
2
iex> %{} = numbers_map
%{one: 1, three: 3, two: 2}
```

Brilliant. Pattern matching will fail in these conditions:

- If the pattern on the left is a map and the pattern on the right is not, the pattern won't match.

- If a key is in the pattern on the left, but not the expression on the right, the pattern won't match.
- If any of the values on the left don't match the associated keys on the right, the pattern won't match.

Otherwise, the maps will match. These matching techniques are especially useful for pulling values out of maps. There are a few peculiarities, though.

You can use pattern matching to extract the values for a given key, but you can't extract the key for a value. That's true because Elixir's maps are built to look up by keys efficiently, but not the other way around.

You don't have to specify all of the keys in a map. A successful match guarantees only that the keys you specify match. That means:

An empty map matches every map. That's right. If you need to identify an empty map, you need to use some other tool to do so.

Map pattern matches in function heads are especially useful for extracting values:

```
def total(%{quantity: quantity, price: price}), do: quantity * price
```

The function head does most of the work, pulling out the quantity and price from the line item.

Or verifying structure:

```
def map?(%{}), do: true
def map?(_other), do: false
```

Remember, the empty map matches every map, not only empty ones. This way, we can tell whether we're getting a data type we expect.

Or comparing items:

```
def identity?(%{x: value, y: value}), do: true
def identity?(_point), do: false
```

Now that you've seen a few ways to access the data within a map, we can begin to work on some specialized code to transform them.

Let's look at a few ways we can transform maps.

Map Manipulation

Remember, Elixir data structures can't be updated. When we say "change" or "update", we're talking about returning a new changed map. Let's look at a few techniques we can use to transform maps.

We'll look at a convenient way to change one or more keys in a map using an expression called the map update syntax. Then we'll look at a few functions in the Map API that transform maps. These are two ways to change one map.

We'll also do our usual exploration through maps using the IEx console. We'll look at the underlying data type and explore it a bit. Let's get started.

Map Update Syntax

The map update syntax works best when maps have a known set of keys. Say we have a map representing a person:

```
iex> person = %{first: "Joe", last: "Armstrong", profession: "Programmer"}
%{first: "Joe", last: "Armstrong", profession: "Programmer"}
```

Let's say we wanted to change the profession to author. We could do so like this:

```
iex> %{person|profession: "Author"}
%{first: "Joe", last: "Armstrong", profession: "Author"}
```

Nice. We can change one or more keys, but introducing a new key will throw an error. The map update syntax is good for building reducers in modules that work on maps (or their sister data types structs that we'll explore next.) The variable to the left of the | must contain a map, and the expression will return a new map with changes to only the keys we specify.

Let's look at some functions for working with maps.

Map Transformation Functions

Let's say we had a map with some page counts. Let's start with an empty map and then use functions to add some counts to it, like this:

```
iex> counts = %{} |> Map.put(:index, 1) |> Map.put(:elixir, 1)
%{elixir: 1, index: 1}
```

We use the Map.put function to add a couple of items to our map. Now, let's use an anonymous function to update a key, like this:

```
iex> inc = fn x -> x + 1 end
#Function<6.128620087/1 in :erl_eval.expr/5>
iex> counts = Map.update(counts, :elixir, 0, inc)
%{elixir: 2, index: 1}
iex> counts = Map.update(counts, :erlang, 0, inc)
%{elixir: 2, erlang: 0, index: 1}
```

Great. If the key is found, Elixir applies the function in the fourth argument. If not, Elixir uses the default value in the third argument.

We can use various functions to get the keys or values, delete keys, or drop specific keys. We'll let you look them up in IEx with h Map and exports Map, or look them up in Elixir's excellent online documentation.[2]

So far, the techniques we've talked about were good for working with a single map. Sometimes, we need to work with all of the values in a map, or at least a healthy number of them. To do so, it's best to use the Enumerable protocol.

Protocols and Enumerable

Working with a map *as a whole* instead of piecemeal is different. It's going to take a new set of tools beyond the ones we've seen so far. Let's look at the Map in the IEx console and search for clues, like this:

```
iex> %{}
%{}
iex> i
Term
  %{}
Data type
  Map
Reference modules
  Map
Implemented protocols
  Collectable, Enumerable, IEx.Info, Inspect
```

Interesting. We've used the IEx.Info protocol, and if you look at the bottom of that listing, you can see that Elixir uses the Collectable and Enumerable protocols as well!

A *protocol* provides a contract. To implement a protocol, a programmer must implement a fixed set of functions. In exchange, the protocol provides a set of services. For example, by implementing Enumerable, the creator of the Map module ensured all maps can work with the functions in the Enum module.

Said another way, if you want your module to work with Enum, you have to implement the Enumerable protocol. That means your module needs to implement the functions reduce/3, count/1, member?/2, and slice/1. Similarly, by implementing into/1, your module can participate in the Collectable protocol, a service for traversing data structures in a linear way.

If you don't understand how all of this works, simply remember these two things: you can use maps with the Enum functions, and you can use maps with features that work with into. If you're confused, sit tight. We'll give you plenty of examples in a bit.

2. https://hexdocs.pm/elixir/Map.html

Using Maps in Bulk

Let's build a program to create all of the Roman numeral values from 1 to 1000. We could do so by adding a number to a map, one at a time, but that's not a functional design.

There's a better way, though. When you can, it's better to think more broadly. When you're building a big, predictable map, don't build it one piece at a time. Build all of the tuples at once, and then dump them all at once into a new map.

Let's take an example. Create a new mix project called Romans, like this:

```
[elixir] ➔ mix new roman
...
[elixir] ➔ cd roman
[roman] ➔
```

Now, let's think a bit. We'll want to build a function to calculate the first Roman digit from the left, given a decimal number. The code will look like this:

```
def digit(number) do
  cond do
    number >= 1000 ->
      "M"
    number >= 900 ->
      "CM"
    number >= 500 ->
      "D"
    number >= 400 ->
      "CD"

    ...and so on...
  end
end
```

That program will get us the next digit. Still, it would be better to build a function that we could use either with *reduce* or with *recursion*. Reduce works with lists of things, but we don't actually have a list. Let's make it work with recursion instead.

Let's make our program work with a tuple that looks like {decimal, roman}. With each pass, our function will subtract from the decimal value and add a digit to roman. The function will append to our list head first, for efficiency.

Open up roman.ex, and key this in:

```
defmodule Roman do
  def next_digit({decimal, romans}) do
```

```
    result =
      cond do
        decimal >= 1000 ->
          {decimal - 1000, ["m"|romans]}
        decimal >= 900 ->
          {decimal - 900, ["cm"|romans]}
        decimal >= 500 ->
          {decimal - 500, ["d"|romans]}
        decimal >= 400 ->
          {decimal - 400, ["cd"|romans]}
        decimal >= 100 ->
          {decimal - 100, ["c"|romans]}
        decimal >= 90 ->
          {decimal - 90, ["xc"|romans]}
        decimal >= 50 ->
          {decimal - 50, ["l"|romans]}
        decimal >= 40 ->
          {decimal - 40, ["xl"|romans]}
        decimal >= 10 ->
          {decimal - 10, ["x"|romans]}
        decimal >= 9 ->
          {decimal - 9, ["ix"|romans]}
        decimal >= 5 ->
          {decimal - 5, ["v"|romans]}
        decimal >= 4 ->
          {decimal - 4, ["iv"|romans]}
        decimal >= 1 ->
          {decimal - 1, ["i"|romans]}
      end
    next_digit(result)
  end
end
```

We take a tuple with the digits we've accumulated so far, and a number. Then we match the largest decimal value we can and return a tuple with the new values. Notice we're building our list head first for efficiency. When we're done, we make a recursive call.

This function is long and tedious. There are ways to tighten up this code, but in the interest of making the program easy to understand for our beginners, we'll leave those improvements to our advanced readers. For now, let's give the function a try in IEx:

```
iex> Roman.next_digit({10, []})
{0, ["X"]}
iex> Roman.next_digit({8, []})
{0, ["I", "I", "I", "V"]}
iex> Roman.next_digit({158, []})
{0, ["I", "I", "I", "V", "L", "C"]}
```

Excellent! We're well on our way. Our code does exactly what we need. Now, we need a convenience function to convert those values to a friendly format. Add this function to your program:

```
def convert(decimal) when is_integer(decimal) and decimal > 0 do
  {0, romans} = next_digit({decimal, []})

  romans
  |> Enum.reverse
  |> Enum.join("")
  |> String.to_atom
end
```

We take a decimal number, build the digits, and then convert the digits in romans to a friendly format. We reverse the list and then join them together. For good measure, we convert the string to an atom, which will make it easier to access the Roman numerals in our map, as you'll see in a bit.

This pattern of building lists head first and then reversing the result in the end is a common one in Elixir and other functional languages that rely on linked lists.

Try it out:

```
iex> recompile
:ok
iex> Roman.convert 3
:iii
iex> Roman.convert 64
:lxiv
```

Perfect! Our conversions are working the way they're supposed to. Now, we can use our Roman.convert function to build out our whole, big map. Add this final function to our roman.ex file:

```
def map() do
  for x <- (1..1000), into: %{} do
    {convert(x), x}
  end
end
```

We're using a new feature called a for comprehension. You can get help on it by typing h for in iex.

Let's focus on the top of the for expression first. The value (1..1000) is called a range, and it's an Enumerable. Reading the expression in English, you might say "For every possible x taken from the range of numbers from 1 to 1000, build a list of tuples where the first element is the result of convert(x) and the second element is x. Dump the result into a map."

We've barely scratched the surface of what for can do. You can read about them later. For now, let's try it out!

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> map = Roman.map
%{
  cclxiv: 264,
  dccv: 705,
  xliv: 44,
  clv: 155,
  cxcii: 192,
  dclxvii: 667,
  ...
%}
iex> map.iii
3
iex> map.clv
155
```

It works! Building larger maps from lists of tuples is a great way to bring the whole Enumerable protocol to bear on problems that require large blocks of associative data.

Now that you've seen maps and put them into practice, it's time to address the last Elixir key-value type, structs!

Structs Are Restricted Maps

A struct is a specific kind of map. Where a map can have any combination of keys, every struct that conforms to a type has a specific list of keys.

Structs are ideal for holding metadata for a type. Let's put structs into practice to build an SVG graphic.

We're going to create an SVG graphic representation that we'll eventually put in a text file that looks like this:

```
<svg viewBox="0 0 200 200" xmlns="http://www.w3.org/2000/svg">
  <polygon points="100,100 150,25 150,75 200,0"
            fill="none" stroke="black" />
</svg>
```

defstruct Defines a Struct

Let's switch back to our graphics project. Open up a file called shape.ex and key this in:

```
defmodule Shape do
```

```
  defstruct([:points, :stroke, :fill])
end
```

We've defined the structure we'll use for the Shape module. The single defstruct line completely changes the nature of our module. Now, the data type associated with our module is a Shape *struct*. Rather than define the word, let's try it out instead.

```
[graphics] ➔ iex -S mix
...
iex> exports Shape
__struct__/0     __struct__/1
iex> Shape.__struct__
%Shape{fill: nil, points: nil, stroke: nil}
```

We got the exports of the Shape module and found one more. By virtue of the defstruct definition, we created a struct, and we can access it with the command Shape.__struct__. That's the __struct__/0 form, but there's also a version of __struct__ with one argument, and we can use a Keyword to pass key-value pairs to it, when we initialize it:

```
iex> shape = Shape.__struct__(fill: :white, stroke: :black,
  points: [Point.new(1, 2), Point.new(2, 2), Point.new(2, 1)])
%Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
```

Notice that we've built a map, but the map has the Shape name right after the % character. That's the signal that our map is in fact a struct.

In fact, maps are structs in every sense. You can access individual keys, or use pattern matching just as you can with other maps:

```
iex> shape.fill
:white
iex> %{points: [first, second, third]} = shape
%Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
iex> first
{1, 2}
```

You can use the struct syntax to create a new struct:

```
iex> %Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
%Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
```

You can also provide default values for some or all of your keys, and require keys to be supported. Tweak your shape.ex file to look like this:

```
defmodule Shape do
  @enforce_keys [:points]
  defstruct([
    :points,
    stroke: :blue,
    fill: :black
  ])
end
```

Now, when you create a struct, you'll get the default values:

```
iex> Shape.__struct__
%Shape{fill: :black, points: nil, stroke: :blue}
```

Notice you're getting the default values, but it's not honoring the @enforce_keys module attribute. For this reason, when you create a struct, use the struct syntax.

So, let's look at some of the differences between maps and structs.

Structs vs. Maps

Before we shut this section down, let's do a bit more exploration. The thing that distinguishes a struct from a map internally is one specific key:

```
iex(17)> Map.keys(shape)
[:__struct__, :fill, :points, :stroke]
iex(18)> shape.__struct__
Shape
```

If the _struct_ key is set to a module, the data structure will act like a struct. You can convert from structs to maps by dropping the key, or by calling the convenience method Map.from_struct, like this:

```
iex> map = Map.from_struct(shape)
%{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
iex> map = Map.drop(shape, [:__struct__])
%{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
```

You can also convert from a map to a struct by adding that key, like this:

```
iex> Map.put(map, :__struct__, Shape)
%Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
```

Bingo. Let's talk about the biggest difference between structs and maps, the ability to access keys.

You can use the . syntax or the Map functions to access the data in a struct:

```
iex> Map.get(shape, :fill)
:white
iex> shape.fill
:white
```

But you can't use the access protocol, the one that uses the map[:key] syntax:

```
iex> shape[:fill]
```

```
** (UndefinedFunctionError) function Shape.fetch/2 is undefined
(Shape does not implement the Access behaviour)
    (graphics 0.1.0) Shape.fetch(
      %Shape{fill: :white,
        points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}, :fill)
    (elixir 1.10.1) lib/access.ex:286: Access.get/3
```

You should prefer the map update syntax to update maps as it won't allow you to access a key that's not there:

```
iex> %{shape|backgrond: :plaid}
** (KeyError) key :backgrond not found in:
%Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black}
    (stdlib 3.7) :maps.update(:backgrond, :plaid,
    %Shape{fill: :white, points: [{1, 2}, {2, 2}, {2, 1}], stroke: :black})
    (stdlib 3.7) erl_eval.erl:259: anonymous fn/2 in :erl_eval.expr/5
    (stdlib 3.7) lists.erl:1263: :lists.foldl/3
```

Don't use Map.put to alter a struct, because it doesn't honor the struct restrictions:

```
iex> Map.put(shape, :background, :polka_dot)
%{
  __struct__: Shape,
  background: :polka_dot,
  fill: :white,
  points: [{1, 2}, {2, 2}, {2, 1}],
  stroke: :black
}
```

That's a good whirlwind tour through the primary differences. Before we move on, let's address one more thing. You might be wondering where you should use structs and maps in your greater architecture. In the next section, we'll provide a little guidance.

Structs, Maps, and Public APIs

When you're building software that's bigger than a side project, two language features often come into conflict: type safety and API design. When you're building a structure with type safety, the idea is to give the compiler all of the data that you can so that the compiler can catch mistakes.

So, let's think about the API for our SVG library. So far, the main output for our Shape will be an SVG string. That's easy.

Add this function to your shape.ex file:

```
def to_svg(shape) do
  """
  <polygon
```

```
    points="#{render_points(shape)}"
    style="#{render_style(shape)}"
  />
  """
end
```

That's a simple function. The main syntactic trick is a feature called the heredoc, which uses """ to surround multiline strings, and the #{} delimiters to interpret Elixir code within a string.

Now we come to a crossroads. We could blindly rush through the code for the render_points/1 function, but the devil is in the details. A naive implementation might fail silently, or fail with poor error messages. We need to make sure the points and colors are valid. Let's make sure to catch any errors and reraise them with convenient error messages. These patterns will give us a chance to explore the error handling in Elixir and show off some common API patterns. Add this function to your shape.ex module:

```
def render_points(shape) do
  shape.points
  |> Enum.map(fn {x, y} -> "#{x},#{y}" end)
  |> Enum.join(" ")
rescue
  _exception ->
    reraise "Invalid points: #{inspect shape.points}", __STACKTRACE__
end
```

We render the points using some techniques you've seen before. We use Enum.map to call our custom function to put our points into the correct format for our SVG code. Then, we join it all together.

If anything fails, we want to give the user a clear error message, so we raise a clear error message. Rather than using raise, we use reraise to preserve the stack trace.

Let's finish things out with one more function to render the style which will handle fills for us. Add this code to shape.ex:

```
def render_style(shape) do
  fill = "fill:#{Color.hex_code(shape.fill)}"
  stroke = "stroke:#{Color.hex_code(shape.stroke)};stroke-width:1"
  "#{fill};#{stroke}"
end
```

Notice that these functions go into the Shape module, so we try our hardest to make the first argument a Shape, rather than a color name. It's a small thing, but doing so will keep your Elixir code clear and easy to use.

Let's try things out, first with a bad list of points. Run iex -S mix within the graphics project and try out your code:

```
iex(12)> IO.puts Shape.to_svg(%Shape{points: "1, 2"})
** (RuntimeError) Invalid points: "1, 2"
    (elixir 1.10.1) lib/enum.ex:1: Enumerable.impl_for!/1
    (elixir 1.10.1) lib/enum.ex:141: Enumerable.reduce/3
    (elixir 1.10.1) lib/enum.ex:3383: Enum.map/2
    (graphics 0.1.0) lib/Shape.ex:22: Shape.render_points/1
    (graphics 0.1.0) lib/Shape.ex:13: Shape.to_svg/1
```

We get a full stack trace, but one with a descriptive error message. That's ideal for debugging.

Now, let's try out the function with a valid list of points:

```
iex> shape = %Shape{points: [{10, 20}, {10, 30}, {20, 20}]}
%Shape{fill: :black, points: [{10, 20}, {10, 30}, {20, 20}], stroke: :blue}
```

We get a valid Shape, but if you call the Shape.to_svg(shape) function, you'll find our Color.hex_code/1 function doesn't support the color :blue! Let's fix that in color.ex, like this:

```
def hex_code(:blue) do
  "#0000FF"
end
```

Now, you can successfully print out some SVG code:

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> IO.puts Shape.to_svg(shape)
<polygon
  points="10,20 10,30 20,20"
  style="fill:#000000;stroke:#0000FF;stroke-width:1"
/>

:ok
```

And we're happy. We're returning a string. The one problem is that to use the API, other developers must use the %Shape{} constructor. Actually, that's OK, as long as we provide sensible defaults for all new fields and don't add any new required fields. That would let outside users take advantage of our code without needing to know about any new arguments.

On the other hand, if you have multiple systems that use the same types, adding any required field will force an incompatible change. For example, if we were to push our Shape module broadly and add a required background field, *every system that used the struct would have to change.*

For this reason, it's recommended to use structs for internal APIs where you can for additional compiler benefits. But you should prefer maps around the perimeter. That means maps are best for public API boundaries, process boundaries, and remote service layers.

Further, your APIs should encourage deprecating keys instead of removing them. Additionally, any new key should be *optional*, and the types associated with keys shouldn't change.

This has been a pretty full chapter, and it's time to wrap up.

Your Turn

This chapter was extremely long, and there's a reason for that. Key-value data is important to all functional programming. The fact that Elixir added maps after much of the language was designed complicates things some.

Let's review.

Elixir Places Heavy Emphasis on Key-Value Data

Elixir supports three main kinds of key-value stores. The Keyword type is simply a list of tuples. It has the same performance characteristics as lists, so it's not ideal for handling large key-value datasets where you may need to access any individual element. The Map type is designed from the ground up as an efficient key-value data structure with random access. The Struct is a type that restricts maps to a known set of keys.

We built a couple of programs to illustrate these concepts. The roman project worked with data in bulk, building a complex map with a list of tuples instead of inserting key-value sets one at a time. We also extended the graphics project to illustrate the use of structs and added some lessons about public APIs.

We've given you a pretty good start with associative data. It's time to put it all to use.

Try It Yourself

In this section, we're going to push full steam ahead into the Exercism problem set. Many of them rely heavily on maps. To install an exercise, you'll first install Exercism, and then join the Elixir track. Then, you'll download the problem. For example, to download a problem called hello-world, you'd do this:

```
exercism download --exercise=hello --track=elixir
```

These *easy* problems are a bit harder than the ones you've seen so far. Many will require you to mix and match maps, lists, and tuples.

- `word-count`: Without using the built-in functions, implement a word counter that outputs a map.
- `roman-numerals`: We solved the problem in this chapter. Use this problem as an opportunity to solve the problem in the other direction: convert a Roman numeral to a decimal number.

These *medium* problems work with lists and recursion.

- `rotational-cipher`: Use Elixir to build a cipher to decode messages using this iconic weak cipher.
- `robot-simulator`: This problem is great for thinking through reduce.
- `markdown`: A refactoring exercise. Take code that works, and turn it into better code that works.
- `minesweeper`: Add the numbers to a minesweeper game board.

These *hard* problems are good for more experienced developers:

- `zipper`: Zippers are pure functional algorithms for dealing with trees.
- `bowling`: Score a bowling game. This problem has enough special cases to make it a great problem for exploring data structures.

This is a good set of problems for lists and strings. We'll fold in more complex problems as we go on.

Next Time

In the next chapter, we'll continue to get more advanced. We'll address the more advanced programming features in Elixir, like concurrency and processes. Let's get started!

CHAPTER 6

Processes and Concurrency

Though you might not know it yet, every Elixir program runs in a process. So far, we've not had to write any code to start our own process. In this chapter, we'll change all of that. We'll look at the functions and primitives for spawning processes, sending messages, and building connections between them.

Let's plan our attack. First, we're going to walk through what Elixir processes are and how you can work with them. We'll explore them first in IEx.

Next, we'll use processes to build a key-value store using primitives. We'll talk about the pros and cons of our approach and also concepts like message passing and back pressure.

Finally, we'll give you the chance to try some of these concepts yourself. The chapter will be fairly intense.

Let's get started.

Processes, Inboxes, and Pattern Matching

As you find in other languages, *processes* execute programs. Elixir processes are completely isolated from one another. Processes run concurrently to one another using *preemptive multitasking*. That means that Elixir is responsible for dividing up work, and processes don't share memory with one another. All communication between processes uses message passing.

Elixir processes aren't the same as operating system processes. They're extremely lightweight in terms of resources, including memory. Elixir *processes* are even more lightweight than operating system *threads*. It's common to have hundreds of thousands of them running concurrently.

The techniques and data structures you've seen so far might make some sense to you. When they use message passing and pattern matching together with tuples, many developers find that Elixir begins to resonate in earnest because working with processes in Elixir is often much easier than in other languages.

Processes and message passing are the primitives that serve to build the foundations of Elixir's OTP system, the library that Elixir uses to abstract concurrency, application lifecycles, message passing, supervision, and failover. Because it's such a central concept in other functional languages like Erlang and Scala, Programmer Passport will focus an entire book on OTP. This chapter will provide many of the primitives Elixir builds on for OTP.

Let's start to play with processes.

Processes Have Pids and Message Queues

Regardless of whether an Elixir program is running in a test script, IEx, or mix task, it's running in a process. Let's open up IEx and start to play. Let's find the IEx process ID, like this:

```
iex> iex = self()
iex = #PID<0.103.0>
```

All processes are identified by a process ID, called a *pid*. Let's get the info for our pid:

```
iex> i
Term
  #PID<0.103.0>
Data type
  PID
Alive
  true
Name
  not registered
Links
  none
Message queue length
  0
Description
  Use Process.info/1 to get more info about this process
Reference modules
  Process, Node
Implemented protocols
  IEx.Info, Inspect
```

This list is interesting. In addition to the usual information, you can see a few extra bits of information. For example, the Alive section tells us our process is in fact alive.

The modules for working with a process are Process for processes on one system and Node for working with *distributed systems*. A distributed system divides a single system into slices, potentially across different networked computers.

Notice also the message queue length. Think of the message queue as a mailbox. We can send messages to our process, like this:

```
iex> send iex, :hello
:hello
iex> send iex, {:is, :anyone, :home}
{:is, :anyone, :home}
iex> i iex
Term
  #PID<0.103.0>
...
Message queue length
  2
...
```

Nice! We've sent a message to ourselves, so there are two messages in our own mailbox. Let's use the API to get information about this process:

```
iex> Process.info iex
[
  current_function: {Process, :info, 1},
  status: :running,
  message_queue_len: 2,
  ...
]
```

I've shortened this list, but you get the idea. Our process is running, and it has a message queue length of 2. In IEx, you can issue the command flush to empty the queue and print all of the messages, like this:

```
iex(14)> flush
:hello
{:is, :anyone, :home}
```

And we see the two messages we sent previously!

Typically, you don't receive messages with flush/0. That function is in the IEx.Helpers module. Let's look at a more realistic example.

Get Matching Messages with receive

The receive control structure lets us receive a message matching a pattern. Let's build a function to send a message to IEx. A common pattern in Elixir is to shape messages into tuples. These messages often have some type of atom to match against, some kind of numeric code, and a string with an English message so we can easily tell what the message means. Let's build a function called deliver to send such a message:

```
iex> deliver = fn i -> send self(), {:message, i, "Message#{i}"} end
#Function<6.128620087/1 in :erl_eval.expr/5>
iex> Enum.each((1..6), deliver)
:ok
iex> Process.info(self())[:message_queue_len]
6
```

We build the deliver function to send a message in a three-tuple based on some integer to self(). The message is a three-tuple. Then, we deliver messages to each of the six integers in the range from 1 to 6. To make sure the messages got sent, we check the :message_queue_len using Process.info/1.

We already know how to flush all of the messages. Let's receive a message:

```
iex> receive do
...>   message ->
...>     IO.inspect(message)
...> end
{:message, 1, "Message1"}
{:message, 1, "Message1"}
```

We ask Elixir to receive a message and give it the message pattern to match. Elixir will return the first message in the queue that matches the pattern. We match the first message in the queue, print it out, and return it.

Let's look at receive in more detail. The expression we'll use is the receive message. It takes the form:

```
receive do
  pattern ->
    expression
  after
    timeout_in_milliseconds ->
      timeout_expression
end
```

Let's try another one. Let's match an explicit message, like this:

```
iex> receive do
...>   {:message, 4, text} ->
...>     text
```

```
...> end
"Message4"
iex> Process.info(self())[:message_queue_len]
4
```

We ask Elixir to match the message that matches the tuple {:message, 4, text}. Elixir skips the next two messages until it reaches the tuple {:message, 4, "Message4"} and binds the text variable to Message4. Four messages remain: 2, 3, 5, and 6. We've not received them yet.

Let's see how the timeout works.

```
iex> receive do
...>   {:message, 4, text} ->
...>     text
...> after
...>   1000 ->
...>     {:error, :no_message}
...> end
{:error, :no_message}
```

We ask for message 4. Finding none, Elixir waits one thousand milliseconds, or one second, and returns the value we provide. Using receive, we can match any pattern we choose. If there's no message yet in the message queue, Elixir will wait until there is a message. If you specify an optional after block, Elixir will raise a timeout error if no message is received.

receive is the way all processes interact with each other.

There's still an important tool you haven't seen yet. The next step in working with processes is starting them.

Start a Process with spawn

The easiest way to start a process is to provide a function. There are several versions of spawn. We'll use the one that takes a function with no arguments. Let's start a process that uses receive to wait for a message, and then print it out, like this:

```
iex> catcher =
...>   fn ->
...>     receive do
...>       m ->
...>         IO.inspect(m)
...>     end
...>   end
#Function<20.128620087/0 in :erl_eval.expr/5>
iex> pid = spawn(catcher)
```

Now, we have a process running a function. The process is called receive, so it's waiting on a message:

```
iex(42)> Process.alive?(pid)
true
```

Let's send a message:

```
iex> send pid, :boink
:boink
:boink
iex> Process.alive? pid
false
```

Don't worry. We didn't actually kill the catcher. The catcher simply received the message and then finished processing it. There's no more work to do, so it exited.

We can use these tools to build programs. Let's put them into practice. Let's build a key-value store.

Put It All Together in a Message Loop

A common way to use Elixir processes is within a message loop. Let's build a quick key-value store, a process that can be shared across processes.

Our general strategy is to divide our application into three parts: the Core, the Server, and the API. Let's talk about each piece in turn.

The Core will have the simple functions we need to process individual requests. These functions will be reducers. They'll take a key-value store and return a transformed store.

The Boundary will wrap the Core layer. It'll manage the process machinery that starts processes, receives messages, and returns responses. We can't completely remove the complexity in the Boundary. The best we can do is to hide the complexity of the service features, the ones we place in the Core, so that the programmer can deal with one bit of complexity at a time.

The API layer will wrap the Server layer. That way, programmers can make simple function calls rather than sending messages between processes.

Let's get started.

Create the Project

Our project will take the shape of all other Elixir projects. We'll build it using mix new, like this:

```
[elixir] mix new key_value_store
...
* creating lib
* creating lib/key_value_store.ex
* creating test
* creating test/test_helper.exs
* creating test/key_value_store_test.exs

...
cd key_value_store
[key_value_store] ➔
```

Now, we have a bite-sized Elixir project that handles one concern, a key-value store. Next, we'll go through the system, layer by layer, writing the code and test-driving it.

Establish a Core

The first step in our plan is to create our business features. Our business layer answers the question "What?" It defines the features our clients will want to see.

We're going to separate the process machinery from our business features. We'll create the business layer in lib/core.ex, like this:

```
defmodule KeyValueStore.Core do
  def new, do: %{}
  def add_or_update(store, key, value), do: Map.put(store, key, value)
  def delete(store, key), do: Map.delete(store, key)
  def retrieve(store, key), do: Map.get(store, key)
end
```

The Core module has all of the functions we have that are unrelated to processes. They're sometimes called pure functions. This core makes it easy to capture the essence of our program. We can create, read, update, and delete values in our store. All of our functions are constructors or reducers. That means we're ready to try it out.

Let's try it out.

```
iex> alias KeyValueStore.Core
KeyValueStore.Core
iex> KeyValueStore.Core.new
%{}
iex> KeyValueStore.Core.new |> Core.add_or_update(:one, 1)
%{one: 1}
iex> %{} |> Core.add_or_update(:one, 1) |> Core.retrieve(:one)
1
```

It works. The pipes let us know that we're on the right track because it means our module is structured in the right way. Of course, we're using Elixir's Map module to do all of the work. We could have easily used `Map` in our modules instead of `Core`, but put that idea aside for the moment. Instead, focus on the way we've isolated a back-end feature, one with its own functions, into a functional core.

Let's look at this problem in yet another way. Our functional core works with *one slice of time*. Our key-value store builds the *next key value store* by adding or deleting one key at a time. A game server would build the *next frame*. A bank account would process a *single transaction*. If you understand this concept, you're one step closer to understanding Elixir.

Everything is `reduce`.

Now, let's build our process layer around all of these reducers.

Wrap the Core in a Boundary

Our next step is to build the boundary. The best way to describe how this layer works is to see one in action. The boundary deals with the process machinery we'll use to maintain state over time.

We'll put our boundary in `server.ex`. We'll describe the program piece by piece. Start with an empty module, like this:

```
defmodule KeyValueStore.Server do
  alias KeyValueStore.Core
end
```

Our module is called a `Server`, not in the sense of a networked server, but in the sense of a process establishing a service. This terminology is common in the Elixir ecosystem based on the language established in Erlang, many years ago.

This server will be the only module that directly accesses our functional core, so we alias it. The only job of the server is to handle boundary concerns. the process machinery in the application.

The first step is to write a function to spawn our process, like this:

```
def start do
  spawn(fn ->
    run(Core.new())
  end)
end
```

We use a spawn function to start a process and pass it the run/1 function. Notice that run/1 takes the initial value of our key-value store. This function is important because it processes our *message loop*. In Elixir, a message loop is a recursive function that receives messages.

Let's write run now. Add it beneath the start function, like this:

```elixir
def run(store) do
  store
  |> listen
  |> run
end
```

That tiny function packs a punch. It starts with the state, pipes it first into listen to process a message, and finally pipes recursively back into run itself to complete the message loop.

The last step is to write a tiny function whose only job is to listen for a single response, like this:

```elixir
def listen(store) do
  receive do
    {:put, key, value} ->
      Core.add_or_update(store, key, value)
    {:delete, key} ->
      Core.delete(store, key)
    {:get, pid, key} ->
      value = Core.retrieve(store, key)
      send(pid, {:value, value})
      store
    {:state, pid} ->
      send(pid, {:store, store})
      store
  end
end
```

This function is a *reducer* with a *side effect*. It's a reducer because the function takes a key-value store, and each of our messages returns a key-value store. Our function also has a side effect. In functional programming, a side effect is something that changes the state of the environment. In our case, the side effect is to receive a message.

Two of our messages simply modify our store. The other two don't modify the core but send some information back to the caller.

Let's give it a try. Recompile your iex session, alias the server, and start the server, saving a pid, like this:

```
iex> alias KeyValueStore.Server
```

```
KeyValueStore.Server
iex> pid = Server.start
#PID<0.157.0>
```

We can get information about the process:

```
iex(9)> Process.alive?(pid)
true
iex(10)> Process.info(pid)
[
  current_function: {KeyValueStore.Server, :listen, 1},
  initial_call: {:erlang, :apply, 2},
  status: :waiting,
  message_queue_len: 0,
  links: [],
  dictionary: [],
  priority: :normal,
  total_heap_size: 233,
  heap_size: 233,
  stack_size: 4,
  reductions: 29,
  ...
]
```

I've trimmed some of the information out of this list, but there's plenty of useful information remaining. We know that the process is waiting in the function KeyValueStore.Server.listen/1, that it's in the :waiting state, and that the message queue length is zero.

The message queue length is an important attribute when you're working with processes in production because a backed-up process will have many unprocessed messages in the message queue. We can also see information about the memory heap and stack sizes, and the amount of work done, measured in *reductions*. When you see reductions, think Enum.reduce. We could also see other linked processes, and important monitoring relationships you'll see in the OTP release.

Now, let's interact with the server. Our Server is perfectly functional, if a bit awkward:

```
iex> send pid, {:put, :one, 1}
{:put, :one, 1}
iex> send pid, {:put, :two, 2}
{:put, :two, 2}
iex> send pid, {:state, self()}
{:state, #PID<0.137.0>}
iex> flush
{:store, %{one: 1, two: 2}}
:ok
```

This server works, but it's a bit awkward. We can do better. Let's focus next on cleaning up the API.

Establish an API Layer

We're going to put the code in the existing lib/key_value_store.ex file. Let's start with a simple module that starts the server:

```
defmodule KeyValueStore do
  alias KeyValueStore.Server

  def start() do
    Server.start()
  end
end
```

All of our access goes through the Server module. None of our code uses the Core layer. This design is intentional. We want to work with a single layer at a time, if possible.

Next, let's implement the functions that modify the store:

```
def put(server, key, value) do
  send(server, {:put, key, value})
  :ok
end

def delete(server, key) do
  send(server, {:delete, key})
  :ok
end
```

These two functions send messages to the server process to transform the store and return an :ok tuple. Users will be able to call the put and delete functions rather than send messages. That's a simpler API that's less error prone.

Now, let's work on the complex cases, the ones that must receive responses:

```
def get(server, key) do
  send server, {:get, self(), key}
  receive do
    {:value, value} -> value
  end
end

def state(server) do
  send server, {:state, self()}
  receive do
    {:store, store} -> store
  end
end
```

These functions are a little more sophisticated, and we're happy to process the receive in these functions so our clients won't have to do so. For example, the get function sends the :get tuple with our process ID and the key we want to retrieve. Next, we receive the response from the server, matching the message against the {:value, value} tuple we expect to receive. The state function looks more or less the same.

We have all we need to take our API for a test-drive! Let's do that now:

```
iex> recompile
Compiling 1 file (.ex)
:ok
iex> server = KeyValueStore.start
#PID<0.165.0>
iex> KeyValueStore.state server
%{}
iex> KeyValueStore.put server, :one, 1
:ok
iex> KeyValueStore.state server
%{one: 1}
iex> KeyValueStore.get server, :one
1
iex> KeyValueStore.delete server, :one
:ok
iex> KeyValueStore.state server
%{}
```

Everything works perfectly! We no longer have to worry about sending messages. The API layer handles that problem. Especially refreshing are the APIs that require receiving a response. We simply let the API layer do the work.

Before moving on, we should talk about potential problems with our solution.

What's Wrong

Our server is nice, but there are several problems. It won't handle failure well. A failing server will simply stop, causing hangs or crashes in our API code. We also have to write a bit of boilerplate code to build something as trivial as a key-value store.

What's needed is a generic implementation that builds all of these services into Elixir, one step at a time. Fortunately, such a service exists. It's called OTP.

If you want to understand Elixir, you'll eventually need to dive into OTP. The documentation for both Elixir and Erlang are excellent, and several books cover the topic well, including *Programming Elixir 1.6 [Tho18]* by Dave Thomas, *Designing Elixir Systems with OTP [IT19]* by James Edward Gray II and Bruce

A. Tate, and *Functional Web Development with Elixir, OTP, and Phoenix [Hal18]* by Lance Halvorsen.

We've almost exhausted the content for this chapter. It's time to move on.

Your Turn

Elixir exists partially because its creator José Valim wanted abstractions for dealing with concurrency at scale. The processes you see in this chapter begin to tell that story.

Processes

Elixir's lightweight processes are ideally suited for building multiprocess frameworks for dealing with concurrency at scale. Using processes in Elixir feels natural. The processes are even more lightweight than operating system threads. Elixir processes can send messages to one another. Each process has its own message queue, and processes can receive messages that match individual patterns.

Programs use these primitives within message loops. Programs use receive to match messages of a particular type to do a job. Our program matched messages to put, get, and delete keys in our KeyValueStore. In addition, our program had an API to spawn a new key-value store.

Finally, we looked at flaws in programs built out of primitives. While understanding these building blocks is helpful, most Elixir programmers use a prebuilt framework called OTP to build their multiprocess applications.

Now that you've seen message passing in action, you can write your own programs that use these techniques.

Try It Yourself

In this section, you're going to extend existing exercism programs to wrap a message loop around them. Since these projects are more involved, we'll only offer four problems. All are around the *medium* level of difficulty.

- Implement the Exercism bob problem using a message passing API with a message loop.
- Implement the Exercism robot-simulator problem using a message passing API that responds to messages to turn left, right, and move ahead.
- Work the Exercism bank-account problem.
- Work the Exercism parallel-letter-frequency problem.

Next Time

In the next chapter, we'll look at sigils, binary pattern matching, and date/time types. These are typical weak spots for intermediate Elixir developers.

CHAPTER 7

Blind Spots

Many Elixir programmers, regardless of whether they're beginners, intermediates, or experts, struggle with some pretty common issues. This chapter is dedicated to filling in typical blind spots for Elixir programmers.

We'll start this tour with a bit of syntactic sugar called *sigils*. These little gems improve your code by making it easier to express awkward concepts like strings with quotes in them and lists of atoms or strings. Elixir also makes it much easier to express dates and times with sigils.

We'll then shift toward a discussion about time. Dealing with time, dates, calendars, and time zones has knocked many great programmers to the ground. This area is ripe for bugs and misunderstandings.

Finally, we'll shift to booleans, bits, and binary pattern matching. These tips can let you shorten functions and pack a lot of information into a small number of bytes when space is at a premium.

These quick tools will definitely help intermediates smooth out their Elixir. Let's get started.

Sigils

Elixir developers must deal with many different data types with different representations. Sigils exist to make text representations of data types less awkward. Let's look at an example.

Every programmer knows it's hard to read code with strings containing quotes. In Elixir, you'd represent quotes within a string with a preceding backslash, like this:

```
IO.puts "\"string with quotes\""
"string with quotes"
:ok
```

Now, how would you represent the first line in the previous listing as a string? You would need to escape every quote and backslash! Or, you could use a sigil, like this:

```
iex> ~s("string with quotes")
"\"string with quotes\""
```

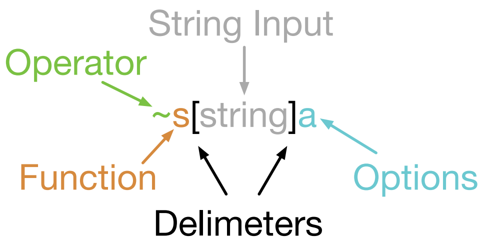
~s is a sigil. The alternative syntax for strings without quotes means we don't have to escape anything.

Even experienced Elixir developers may not know how to get the most out of sigils. That's a shame because they can make your code much more readable. Don't worry. We'll fill in some of the knowledge gaps for you.

We'll start with the pieces that make up a sigil.

The Shape of a Sigil

Think of a sigil as a bit of syntax that has a ~ character, a sigil character or word, a string surrounded by delimiters, and some option characters, like this:



The *operator* is the leading ~ character. That character lets Elixir know the next bit of syntax will be a sigil. The *function* defines how Elixir will convert the string to some data type. The *delimiters* are the two characters that surround the sigil's input. We'll look at the list of supported delimiters in a second. The *options* are one or more characters that define customizable options for each sigil.

For example, look at the following sigil:

```
iex> ~w[one two three]
["one", "two", "three"]
iex> ~w(one two three)a
[:one, :two, :three]
```

Both of these sigils are w sigils, used to build a list of words. The first invocation uses [] characters as delimiters, and specifies no options. The second sigil uses the () delimiters, and the a option denotes a list of atoms instead of strings.

The delimiters you choose can have a tremendous impact on the readability of your code. Elixir sigils allow eight delimiters:

- ~s"string"
- ~s'string'
- ~s/string/
- ~s|string|
- ~s(string)
- ~s[string]
- ~s{string}
- ~s<string>

You can use any delimiter with any kind of sigil. Regular expressions often have the / delimiter, and arrays of words will often use (), but they don't have to. Pick the delimiter that makes your code the prettiest.

Sigils for Strings and Lists

String in Elixir will usually use the "" delimiters, but sigils can make it easier to represent strings with special characters. Two sigils support strings. The s sigil allows interpolation, and the S doesn't:

```
iex> int = 42
42
iex> ~S[The magic number is "#{int}"]
"The magic number is \"\#{int}\""
iex> ~s[The magic number is "#{int}"]
"The magic number is \"42\""
```

The c sigil works with Charlists the same way the s sigil works with strings:

```
iex> ~c[This math quote has p and p' variables]
'This math quote has p and p\' variables'
```

The w builds lists of words. It has options for both strings and atoms, like this:

```
iex> romans = ~w[i v x v c l]
["i", "v", "x", "v", "c", "l"]
iex> roman_atoms = ~w[i v x v c l]a
[:i, :v, :x, :v, :c, :l]
```

The w sigil with the a character is particularly good for building modules that have structs with a bunch of attributes, like this:

```
defmodule Person do
```

```
  defstruct ~w[first middle last prefix suffix id gender profession etc]a
end
```

The representation is more compact than spelling out the syntax:

```
[:first, :middle, :last, :prefix, :suffix, :id, :gender, ...]
```

These s and w sigils help us by simplifying the process of building long lists or escaping strings. Sometimes, remembering all of the details associated with a particular data type makes representations difficult. Such is the case with regular expressions, and they're next.

Regular Expressions

Regular expressions are data types that match complex strings. They're an ugly but necessary part of many modern programming languages. We won't offer you a comprehensive guide for them, but the chances are good that you'll encounter them in Elixir, so we'll offer you a brief overview.

By far the most common way to express a regular expression is with the r sigil with the / delimiters. For example, to build a regular expression that matches one or two, you'd do this:

```
iex> either = ~r/one|two/
~r/one|two/
```

Then, to determine whether a regular expression matches, you'd use the =~ operator, like this:

```
iex> "one" =~ either
true
iex> "three" =~ either
false
```

Notice your regular expression is case sensitive:

```
iex> "ONE" =~ either
false
```

You can add an i option to make a regular expression case insensitive:

```
iex> either = ~r/one|two/i
~r/one|two/i
iex> "ONE" =~ either
true
```

Sometimes, when you're matching slashes, it pays to change up the delimiter:

```
iex> path = ~r|/index/|
~r/\/index\//
iex> "/products/index/1" =~ path
```

```
true
```

The power of the sigil shines through! We've covered two of the most common sigil families. Let's move on to a third, dates and times.

Sigils for Dates and Times

Remembering the functions for creating dates and times can be demanding. In this section, we'll walk through some of the sigils that allow you to represent various common date and time formats. We'll work through the list quickly.

You can use a sigil to represent a Date:

```
iex> d = ~D[2020-04-06]
~D[2020-04-06]
iex> d.day
6
```

Or a time:

```
iex> t = ~T[20:00:00.0]
~T[20:00:00.0]
iex> t.second
0
```

Or a DateTime:

```
iex> dt = ~U[2019-10-31 19:59:03Z]
~U[2019-10-31 19:59:03Z]
iex> %DateTime{minute: minute, time_zone: time_zone} = dt
~U[2019-10-31 19:59:03Z]
iex> minute
59
iex> time_zone
"Etc/UTC"
```

Or a NaiveDateTime, which is a DateTime type without time zone information:

```
iex> ndt = ~N[2019-10-31 23:00:07]
~N[2019-10-31 23:00:07]
```

These date and time formats are typically easier to remember and read. You'll often find that the inspect protocol for these types of data returns a sigil form. We'll work more with these sigils in a little bit.

For now, let's find out how to write our own sigil.

Define Your Own Sigil

Under the hood, a sigil is a function. Its name takes the form sigil_c/2, where c is the character representing your sigil. The function takes two arguments, a string and the options. Let's write a sigil that can represent a number in binary format. Create an Elixir project like this:

```
[elixir]-> mix new sigil
[elixir]-> cd sigil
[sigil]->
```

Key this program into lib/sigil.ex:

```
defmodule Sigil do
  @erlang_32_bit_format_string "~32.2B"
  def sigil_b(string, []) do
    string
    |> to_bits_string
    |> String.to_integer
  end

  def sigil_b(string, [?s]) do
    string
    |> to_bits_string
  end

  defp to_bits_string(string) do
    @erlang_32_bit_format_string
    |> :io_lib.format([String.to_integer string])
    |> List.to_string
    |> String.trim
  end
end
```

We have two function heads for the sigil. The first takes no options, and an inbound string. It pipes the string through the function to_bits_string, which uses an Erlang function and a few Elixir helper functions to represent a number as a string of bits.

The second function head takes the options. Our sigil has one, a charlist. A common way to match single options is to use the ? operator to get the code point for a character. In this case, we match ?s which stands for "string".

Now, we can put our sigil to use:

```
iex> import Sigil
Sigil
iex> ~b/254/
11111110
iex> ~b/254/s
"11111110"
```

It works like a charm! Whenever we use the ~b sigil, Elixir will invoke sigil_b, passing the string between the delimiters as a string and the options after the final delimiter as a charlist.

Now, you know a bit more about using sigils. Let's put some of those date and time sigils to use as we explore dates and times.

Dates, Times, and Comparisons

Date math is like the apple in the garden of Eden. It looks so simple and delicious, and by the time you take a bite, it's too late to change your mind. Date math nearly brought the world to its collective knees when the calendar flipped over from 1999 to 2000 because the industry wrote code that expressed years with two characters instead of four. Those two characters worked fine until close to the turn of the century. Then, billions of lines of code tried to incorrectly compare the years of `1999` to `2000` in millions of codebases. Since `00` is less than `99`, we had a huge problem, one that even had its own acronym: Y2K.

Dates Are Structs

Elixir has plenty of tools for dealing with dates and times, but you need to know about the pitfalls. The main gotchas all relate to date comparison:

```
iex> {:ok, earlier} = Date.new(2000, 12, 1)
{:ok, ~D[2000-12-01]}
iex> {:ok, later} = Date.new(2000, 12, 1)
{:ok, ~D[2000-12-01]}
iex> {:ok, earlier} = Date.new(1999, 11, 30)
{:ok, ~D[1999-11-30]}
iex> earlier < later
false
iex> earlier
~D[1999-11-30]
iex> later
~D[2000-12-01]
```

That was unexpected. What's going on here?

If we were dealing with simple tuples, we'd be OK:

```
iex> {1999, 12, 1} < {2000, 11, 30}
true
```

The sad truth is that we're not. Let's get a few more clues about the data type:

```
iex> i earlier
Term
  ~D[1999-11-30]
Data type
  Date
Description
  This is a struct representing a date. It is commonly
  represented using the `~D` sigil syntax, that is
  defined in the `Kernel.sigil_D/2` macro.
Raw representation
  %Date{calendar: Calendar.ISO, day: 30, month: 11, year: 1999}
  ...
```

IEx.info comes to the rescue again! The raw representation is a struct, and that's the problem. Elixir is trying to compare raw structs but doesn't know how. Help is on the way, but as of this writing, it's not here yet. Now, the best way to compare a Date, Time, or Datetime is with the module's compare function, like this:

```
iex> Date.compare(earlier, later)
:lt
```

This function will return :lt, :gt, or :eq for less-than, greater-than, or equal. The comparison works like a charm. Let's look at some of the other Date and Time features.

Times Are Structs Too

As you might imagine, under the hood, times are structs too. We can do a little exploration in IEx to see what's happening.

```
iex> Time.new(12, 11, 10)
{:ok, ~T[12:11:10]}
iex> ~T[12:11:10]
~T[12:11:10]
```

We call Time.new. Notice that it returns an :ok tuple when successful. That means you'll have to be able to handle an error if your code passes bad time data. Notice the sigil representation that comes back. Let's get a little more information about times with info. Type i:

```
iex> i
Term
iex(31)> i Calendar.ISO
Term
Calendar.ISO
Data type
Atom
Module bytecode
/Users/batate/.asdf/installs/elixir/1.10.1/bin/...
Source
/home/build/elixir/lib/elixir/lib/calendar/iso.ex
Version
[321360612332988397742151427117946492255]
Compile options
[:dialyzer, :no_spawn_compiler_process, :from_core, :no_auto_import]
```

```
Description
Use h(Calendar.ISO) to access its documentation.
Call Calendar.ISO.module_info() to access metadata.
Raw representation
:"Elixir.Calendar.ISO"
Reference modules
Module, Atom
Implemented protocols
IEx.Info, Inspect, List.Chars, String.Char
Data type
  Time
Description
  This is a struct representing a time. It is commonly
  represented using the `~T` sigil syntax, that is
  defined in the `Kernel.sigil_T/2` macro.
Raw representation
  %Time{calendar: Calendar.ISO, hour: 12, microsecond: {0, 0},
        minute: 11, second: 10}
Reference modules
  Time, Calendar, Map
Implemented protocols
  IEx.Info, Inspect, String.Chars
```

Times are also structs under the hood, and comparing them works the same way comparing dates does. Working with an individual time is pretty straightforward. Once you know the underlying structure, you can quickly access the individual pieces of a time, just as you might expect:

```
iex> time = ~T[12:11:10]
~T[12:11:10]
iex> time.hour
12
iex> time.minute
11
iex> time.mi
microsecond    minute
iex> time.microsecond
{0, 0}
iex> time.calendar
Calendar.ISO
```

We create a time with a sigil and then pick off the pieces of the Time struct. Notice that the microsecond is simply a two-tuple, with each element an integer from zero to one thousand.

But what's the calendar field? Let's find out now.

Calendars and Dates

While we're here, let's look at the concept of a Calendar. If you look closely, you can see the ~T[12:11:10] time has a :calendar field with the Calendar.ISO value. Let's see what that means.

A *calendar* is an API. In Elixir, the Calendar module is a *behaviour* that makes it easier to deal with date math. (Yes, that spelling of behaviour is correct. Elixir and Erlang use the British spelling.) You might use a Calendar to find out the day of the week or to find out how much time has passed between two dates. To satisfy this behaviour, a module must implement the functions specified in the behaviour.[1] Calendar.ISO implements the Calendar behaviour. Let's get a little more information about it:

```
iex(31)> i Calendar.ISO

Term
  Calendar.ISO
Data type
  Atom
Module bytecode
  /Users/batate/.asdf/installs/elixir/1.10.1/bin/...
...
Description
  Use h(Calendar.ISO) to access its documentation.
  Call Calendar.ISO.module_info() to access metadata.
```

IEx is telling us this is an Erlang module and suggesting a better way to get information. Typing h Calendar.ISO tells us that Calendar.ISO is the implementation of a particular calendar, one that follows *ISO 8601*, an international standard for representing dates and times.

Providing a full description of dates and times is beyond the scope of this book. Keep in mind that when you're doing date math, this is the place to do it. This is true whether you're converting dates or times or you're adding time to a calendar date.

Let's cover one more concept before moving on to binaries and bit strings.

Time Zones

In the real world, a time zone represents a local representation of time. The rules are surprisingly complex. 12:00 AM in New York City is three hours ahead

1. https://elixir-lang.org/getting-started/typespecs-and-behaviours.html#behaviours

of the same time in San Francisco. Some time zones respect daylight savings time and some don't. The rules can get even more complex.[2]

In Elixir, there are two specific types of date/time representations. A DateTime has built-in information about its time zone. A NaiveDateTime doesn't. In Elixir, the recommendation is that you work strictly with a single time zone, the UTC time zone. If that's not an option for you, work with NaiveDateTime, which has no underlying time zone representation.

We're not going to go into much depth here, but since there are potential problems, we want you to know where to find the right information. Working with dates and times is surprisingly fluid in Elixir, so it's best to stay within Elixir's standard library documentation[3] or the resources maintained by groups[4] rather than trying to rely on older articles. That way you can stay up to date.

That's probably enough about times and time zones. Let's shift to another kind of obscure math, working with bits, bytes, and binaries.

Binaries and Bit Strings

Elixir is a language built on top of Erlang, and Erlang was built to work on phone switches. As you might expect, Erlang is good at working with data down at the level of bits and bytes, and Elixir has inherited many of these capabilities. In this section, we'll quickly cover the tools you can use if you ever find yourself working with low-level protocols like parsing a .jpg[5] image or taking apart a low-level TCPIP[6] header.

Binaries

Let's dive a little deeper. At their lowest level, computers are about the flow of data, and that data is all made up of 1s and 0s. These bits represent a numerical format called *binary*. One binary digit is a *bit*. While decimal systems are made up of digits from 0 to 9, binary numbers are made up of digits from 0 to 1. Check out this binary number system[7] website for more information.

We'll need a few numbers to work with. Here are the first eight numbers in binary:

2. http://www.creativedeletion.com/2015/01/28/falsehoods-programmers-date-time-zones.html
3. https://hexdocs.pm/elixir/DateTime.html
4. https://elixirschool.com/en/lessons/basics/date-time/#working-with-timezones
5. https://web.stanford.edu/class/ee398a/handouts/lectures/08-JPEG.pdf
6. https://tools.ietf.org/html/rfc793
7. https://www.mathsisfun.com/binary-number-system.html

decimal	binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

The most basic form of non-structured data in Elixir is the binary. It consists of bytes, listed sequentially, in the *binary syntax*. The binary syntax uses the <<>> delimiters, like this:

```
iex> binary = <<1>>
<<1>>
iex> byte_size binary
1
iex> bit_size binary
8
```

Internally, a byte is a number eight bits long. Binaries are made of bytes.

Strings are binaries with code points in them. Remember, code points are integers representing characters:

```
iex> <<?c, ?a, ?t>>
"cat"
```

Bits and Pattern Matching

Communications and data storage are problem spaces that often require data placement at the single-bit level. Elixir has a great tool for dealing with bit-level data, the *bit strings*. A bit string represents sequential collections of bits in memory using the binary syntax.

Data formats often have several bits within a byte that are dedicated to some purpose, such as flags or indicators. Let's say you wanted to represent some data that was eight bits long, with numbers 2 bits long, then 3 bits. The following table tells the story:

3	6	6
11	110	110

You could represent it this way:

```
iex> bit_string = << 3 :: size(2), 6 :: size(3), 6 :: size(3) >>
<<246>>
```

That makes sense. This number consists of the bits in 3, 6, and 6, placed back-to-back. The binary number is 11 110 110, which is 246.

Now, let's say you want to break 246 into three pieces. Here's how you could do so:

```
iex> << first :: size(2), second :: size(3), third :: size(3) >>  = <<246>>
<<246>>
iex> first
3
iex> second
6
iex> third
6
iex>
```

You do it the other way around! Elixir takes it apart. Of course, you're not limited to bytes that are 8 bits long. You can work with whatever byte sizes you want.

Pattern Matches for Longer Data

Sometimes, to work with patterns, we'll need to be able to match longer streams of data. As you might expect from the relationship between binaries and strings, binaries can represent variable-size data. That means we need to be able to match variable sizes of data.

Let's build a binary out of different kinds of data. Our super-sophisticated binary format has four consecutive bytes that we'll interpret as a large number, followed by a string, like this:

```
iex> large_number = <<1, 2, 3, 4>>
<<1, 2, 3, 4>>
iex> string = "This is a string"
"This is a string"
iex> combined = large_number <> string
<<1, 2, 3, 4, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 115, 116, 114, 105,
  110, 103>>
```

Keep in mind binaries are simply numbers in memory. In our case, we're choosing to look at the first part of our binary as one big 32-bit number, like this:

```
iex> <<16909060 :: size(32)>>
<<1, 2, 3, 4>>
```

Now, let's use pattern matching to extract the elements of our binary, like this:

```
iex> <<big :: size(32), rest :: binary>> = combined
<<1, 2, 3, 4, 84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 115, 116, 114, 105,
  110, 103>>
iex> rest
"This is a string"
```

Perfect! We have the two parts of our design, tucked neatly away into variables. Now you can represent any large binary format using Elixir pattern matching.

Pattern matching with binaries works much like it does with lists. You need to specify a fixed number of bytes at the beginning of a binary, but the rest of the list can be a variable size. Since strings are binaries, you can use pattern matching on them in interesting ways.

Binary Pattern Matching on Strings

This technique is often useful for matching strings with a prefix. Let's say there's a list of return codes of functions. We don't care about the successful ones. We only care about the failures. We can build a multi-headed function that uses binary pattern matching to match a "success_" prefix, like this:

```
iex> successful = fn
...> "success_" <> _suffix -> true
...> _ -> false
...> end
#Function<6.128620087/1 in :erl_eval.expr/5>
iex> successful.("success_result")
true
iex> successful.("error_result")
false
```

Then, we can take some list of return codes:

```
iex> returns = ~w[success_worked success_worked error_failed]
["success_worked", "success_worked", "error_failed"]
```

And throw away the ones that succeeded:

```
iex> Enum.reject(returns, successful)
["error_failed"]
```

Nice! The pattern match does have its limits, though. You can only match constants, not variables. And you can only match a fixed pattern at the beginning of a string.

Using these tools, you can take the specification for an image, or a communications protocol, and break down all of the individual bits, piece by piece. You only need to know the dimensions of each piece of the binary.

Before we break for the chapter, there's one more quick section we should cover: bitwise comparisons.

Working with Binary Data

You've already seen a series of operators called Boolean operators that work with true and false. There's another family of operators called BitWise operators that work with 1s and 0s as if they were true and false, respectively. These operators form the foundation of many types of math, including working with binary images or compressing many bits of data into a singular value.

You know the drill. We won't give you a comprehensive walkthrough. But we'll show you enough to find what you're looking for. To enable bitwise math, the first thing you need to do is include the code use Bitwise:

```
iex> use Bitwise
Bitwise
```

As you recall, use is a macro. Its job is to write code that writes code, making the macros and functions in the Bitwise module available for your use. There are two kinds of functions in the module. nfix functions are like + and - operators. They're used as inline operators, like this:

```
iex> import Sigil
Sigil
iex> ~b|3|
11
iex> ~b|12|
1100
iex> 3 ||| 12
15
iex> ~b|15|
1111
```

We use our binary sigil to look at a couple of numbers. Then, we combine the binary digits in 3 with the ones in 12, getting all of the digits in 15. We could accomplish the same thing in a function:

```
iex> Bitwise.bor 3, 12
15
```

Bitwise.bor/2 is the function that the ||| infix operator calls. You can get all of the exports for Bitwise and see the general shape of the API:

```
__using__/1     &&&/2           <<</2           >>>/2           ^^^/2
```

```
band/2          bnot/1          bor/2           bsl/2           bsr/2
bxor/2          |||/2           ~~~/1
```

The _using_ API supports the use Bitwise we typed earlier. The rest are prefix operators and postfix operators. Here's a table showing how to use them:

function	infix	name
band	&&&	bitwise and
bor	|||	bitwise or
bxor	^^^	bitwise x-or
bnot	~~~	bitwise not
bsr	>>>	bitwise shift right
bsl	<<<	bitwise shift left

This table shows all of the Bitwise infix operators, the functions that go with them, and what they do. Check out Elixir's Bitwise[8] module for more details.

Elixir is a rich language, so we could peruse it for intermediate tips and tricks all day, but this chapter has gone on long enough. It's time to wrap up.

Your Turn

Elixir is a robust and powerful language. Even advanced programmers can improve their skills by brushing up on certain specifications. Binaries, dates, and sigils are areas that most developers could stand to improve. This chapter is only the beginning.

Sigils, Dates, Times, and Binaries Add Power to Programs

A sigil is a bit of syntactic sugar in Elixir to represent complex or cumbersome ideas with simple text. String and list sigils can shorten code and make strings with quotes easier to understand. Date and time sigils allow rapid representation of date and time formats of various kinds. Custom sigils extend the power of sigils beyond Elixir. Knowing them will sharpen your tools, allowing you to represent complex ideas quickly.

Dates and times are common breeding grounds for bugs in many languages and Elixir is no exception. Comparison of dates and times should use Date, Time, and DateTime library comparison features rather than <. Calendars are behaviours that group together features supporting date math.

Elixir provides superlative support for dealing with binary data. Elixir binaries allow access to individual data elements at the bit or byte level.

8. https://hexdocs.pm/elixir/Bitwise.html

The best way to learn these tools is to put them into practice.

Try It Yourself

This *easy* problem starts with an Exercism problem and refactors it to use tuples.

- `secret-handshake`: Use bitwise operators to build a secret protocol.

This *medium* problem deals with sigils.

- `translate`: Make a sigil to translate an integer to binary, octal, and hex formats.

This *hard* problem deal with binary representations.

- Use pattern matching to extract the `exif` orientation of a JPEG file

Next Time

In the final Elixir chapter, we're going to shift our attention to macros. We'll use macros to write code that writes code.

CHAPTER 8

Macros

In this chapter, our programs are going to write Elixir code. That concept may sound strange to you, but thanks to the concept of macros, it's possible.

Before we get too far down that road, we should issue the standard warning. In *Programming Clojure, Third Edition [HB18]*, Stuart Halloway and Alex Miller of Clojure fame said:

> The first rule of Macro Club is Don't Write Macros.

Sometimes, languages have *important features* that are *rarely used.* In the case of macros, they're important because of their ability to build stunningly beautiful APIs. They're rarely used because the implementation and maintenance of those beautiful APIs can be complex and awkward. In fact, much of Elixir itself is written in Elixir macros.

Macros are indeed dangerous in the sense that code that relies on them too heavily is difficult to understand and maintain. In the right dose, though, macros can convert the bulk of your code to something that more fully expresses your intent.

Our advice? Avoid macros when functions or data will suffice. Use macros when they give your users a significant boost. When you use them, write your code in layers to expose a little bit of complexity at a time. Whether or not you use them, understand them so you can navigate code that has them.

Elixir macros work by letting users directly modify Elixir's AST, Elixir's abstract syntax tree. The AST is the internal representation of code in Elixir, and it's made up of regular Elixir data structures. Like Lisp, Elixir's AST is made up of the same data structures you see in the language: tuples, lists, and atoms.

Throughout this chapter, we'll look at Elixir's AST. Then, we'll use macros to build our own function using the functions quote and unquote. Finally, we'll look at a brief technique for using module attributes to help build macros.

It's going to be an interesting ride, so let's get started.

Elixir's AST

The building block of the whole Elixir language is the AST. This concept is extremely powerful, and not one that every programming language supports. Building Elixir's AST with Elixir data structures means the Elixir language and the tools it depends on can be built layer by layer, mostly in Elixir itself.

Every line of code you write has an underlying representation in the AST. Fortunately, it's easy to see the representation of any line of code. Let's see the AST for some primitive types:

```
iex> quote do "string" end
"string"
iex> quote do :atom end
:atom
iex> quote do [:list, :of, :atoms] end
[:list, :of, :atoms]
iex> quote do 1 end
1
iex> quote do {:one, 1} end
{:one, 1}
```

The quote function asks Elixir for the internal representation for the code wrapped in the do and end operators. Quoting any of the simplest Elixir data elements returns themselves. The first building blocks of the AST are these primitive data types.

It turns out that the major building block of the AST is the three-tuple:

```
iex> quote do 1 + 2 end
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

That's a bit more interesting. This simple line of code says "Give me the internal AST representation for the code 1 + 2."

We get back a three-tuple. Every AST is made up of this building block, shown here:

{ :function, [metadata_key: metadata_value], [arguments] }

function
: Since Elixir is a functional language, every element of an Elixir program is a function call. In the AST, each function call is an atom.

metadata
: Elixir sometimes needs additional information to execute a program. The *metadata* provides this data in a Keyword, a list of two tuples, where the keys and values provide information such as line numbers and imports.

arguments
: Functions take arguments, and the *argument* list is passed to the function in the first argument.

With that information in your pocket, remember the AST for 1 + 2. It looked like this:

```
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

So, :+ is the function and the arguments are 1 and 2, and there's also some metadata that may be needed to execute the program. The context is the module we're running in. We're running in IEx outside of any module, so the context is simply Elixir. The import is the module that's been imported to make this statement run, or Kernel. The + function is from Kernel, and Elixir imports it by default.

Since the arguments to functions can themselves be expressions, function arguments can be ASTs themselves! That means programs aren't merely tuples. They're trees of tuples.

You might ask what to do with syntax trees. Why, you execute them, of course!

Evaluate a Syntax Tree

If you have a syntax tree, you can run the program, like this:

```
iex> code = {:*, [], [6, 7]}
{:*, [], [6, 7]}
iex> Code.eval_quoted code
{42, []}
```

We build an AST that multiplies two numbers together. We use the multiplication function :* from Kernel, and the arguments of [6, 7]. Then, we call Code.eval_quoted/1 to execute the function.

Elixir's functions return a single value, but surprisingly, the result of eval_quoted is a two-tuple. Let's find out why.

Variables and Bindings

Let's open a brand new IEx session and set a variable, like this:

```
iex> x = 42
42
iex> binding()
[x: 42]
```

These bindings represent the values of variables in our program's execution. You can see the *bindings* from IEx. Those are the variables we've set so far.

Let's see how we might represent a variable:

```
iex> quote do x end
{:x, [], Elixir}
```

That makes sense. Instead of a function, we have an atom representing the variable name. And instead of an argument list, we have the context for the variable.

Now, let's use all of that knowledge we've gathered to work with the AST! Let's say we want to work with a binding to `ultimate_answer`, like this:

```
iex> variable = {:ultimate_answer, Elixir}
{:ultimate_answer, Elixir}
```

The AST for introducing a variable is `{:ultimate_answer, [], Elixir}`. We can build a little program to calculate that answer, like this:

```
iex> code = {:*, [], [7, 6]}
{:*, [], [6, 6]}
```

That program will execute `6*6`. Now, let's use these two parts to write our program. We can bind the variable with the `=` operator, like this:

```
iex> Code.eval_quoted( {:=, [], [variable, code]} )
{42, [{{:ultimate_answer, Elixir}, 42}]}
```

Beautiful! Notice that we now have a variable binding to `:ultimate_answer` in the context of `Elixir`. The only problem is that our program is wrong. That's OK. Elixir code *is the same as Elixir data!* That means we can change it because we know the structure of the AST, like this:

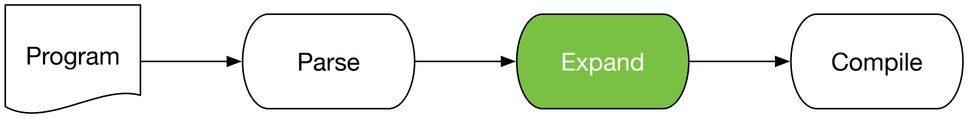
```
iex> {function, metadata, [arg1, _arg2]} = code
{:*, [], [6, 6]}
iex> revised_code = {function, metadata, [arg1, 7]}
{:*, [], [6, 7]}
iex> Code.eval_quoted( {:=, [], [variable, revised_code]} )
{42, [{{:ultimate_answer, Elixir}, 42}]}
```

That's amazing! Don't gloss over what we've done. We took an AST, changed its representation with an Elixir program, and we executed the revised program.

This is the essence of Macro programming.

Unquoting, Quoting, and defmacro

Let's take a moment to recap. Elixir programs are made up of ASTs. These trees have the representation of all Elixir code. Building the AST doesn't happen all at once, though. Each program is compiled with several different steps. The following figure tells the story.



After a program is loaded, Elixir translates the program into an initial AST in a step called *parsing*. Then, Elixir applies all macros to the AST in a step called *macro expansion*. Only then is the program turned into its final executable form.

Because Elixir is a language built mostly in itself, the macro expansion step is important. Let's look at how it's done.

The defmacro expression means *define a macro*. Each defmacro statement takes valid Elixir code that compiles and modifies the AST to build other Elixir code that compiles. This is the *macro expansion* step in the previous figure.

A Simple Macro

Let's build a prettier interface into our Roman numerals program. You can find the original at Groxio's Github.[1] Initially, our plan is to build a module with all of the Roman numerals built in, looking something like this:

```
defmodule R do
  def i, do: 1
  def ii, do: 2
  def iii, do: 3
  ...and so on...
end
```

Then, your users could quickly access a Roman numeral like R.iii, or by importing the file, by simply typing iii. Next, we'll want to create a macro.

1. https://github.com/groxio-learning/progpass-elixir/

Define a Macro with defmacro

We're finally ready to define a macro. Open up a new file in lib/roman_macro.ex. The base of our macro is going to look like this:

```
quote do def roman, do: decimal
```

We'll wrap that code in a macro, like this:

```
defmodule RomanMacro do
  defmacro roman_function(roman, decimal) do
    quote do
      def roman(), do: decimal
    end
  end
end
```

Then, create a module called lib/hardcoded.ex to use the macro, like this:

```
defmodule Hardcoded do
  require RomanMacro

  RomanMacro.roman_function(:i, 1)
  RomanMacro.roman_function(:ii, 2)
end
```

The require asks Elixir to do macro expansion using the macros in RomanMacro. When you try that much out, you'll find the code doesn't compile:

```
** (CompileError) lib/hardcoded.ex:4: undefined function decimal/0
  ...
```

That's because instead of using the roman and the decimal that we pass in, Elixir is trying to actually use the words roman and decimal, and those don't exist. We need a way to use the variables in our program. We need a sort of interpolation for macros.

That feature is called unquote. Change your macro a tiny bit to unquote the decimal and roman values, like this:

```
defmodule RomanMacro do
  defmacro roman_function(roman, decimal) do
    quote do
      def unquote(roman)(), do: unquote(decimal)
    end
  end
end
```

Perfect. We should pick up the roman and decimal arguments from our function. Now, try it out:

```
iex> recompile
```

```
Compiling 3 files (.ex)
:ok
iex> Hardcoded.i
1
iex> Hardcoded.ii
2
```

So far, so good! Let's apply the same techniques to a module that defines all Roman numerals!

Introduce a DSL

Domain specific languages (DSLs) are tiny, specific languages written in a generic programming language. If you know where to look, Elixir embeds DSLs everywhere. ExUnit uses a DSL to build unit tests. Ecto Query uses a DSL to express queries and schemas. Phoenix uses a DSL to build a router table.

If you think about it, our code is building a tiny DSL that expresses Roman numerals. Some of the work is defining an API that our consumers can use. Other pieces of the problem involve packaging up our DSL in a convenient package our customers can consume.

Our Roman API already has some convenient functions that return a map of Roman numerals and an API to convert singles. It would be nice to introduce a feature that allows users to announce their intention to use our functions like iii to express individual numbers in roman notation.

To do so, we'll need to write two bits of code. One will define all of the functions in a module, and another will import them.

Define Many Functions with unquote and def

It turns out that you don't need to use quote and unquote with defmacro. They work fine inside modules. Remember, we already have a function called Roman.map/0 that returns a map with all of the Roman numerals and their decimal values in them. All we need to do is iterate over them at compile time, calling def on each one.

Crack open a new file called numbers.ex that we'll use to hold all of our Roman numerals, like this:

```
defmodule Roman.Numbers do
  Enum.each(Roman.map, fn {roman, decimal} ->
    def unquote(roman)(), do: unquote(decimal)
  end)
end
```

Perfect! That code should look somewhat familiar. The def unquote(roman)(), do: unquote(decimal) does the bulk of the work. Each one defines one function within the module. The unquote statements allow us to substitute values from our Roman.map list.

The first value in Roman.map will be {:i, 1}, and we'll match roman to :i and decimal to 1. That means when Elixir replaces unquote(roman) with :i and unquote(decimal) with 1, we're left with def i(), do: 1!

When you try it out, you'll see that it works:

```
iex> Roman.Numbers.xix
19
```

It works perfectly. The last step is to write a little code to do the import so our API clients won't have to type Roman.Numbers.

Smooth Out the End User Experience with __using__

The __using__ macro helps developers set up their macros with any initial macros, aliases, or imports. They're easy to use. Let's add a __using__ function to Roman that will import all of the functions in Roman.Numbers.

When an API client calls use SomeModule, Elixir will call the __using__ function within SomeModule. We'll use the technique to add the imports we need for our Roman numerals.

Add this code to lib/roman.ex, like this:

```
defmacro __using__(_opts) do
  quote do
    import Roman.Numbers
  end
end
```

Now, our users don't need to know anything about the underlying modules behind Roman. We can simply have our users use Roman to consume the API, like this:

```
iex> use Roman
Roman.Numbers
iex> iii
3
iex> iv
4
iex> ix
9
```

All is not well in IEx land, though.

```
iex> v
** (CompileError) iex:7: function v/0 imported from both Roman.Numbers
   and IEx.Helpers, call is ambiguous
iex> i
** (CompileError) iex:7: function i/0 imported from both Roman.Numbers
   and IEx.Helpers, call is ambiguous
```

We need to be careful when we export these functions because some of the Roman numerals conflict with IEx helpers, namely `i` and `v`. Still, we can be happy with the results. Our users can now use the Roman numerals we specify, even if we can't call `use Roman` and use every number within IEx. This code serves as a useful example for how to write code that writes code.

This chapter has been short, but intense. It's a good time to wrap up.

Your Turn

Elixir macros are a rarely used but important feature. They're used to write both Elixir code and beautiful APIs called domain specific languages.

Macros

Underneath, Elixir code is represented within the abstract syntax tree, or AST. A macro is code that writes code. Macros take a bit of the AST and translate it into other ASTs. The `quote` function lets developers convert Elixir code into the AST format, and the `unquote` function allows interpolation of Elixir code. Together, they can make existing Elixir code do new and interesting things.

A DSL is a domain specific language. While Elixir is a general-purpose programming language, a DSL uses the features of Elixir, especially macros, to represent other concepts. For example, the Ecto DSL represents database queries and schemas, and the Plug router uses a DSL to represent the routers for a web server.

The `__using__` macro, in conjunction with `use`, allows an API designer to quickly import and alias common modules. The `use` command fires the `__using__` macro for a module. This technique can smooth out rough edges for an API.

Try It Yourself

You may have noticed that our exercises are getting increasingly open-ended. That's intentional. Open-ended exercises will help intermediate and expert developers learn more quickly.

These two *medium* problems use macros, `quote`, and `unquote`.

- Write a macro that uses the same technique our Roman DSL used to convert the number RN to a Roman numeral where N is some number between 1 and 1000, inclusive.
- Write an unless macro that works like if, one that takes :else and :do clauses. It should take the form unless clause, do: expression1, else: expression2.

This *hard* problem uses macros to implement a state machine.

State Machine Problem

Consider this code:

```
defmodule TurnStile do
  use StateMachine,
    initial: :closed,
    states: [
      closed: [coin: :open],
      open: [person: :closed]
    ]
end
```

The preceding code expands to:

```
defmodule TurnStile do
  ...

  def new(), do: :closed

  def enter(:open), do: :closed
  def coin(:closed), do: :open
end
```

This produces the behavior:

```
iex> import TurnStile
TurnStile
iex> new |> coin
:open
iex> new |> coin |> enter
:closed
```

Implement this state machine using a macro.

This concludes our journey through Elixir. A combination of productive syntax with sigils, good support for concurrency, and macros allowing language extensions makes Elixir a powerful tool that can grow with you for years to come. If you like what you see and aren't yet a full Groxio subscriber, we invite you to join. Either way, we hope you've enjoyed our brief exploration and learned things about Elixir you didn't already know. We'll see you somewhere down the road!

Bibliography

[Alm18] Ulisses Almeida. *Learn Functional Programming with Elixir*. The Pragmatic Bookshelf, Raleigh, NC, 2018.

[Hal18] Lance Halvorsen. *Functional Web Development with Elixir, OTP, and Phoenix*. The Pragmatic Bookshelf, Raleigh, NC, 2018.

[HB18] Alex Miller with Stuart Halloway and Aaron Bedra. *Programming Clojure, Third Edition*. The Pragmatic Bookshelf, Raleigh, NC, 2018.

[IT19] James Edward Gray, II and Bruce A. Tate. *Designing Elixir Systems with OTP*. The Pragmatic Bookshelf, Raleigh, NC, 2019.

[Tho18] Dave Thomas. *Programming Elixir 1.6*. The Pragmatic Bookshelf, Raleigh, NC, 2018.

Thank you!