The Phoenix Tutorial

Jeffrey Biles

Introduction

This book will focus on teaching you the fundamentals of web development.

Web development is a powerful tool that can help you in many endeavors–whether it's building a startup that can change the world, getting a job to provide for your family, making a web app as a creative expression, or using web development as a secret weapon to increase your effectiveness in your current career, I think that you'll be glad you invested the time into learning this valuable skill.

We're going to learn web development using what I believe are some of the best tools available: Elixir (a blazing-fast programming language with friendly syntax), Phoenix (a modern web framework that combines functional programming with the best of Ruby on Rails), and Ecto (a maintainable system for interacting with the database). (More on that later in this introduction)

What are we building?

We'll be taking a **hands-on approach** to learning, which means you'll be practicing each concept quickly after you learn it, applying it in a real application when possible. In addition to the code given in each chapter (which you should type into your computer and run), there will also be exercises, which I highly recommend you do–if you want to remember what you're learning.

Throughout this book we're going to be building the Star Tracker app, which boldly goes where no inventory management app has gone before.

Insert picture of finished app here

It's not super fancy, but it gives us many opportunities to solidify the core concepts you'll need to start creating your own Phoenix apps (and understand the more advanced Elixir and Phoenix books, should you choose to read them).

Why this book, and not Learning Phoenix or another book?

There are lots of great resources for intermediate and advanced developers to learn Elixir and Phoenix, but relatively little for beginners. In addition, the resources that do exist tend to rush past the core web framework features in order to get to the cool advanced features that set the framework apart, like Channels and OTP.

This book aims to be your first Elixir and Phoenix book (and maybe even your first web development book), but not your last. We'll go into the basics of the language and the framework, giving you a solid foundation so you can crank out simple web apps and be prepared to read and understand the books written on more advanced topics.

That said, this is not meant to be your first introduction to programming.

Prerequisites

While I don't expect you to have familiarity with Elixir or web programming, I do expect you to know how to put together a simple program in *some* language. If you don't meet that standard, here are some good (free) resources:

- Intro to Computer Science (A video course that teaches Python basics)
- How to Design Programs (this Lisp book is how I learned)
- Invent With Python (Book series that involves making lots of games)

Why Phoenix and Elixir?

> This section uses advanced terminology in order to compare Phoenix and Elixir to other tools. If you don't understand something here that's okay–just read to get the general gist of it. There will be a summary at the end.

The current king of the backend Web Framework world is Ruby on Rails–it's what is taught at hundreds of Coder Bootcamps and used at thousands of startups (as well as some large companies). However, many of the leading lights of the Ruby and Rails worlds have moved on, and a significant number of them have moved to Elixir and Phoenix. Why is that?

The answer is that Elixir combines the power, speed, and reliability of the Erlang VM with the beauty and expressiveness of Ruby syntax. In one stroke it fixes the biggest complaint about each language- speed for Ruby and syntax for Erlang.

The Elixir ecosystem has also learned a lot from both languages. From Ruby it takes many of the most popular tools and combines them into one tool with a consistent interface: Mix. From Erlang it takes… the entire Erlang ecosystem;

Figure 1: Twitter joke about Rails and Phoenix

you can use any Erlang package you want in Elixir (just be prepared to look at some not-as-beautiful code).

Phoenix itself has benefited greatly from seeing what Rails has done. It's taken what worked well (convention over configuration, REST, the Rails MVC style, friendly build tooling, etc.) while doing some much-needed modernization. Some of the modernizations are made available simply by escaping the inertia of the Rails codebase and creator DHH's opinions. Others are unlocked due to the increased power and features of Elixir as compared to Ruby.

So here are the biggest advantages of Phoenix and Elixir, in fancy listicle form:

- SPEED

Elixir is an order of magnitude (~10X) faster than Ruby right out of the box. This not only gets you faster response times and lets you handle more requests per box, but it eliminates a whole class of performance hacks and opens the door for doing things that would not be wise in Ruby.

- Reliability

Elixir is built on the Erlang VM, which has created systems with "nine nines" of reliability- that is, it was working 99.9999999% of the time. This reliability isn't necessarily because individual parts crash less often, but because it can recover seamlessly from crashes- a "self healing" network of processes. We won't get into this mechanism in detail in this book.

- Real-time features

Channels/Sockets that are just as easy as in Node (the current Sockets king), but in Elixir they're more performant. This means that "real-time" communication between server and client is a breeze compared to other languages and frameworks.

- Functional Programming Style

This can occasionally cause frustration for those coming from an Object-Oriented language (such as Ruby), but once you become acclimated it means fewer bugs and more readable code. You also get really cool features like

Pattern Matching and piping that are not available in Object-Oriented languages.

Technobabble: When to use Phoenix vs Other web frameworks?

Phoenix is still a newer technology, but it's well past 1.0 and stable enough to run production apps (and the underlying technologies– Elixir and Erlang–are rock solid).

If you're starting a new project, choose Phoenix! But if you have a legacy project then there are more tradeoffs to consider.

I'd say that if you are having performance problems which can't be solved with basic solutions (such as fixing n+1 database queries) then it may be worth it to switch to Phoenix. High availability requirements are another good reason. It also may be worth it if your project involves lots of concurrency or sockets- stuff that most languages and frameworks do poorly, but Elixir and Phoenix do really really well.

If you're investigating Phoenix, then it's likely that you fall into one of those camps. However, if you're happy with your current tech, or if you're unhappy for different reasons, consider whether the benefits of Phoenix will be worth the cost of a rewrite.

Why Phoenix? (for beginners)

If you're a beginner that didn't quite understand all of the last section, here's the tl;dr on why Elixir/Phoenix/Ecto can be a better choice than Ruby/Ruby on Rails/ActiveRecord or other popular web programming tools:

- Phoenix is an order of magnitude faster (due to Elixir's use of the Erlang VM)
- Your Phoenix code is significantly less likely to descend into spaghetti code (due both to Elixir's functional programming and Phoenix and Ecto's learning many lessons from how web apps can go wrong)
- You'll be prepared to take advantage of Elixir's advanced features *when their use becomes necessary*

Conclusion

I hope you're as excited as I am to begin! Turn the page and we'll start installing Elixir.

Changelog

- **Website Created**
- Updated to Elixir 1.7.1
- Fixed typos/grammar

0.1.2

- Added a Captain's Log for snake_case vs PascalCase
- Added technobabble for assignment vs pattern matching
- Added a note on tab auto-complete in iex
- Renamed `recombinePhraseRequired` function to `recombine`
- Updated Arguments exercise to be easier to understand
- Updated Maps exercise to not accidentally resemble a Ruby on Rails convention
- Added paragraph on destructuring Maps
- Removed errant camelCases and incorrect references to hashes
- Updated String section for clarity
- Fixed some typos/grammar stuff

0.1.1

- Rewrote the introduction
- Rewrote immutability section, starting with simpler examples and showing an example of mutable data
- Rewrote the case statement section, starting with a simpler example
- Demarcated all code samples in exercises as code (so they are easier to copy/paste)
- Indent iex output lines so they're easier to read
- Fixed many typos
- Turned all usages of "method" to "function"
- Made pattern matching exercises a bit more clear
- Added a captain's log about the command line conventions
- Added a captain's log about the interpreter
- Added a captain's log about loading elixir files
- Added a captain's log about function ordering
- Added flavor bits to captain's logs, to hint at a story
- Added acknowledgements
- Added changelog
- Added copyright and license
- Pdf version: links in footnotes
- Pdf version: Give codeblocks a background color
- Pdf version: Decreased margins so codeblocks fit
- Many small improvements/rewordings

0.1

- **Introduction and first section created.**

Acknowledgements

A huge thanks to those who have offered feedback on the manuscript: Derek Wood, Franco Barbeite, Giuseppe Caruso, Richard Poole, Matthew Davis, Peter Karth, and Joep Stender. This book would be much worse without them.

Copyright and License

Installing Elixir

Hello World

Welcome to The Phoenix Tutorial. I'm so glad you decided to join us!

Captain's Log: The Introduction

Stardate: far future.

I have just discovered the first fragments of this fascinating manuscript detailing the Phoenix technology that underlies our fleet.

At first I wasn't sure what I was reading, since it seemed to jump straight into technical instructions, but then I went back and read the Introduction and it all made sense.

I'll be recording my observations, both for myself, and so that others may continue my research should anything… happen.

As we said in the intro, we'll be taking a **hands-on approach** to learning. And that means that before we can do anything, you're going to have to get Elixir installed on your computer.

Installation

Installing Elixir is the most system-dependent installation step of this entire book. Each function shared will install both Elixir and Erlang, the language which Elixir builds on top of.

Check which operating system you are using and follow the associated instructions.

Captain's Log: Command Line Notation

When we find a dollar sign at the beginning of a line of code, it appears to signify that everything following should be typed out on the reader's "command line".

The command line was the 21st-century equivalent of our ship computer; you tell it a command (by typing with your fingers–such barbarians!) and it does exactly what you ask. The most common name for this command line was "bash" (as I said–violent barbarians!), but there were several other variants in use.

> I will continue reporting my findings as I uncover more of this fascinating 21st-century document.

Mac

1. Install or update homebrew

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

or

```
$ brew update
```

2. Install Elixir

```
$ brew install elixir
```

Windows

Download the Installer and run it: https://repo.hex.pm/elixir-websetup.exe

If you're using Windows, make sure you have the Bash shell installed (https://msdn.microsoft.com/en-us/commandline/wsl/about). This new feature, available on Windows 10 and above, will allow you to use the same shell commands as Linux and Mac users.

Other

You can find instructions for other platforms (mostly Linux flavors) at http://elixir-lang.org/install.html.

If your Linux is Debian-based, be sure to explicitly install Erlang so you can get all the necessary packages (instructions here: http://www.phoenixframework.org/docs/installation#section-erlang)

Installing Hex

From here on out things get (generally) platform-agnostic. There are a couple of extra commands for those on Debian-based systems, which can be found on the Phoenix Installation Guide: http://www.phoenixframework.org/docs/installation

Hex is the Erlang package manager, which lets us specify and download our dependencies. Mix is our Elixir task runner. We'll use Mix (which came with Elixir) to install Hex.

```
$ mix local.hex
```

We'll talk more about both of these later.

Conclusion

We've got Elixir installed now and we're ready to start learning! In the next couple chapters we'll go over the basics of the language, with a preference towards what is most commonly used in Phoenix.

Getting Started with Elixir

Elixir is an awesome language that is filled to the brim with exciting features. In part 1 we're only going to cover the basics–enough to get started with Phoenix. Then in subsequent parts of this book we'll add on more language constructs as needed.

The Elixir Interpreter

In order to show off the language features, we'll want to jump down to something simpler than a full Phoenix app. To do that, we're going to use the Elixir Interpreter from the command line.

Open it up by typing `iex` in your command line.

```
$ iex
```

Now you should see something like the following:

```
$ iex
  Erlang/OTP 21 [erts-10.0.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [hip

  Interactive Elixir (1.7.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

This is good! That means it's working.

> This guide uses Elixir 1.7.1, but it should work as long as the first number (major version) is '1' and the second number (minor version) is 7 or greater. The third number is called the patch version, if you're curious.

Now you can type Elixir code in and it'll run right there (after you hit the Enter key).

```
iex(1)> 2 + 3
        5
```

Boom. Type in `2 + 3`, hit enter, and it shows the result `5` in the line below.

> Your interpreter will show the results all the way to the left, but I've indented the results in this book so input/output lines are easier to differentiate and so output is easier to match up with the input that produced it.

When we build our Phoenix app we'll spend the majority of our time interacting with it in other ways, but we'll always have the option to drop back to the interpreter and play around.

```
iex(1)> 2 + 3
        5
iex(2)> (2 + 3) * 3
        15
iex(3)> 2 + (3 * 3)
        11
iex(3)> num_tacos = 3
        3
iex(4)> num_burritos = 3 * 5
        15
iex(5)> num_tacos + num_burritos
        18
```

The above demonstrate numbers, basic math, order of operations with parentheses (this will apply to more than just math), and the assignment and use of variables.

Captain's Log: Command Line Interpreter

Our scientists have discovered how to access the Command Line Interpreter (iex) that this manuscript speaks of.

At first we thought that we had forever trapped our computers in the interpreter, but we eventually discovered that we could exit by hitting Ctrl + C twice in a row. This is good for when you're done, or if you get the interpreter into a weird state you don't understand.

Exiting and re-entering also solved another mystery- the numbers in the interpreter prompt. A few more superstitious in our crew thought them at first some sort of sinister countdown, but they are simply the number of lines since the interpreter started.

Interpreter Exercises

What is displayed when you type each of these into the command line interpreter? After thinking about it, test your answer.

1. `2 * 10`
2. `(2 * 10) + 3`
3. `2 * (10 + 3)`
4. `num_tacos = 2 + 3; pepper_per_taco = 3; num_tacos * pepper_per_taco`

(If you can't guess what `;` does, just go ahead and play around with it. Playing and testing your hypotheses is cheap in the command line!)

Strings

There are two types of Strings in Elixir.

```
iex(1)> "hello universe"
        "hello universe"
iex(2)> 'hello universe'
        'hello universe'
```

For the purposes of this book we'll be using double-quoted strings (sometimes called Binaries). Single-quoted strings (sometimes called Char Lists) are mostly used when interfacing with old Elixir libraries.

Technobabble: Binaries vs Char Lists

Single-quoted strings are Lists of Characters and double-quoted strings are UTF-8 encoded binaries represented by a series of graphemes. Even though we think of both of them as "Strings", we use different sets of functions on each.

For a fun demonstration of a quirk of Char Lists, try putting `[111, 111, 112, 115]` in the command line.

Here's some basic inline usages of strings

```
iex(1)> "hello" <> "universe"
        "hellouniverse"
iex(2)> "hello" <> " universe"
        "hello universe"
iex(3)> place = "universe"
        "universe"
iex(4)> "hello #{place}"
        "hello universe"
```

Aside from these inline usages, most action on strings happens via functions being applied to them. Here's an example:

```
iex(5)> phrase = "hello vast universe"
        "hello vast universe"
iex(6)> String.split(phrase, " ")
        ["hello", "vast", "universe"]
iex(7)> upcase_phrase = String.upcase(phrase)
        "HELLO VAST UNIVERSE"
iex(8)> phrase <> " -> " <> upcase_phrase
        "hello vast universe -> HELLO VAST UNIVERSE"
```

The string usage above is simple but allows us to demonstrate two important facets of how Elixir works.

First, when we call `split` and `upcase`, we're not changing the original phrase. Instead, we're creating a new copy of the phrase, which can be assigned to a variable (such as `upcase_phrase`). The original string is "immutable"–that is, it cannot be changed (although a new immutable value could be assigned to the variable `phrase`). We'll go over immutability in more detail in chapter 5 of this section.

Second, instead of calling a function which is stored on the string itself (`phrase.split(" ")`) we take our function and apply it to the string (`String.split(phrase, " ")`). This may seem like a trivial difference, but in fact it is vital to understanding the functional nature of Elixir.

Functional vs. Object-Oriented

In an object-oriented language the phrase would be an instance of the class String, and would therefore have available to it all the String functions. Hence you could call `phrase.split(" ")`.

In a functional language, the phrase is the data, which can be acted upon by functions. These functions are organized, so the most common functions which can act upon strings are stored in the `String` module, but what's important is we're separating out the data from the function instead of throwing them all in one "object".

If this doesn't make sense to you, that's okay–just move on. We'll cover modules in the next chapter (and make our own), and the functional style of coding will become clearer to you as you write more Elixir.

String exercises

What is displayed when you type each of these into the command line? After thinking about it, test your answer.

1. `String.split("hello amazing universe", " ")`
2. `String.split("hello amazing universe", "i")`
3. `setting = "stun"; "set phasers to #{setting}"`

Enum

The `Enum` module is meant for working with Enumerables. Although there are others, the most common type of Enumerable is a list, so that's what we'll be working with today.

```
iex(1)> split_phrase = ["hello", "vast", "universe"]
        ["hello", "vast", "universe"]
iex(2)> Enum.count split_phrase
        3
iex(3)> Enum.join split_phrase, "-"
        "hello-vast-universe"
iex(4)> Enum.member? split_phrase, "universe"
        true
iex(4)> Enum.member? split_phrase, "univers"
        false
```

Once again we have our functions which are organized by module (`Enum`), which are then applied to the data (`split_phrase`).

We're also demonstrating a parentheses-free function syntax. It's equivalent to having parentheses, except it's a bit cleaner (but, in some cases, more ambiguous. In those cases parentheses should be added back for clarity).

Char Lists vs Binaries

As an aside, remember how we have single-quoted strings (arrays of Chars) and double-quoted strings (binaries)? Those types end up being important. We use `Enum` functions on single-quoted Strings, and `String` functions on double-quoted Strings.

```
iex(1)> Enum.count 'hello'
        5
iex(2)> Enum.count "hello"
        ** (Protocol.UndefinedError) protocol Enumerable not implemented for "hello"
            (elixir) lib/enum.ex:1: Enumerable.impl_for!/1
            (elixir) lib/enum.ex:146: Enumerable.count/1
            (elixir) lib/enum.ex:467: Enum.count/1
iex(3)> String.split 'hello universe'
        ** (ArgumentError) argument error
            (stdlib) :binary.split('hello universe',
            [" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ",
            <<194, 133>>, " ", "\t", "\n", "\v", "\f", "\r"], [:global, :trim_all])
            (elixir) lib/elixir/unicode/properties.ex:344: String.Break.split/1
iex(4)> String.split "hello universe"
        ["hello", "universe"]
```

> Want to see something cool? Press 'tab' in iex to trigger an auto-complete. So you type in "Stri", hit 'tab', and it'll complete it to "String".
>
> Even better, if there are multiple options, it will show you all of them

that match your search. Try hitting tab after typing in "String." (make sure to include the dat)- it'll show you all of the functions available on that module.

Enum Exercises

What is displayed when you type each of these into the command line? After thinking about it, test your answer.

1. `Enum.member?(["hello", "vast", "universe"], "hello")`
2. `Enum.member?(["hello", "vast", "universe"], "hello vast universe")`
3. `Enum.join(["hello", "vast", "universe"], "! ")`
4. `Enum.join(["hello", "vast", "universe"], "joining with this string is a very bad idea")`

Chaining functions together

Let's say we wanted to count how many words are in a certain phrase. We don't have a "word count" function, but we can get there by applying multiple functions in a row.

There are several ways we could do this. Here's the first.

```
iex(1)> phrase = "boldly going where no man has gone before"
        "boldly going where no man has gone before"
iex(2)> split_phrase = String.split(phrase)
        ["boldly", "going", "where", "no", "man", "has", "gone", "before"]
iex(3)> count = Enum.count split_phrase
        8
```

If you're unsure of where you're going and what to check at every step, that's probably the best way to go. However, for production code it can look a bit messy. Here's another way that's a bit more compact:

```
iex(1)> Enum.count(String.split("boldly going where no man has gone before"))
        8
```

Here we're nesting the function calls so the result of one gets fed into the other. This is more compact, but it can still look messy.

A great solution to this is the pipe syntax.

The Pipe Syntax

In the pipe syntax, we can take the results of one function and "pipe" it as the first argument of the next function.

```
iex(1)> "boldly going where no man has gone before" |> String.split |> Enum.count
        8
```

It's a couple more characters than the previous one, but it's far more readable. You have the subject first (the phrase), and then you pass that phrase to `String.split`, then pass the results of that to `Enum.count`.

If you have multiple arguments, you just pass the second (and third and fourth, etc.) arguments with the function.

```
iex(1)> "boldly going where no man has gone before" |> String.split |> Enum.join("-")
        "boldly-going-where-no-man-has-gone-before"
```

When you're piping functions, be sure to use parentheses for your arguments–the compiler can usually infer them, but the pipe syntax makes that more difficult and so it's best to be explicit.

Working with pipes is a core part of the Elixir experience, and so most functions you encounter will be designed with piping in mind–they'll take *data* as their first argument, and then configuration for their other arguments. You can see this with `Enum.join`, which takes in the list as the first argument and then `"-"` as the second argument.

Pipe Exercises

What is displayed when you type each of these into the command line? After thinking about it, test your answer.

1. `"boldly going where no man has gone before" |> String.length`
2. `"boldly going where no man has gone before" |> String.split(" ") |> Enum.member?("boldly")`
3. `"boldly going where no man has gone before" |> String.upcase |> String.split(" ") |> Enum.join("... ")`

Conclusion

Already each individual line of Elixir should be looking less mysterious than before, but there's still plenty to learn. In the next chapter we'll go over how to create and use custom functions.

Functions

So far in our exploration of Elixir we've typed our code directly into the Elixir interpreter, but for complex functions – not to mention building an entire program – we'll want a place to store our functions.

In this chapter we'll learn how to store our code by defining modules and functions.

Storing Code: defmodule and def

Start by creating a file. I'm going to call mine `basic_elixir.ex`, but the only part of this name that you are required to copy is the `.ex` file extension, which signals that this is an Elixir file.

Then we'll create the minimum viable module in this file:

```
defmodule LearningElixir do
  def hello do
    "boldly going where no man has gone before"
  end
end
```

Our module, defined by `defmodule` (DEFine MODULE), is `LearningElixir`. Attached to that we have a `do` and its closing `end`. This pair of "delimiters" indicates that anything in between them is part of the `LearningElixir` module. Some languages use whitespace or curly braces as their delimiters, but Elixir has copied Ruby and uses the more descriptive `do` and `end` keywords.

Our function `hello` is defined by `def`. It also has a do/end delimiting block. Inside that block is the string "boldly going where no man has gone before", which is what is returned when we call that function.

Let's go ahead and load that file in the Interpreter.

```
iex(1)> import_file "basic_elixir.ex"
        {:module, LearningElixir,
         <<70, 79, 82, 49, 0, 0, 4, 52, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 134,
           0, 0, 0, 13, 21, 69, 108, 105, 120, 105, 114, 46, 76, 101, 97, 114, 110, 105,
           110, 103, 69, 108, 105, 120, 105, 114, 8, ...>>, {:hello, 0}}
iex(2)> LearningElixir.hello
        "boldly going where no man has gone before"
```

Captain's Log: Loading Elixir Files

It took much experimentation, but we have finally discovered how to load code into the ship's interpreter.

> Our error was that we were in the wrong folder- the computer didn't know how to find `basic_elixir.ex`. Our solution: be in the same folder as the file when we start the Elixir interpreter.
>
> Our scientists have reported that we may be able to access the file from a different folder by specifying the path but, truth be told, I prefer the simple way.

We load the file into the Interpreter with the command `import_file`. We can ignore the output for now- only notice that we see both `LearningElixir` and `hello` in various forms.

Then we call our function with `LearningElixir.hello` and get back the expected string.

> Note that we precede the function name with the name of the module within which it resides. Apply the logic to our commands from the previous chapter and we can see that `String.split` was calling the `split` function on the `String` module.

Next let's define a new function, which has the code that we had previously inputted directly into the command line:

```
defmodule LearningElixir do
  def phrase do
    "boldly going where no man has gone before"
  end

  def recombine do
    phrase |> String.split |> Enum.join("-")
  end
end
```

We've renamed `hello` to `phrase`, then used it in our `recombine` function. Note that since we're within the `LearningElixir` module we don't need to precede `phrase` with `LearningElixir`- `phrase` is currently directly available because it's "within scope".

> **Captain's Log: Scope**
>
> Scope can be a scary word, but here's a basic way to think about it. If you're in your ship's common room, and you say "I would like to sit on the couch", you don't have to specify which of the millions of couches you're sitting on. You're in the common room scope, so when you're trying to think of couches, the one in the ship's common room comes to mind first.

> It's the same reason that if you talked to someone in the old United States about "the civil war", they'll immediately think of the American Civil War, not the Spanish Civil War, the American Revolution, or the Klingon Civil War. That's because they're scoped to the United States.
>
> Scope isn't as big a deal in Elixir as it is in Object-Oriented languages, but it's still important to understand.

The rest of our `recombine` function is just like what we previously did directly in the interpreter.

Let's load and call this in the interpreter.

```
iex(3)> import_file "basic_elixir.ex"
        warning: redefining module LearningElixir (current version defined in memory)
          iex:1

        warning: variable "phrase" does not exist and is being expanded to "phrase()",
        please use parentheses to remove the ambiguity or change the variable name
          iex:7

        {:module, LearningElixir,
         <<70, 79, 82, 49, 0, 0, 5, 40, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 182,
           0, 0, 0, 18, 21, 69, 108, 105, 120, 105, 114, 46, 76, 101, 97, 114, 110, 105,
           110, 103, 69, 108, 105, 120, 105, 114, 8, ...>>, {:recombine, 0}}
iex(4)> LearningElixir.recombine
        "boldly-going-where-no-man-has-gone-before"
```

The result of `LearningElixir.recombine` is what we would expect if we straightforwardly combined what we previously knew.

Less expected are the two warnings.

The first warning is the result of importing the same file (and thus defining the same module) twice in one interpreter session. If we had restarted the interpreter in between loading the file then this warning would not show up.

The second warning is because `phrase` is somewhat ambiguous- it could be either a variable or a function. The interpreter automatically (and correctly) expands it to `phrase()`, which is the less ambiguous way to call a function. While the language creator recommends changing all ambiguous instances to using the parentheses, you'll likely see a lot of production code without those parentheses because this warning was added relatively recently, in late 2016 (with Elixir 1.4).

There are two ways to solve the ambiguity:

```
defmodule LearningElixir do
  def phrase do
    "boldly going where no man has gone before"
  end

  def recombine1 do
    phrase() |> String.split |> Enum.join("-")
  end

  def recombine2 do
    LearningElixir.phrase |> String.split |> Enum.join("-")
  end
end
```

The first is exactly what's suggested by the warning- add in the parentheses. The second makes explicit the fact that `phrase` is defined on the `LearningElixir` module- thus removing the ambiguity. I personally prefer the second solution (it makes the function more portable because you don't have to worry about scope), but either works.

Finally, now that we're in a proper file we don't have to define everything on one line.

```
defmodule LearningElixir do
  def phrase do
    "boldly going where no man has gone before"
  end

  def recombine do
    LearningElixir.phrase
    |> String.split
    |> Enum.join("-")
  end
end
```

This version of `recombine` does exactly the same as our last version, but now instead of having everything in one line we have the pipes lined up vertically. This can be very convenient for seeing at a glance how a function is composed.

Exercises

1. Create the `LearningElixir` module, with the `phrase` and `recombine` functions in it. Import it on the command line, then run `LearningElixir.recombine`.
2. Within that module, create the `upcase_phrase` function, which returns the phrase, but all in upper case letters. Use the `phrase` function in your solution- you're cheating if you just type out the phrase manually in upper

case.

```
> iex(1)> LearningElixir.upcase_phrase
          "BOLDLY GOING WHERE NO MAN HAS GONE BEFORE"
```

We introduced the relevant `String` function in the last chapter.

Conclusion

Now you know how to create a module that organizes your functions. This increases your ability to organize your code aesthetically (lining up pipe operators) and opens up many new possibilities. However, our current understanding is still very limited.

In the next chapter, we'll show how to create more flexible functions by giving them arguments- and by introducing our first instance of Pattern Matching, the functional programming design pattern that you'll soon grow to love.

Arguments and Pattern Matching

Using Arguments

The `recombine` function is pretty cool, but it always uses the default phrase defined elsewhere in the module. What if we wanted to allow the user the option of putting in their own phrase?

First we'll see how to *make* them input their own phrase:

```
def recombine(phrase) do
  phrase
  |> String.split
  |> Enum.join("-")
end
```

This is the first function we've created that requires an argument, but we've used functions with arguments before. When we call it we must feed it a phrase as input (with or without parentheses)

```
iex(1)> LearningElixir.recombine("hello world")
        "hello-world"
iex(2)> LearningElixir.recombine "hello world"
        "hello-world"
```

Next we'll figure out how to make the phrase argument *optional*.

There are several ways to do that. The first solution we show, having a default value, will be typical of non-functional languages. The second solution, which is modeled after a common Javascript pattern, will show us how different the two language's capabilities are and introduce us to the concept of Arity. Finally, the third solution will take advantage of Elixir's functional abilities and give us our first taste of Pattern Matching.

Default Value

```
def recombine(phrase \\ LearningElixir.phrase) do
  phrase
  |> String.split
  |> Enum.join("-")
end
```

Here the argument has a *default value* of `LearningElixir.phrase`. When defining an argument, you can put `\\` after an argument name and then give a default value- in this case, `LearningElixir.phrase`

```
iex(1)> LearningElixir.recombine
        "boldly-going-where-no-man-has-gone-before"
iex(2)> LearningElixir.recombine "hello universe"
        "hello-universe"
```

> Note: We skipped the `import_file "basic_elixir.ex"` line this time for brevity's sake, and we will continue to skip it, but you should keep calling `import_file` when you make changes.

This is a decent pattern for simple situations like what we have here, but is suboptimal once the complexity of your function starts to grow (specifically, the number and complexity of arguments passed to the function).

Second Solution: Arity

Another way you could attempt to do this is to use the or (`||`) functionality.

```
def recombine(phrase) do
  phrase = phrase || LearningElixir.phrase

  phrase
  |> String.split
  |> Enum.join("-")
end
```

In many languages, if you don't give a default value for an argument the default will automatically be `nil`. However, this is not the case in Elixir. Here's what happens if you try that:

```
iex(1)> LearningElixir.recombine()
        ** (UndefinedFunctionError) function LearningElixir.recombine/0
        is undefined or private.
        Did you mean one of:

              * recombine/1

            LearningElixir.recombine()
```

It's saying that `LearningElixir.recombine/0` is undefined or private… what does that mean? Didn't we define `recombine`? And what's that `0` afterwards? The answer, as you may have guessed from this section header, is "arity".

"Arity" is a fancy word for the number of arguments a given function requires. So we've defined LearningElixir.recombine/1 (a version of `recombine` with 1 argument), but not LearningElixir.recombine/0 (a version of `recombine` with 0 arguments). This can be annoying when we're used to looser languages that

default to `nil` automatically, but it's really helpful for when we accidentally forget an argument- Elixir will help us catch that bug right at the start.

We can, of course, "cheat" the arity system by providing default values:

```
def recombine(phrase \\ nil) do
  phrase = phrase || LearningElixir.phrase

  phrase
  |> String.split
  |> Enum.join("-")
end
```

But the existence of the concept of arity, combined with Pattern Matching, opens up a better possibility.

Pattern Matching

Pattern Matching is a way to define a function multiple times and then run a specific definition based on the arguments given. One way to pattern match is by arity.

```
def recombine(phrase) do
  phrase
  |> String.split
  |> Enum.join("-")
end

def recombine do
  LearningElixir.phrase |> LearningElixir.recombine
end
```

Here we're defining two versions of recombine- the first with an arity of 1 and the second with an arity of 0. When we call it with an argument, we get the first version of `recombine`. When we call it without an argument, we get the second version- which then calls the first version and feeds it the phrase we defined earlier.

```
iex(1)> LearningElixir.recombine "hello world, universe"
        "hello-world,-universe"
iex(2)> LearningElixir.recombine
        "boldly-going-where-no-man-has-gone-before"
```

Pretty cool, right? There are lots of other ways you can employ pattern matching beyond simple number of arguments, and we'll go over those as they come up.

Exercises

1. Use pattern matching to add a 2-arity version of `recombine` which lets you input the join string.

```
iex(1)> LearningElixir.recombine("hello world, universe", " vast ")
        "hello vast world, vast universe"
iex(2)> LearningElixir.recombine("hello universe")
        "hello-universe"
```

2. Redefine the 1-arity version of `recombine` in terms of the 2-arity version.

That is, make it so that the 1-arity version calls the 2-arity version instead of repeating code. You'll note that we already defined the 0-arity version to call the 1-arity version.

Captain's Log: Function Ordering

Our scientists have arranged the above functions of different arities in many orders, and have found no difference in how the program runs.

However, in other situations the order does matter- Elixir will check the functions in order and use the first one that matches. So we have made it a habit to put the more specific cases earlier in the file and the more general cases later in the file.

Conclusion

Learning how to define functions with arguments (and using pattern matching to define multiple versions of a function) make our functions much more flexible and useful.

In the next chapter we'll introduce our first complex data type- Maps. Playing around with them will also lead us to the concept of Immutability- something very important in the functional programming world.

Maps

Maps

Maps (not to be confused with the `Enum.map` function) are a very common programming construct, although they might be called "hashes" or "dictionaries" in other languages (or even, confusingly, "objects" in Javascript).

Here's a basic map:

```
defmodule LearningElixir do
  def my_map do
    %{
      "name" => "Enterprise",
      "type" => "CodeShip",
      "mission" => "Code Boldly"
    }
  end
end
```

Captain's Log: PascalCase vs snake_case

At first we thought the way the names were constructed was mere happenstance, a whim of the creators, but we have since determined a consistent pattern.

Modules, such as LearningElixir, are PascalCase. That means that each word that makes up the name is capitalized, and runs together with the other words with no separation.

Functions and variable names, such as my_map, are snake_case. That means that the words are entirely lower-case, and are separated by underscores.

It's started with a `%{`, ended with a `}`, and in between consists of key-value pairs separated by commas. Each key-value pair has a key (such as "name") before the `=>` symbol (sometimes called the "rocket") and a value (such as "Enterprise") after the `=>` symbol.

We can interact with this map using the functions in the `Map` module.

```
iex(1)> Map.get(LearningElixir.my_map, "mission")
        "Code Boldly"
iex(2)> Map.get(LearningElixir.my_map, "bad_key")
        nil
```

Our first (and most common) Map function, `Map.get`, takes two arguments: the map (`LearningElixir.my_map`) and a key ("mission"). It will then grab the value attached to that key in the map. If the key given doesn't exist in the map, it will return `nil`.

You can use brackets as shorthand for `Map.get`.

```
iex(1)> LearningElixir.my_map["mission"]
"Code Boldly"
```

Another way to get an item from a map is to destructure it:

```
iex(1)> %{"mission" => mission} = LearningElixir.my_map
        %{mission: "Code Boldly", name: "Enterprise", type: "CodeShip"}
iex(2)> mission
        "Code Boldly"
```

More on destructuring in the next chapter.

The next most common Map function is to add new values to the map with `put`.

```
iex(1)> Map.put(LearningElixir.my_map, "captain", "Picard")
        %{"captain" => "Picard", "mission" => "Code Boldly",
          "name" => "Enterprise", "type" => "CodeShip"}
iex(2)> LearningElixir.my_map
        %{"mission" => "Code Boldly", "name" => "Enterprise",
        "type" => "CodeShip"}
```

When we use `put`, it uses the second and third arguments as the key and value respectively, and adds them to the map. The result is seen after the first line. However, you'll notice that when we call `LearningElixir.my_map` again, the new key-value isn't there.

Immutability

That's because data in Elixir is "immutable". That means that each piece of data will never change. What `put` does is create a *new* piece of data and then assigns it to a variable.

To understand what "immutable" means, we'll have to contrast it with data in a mutable language. Here's some Javascript:

```
> var mutableArray = ["Don't", "try", "this", "at", "home"];
  undefined
> mutableArray.shift()
  "Don't"
> mutableArray
  ["try", "this", "at", "home"]
```

> Don't worry if you don't understand the syntax of Javascript. The important thing is that you understand the differences between mutable and immutable languages, not the specifics of this example.

Above, we're assigning 5 strings to the variable `mutableArray`. Then we call the `shift` method on the `mutableArray`, which returns the first value in the array and has the *side effect* of removing that string from the array. The array now only has 4 elements.

Contrast that with immutable data, where there will never be side effects- no matter what you call on an array with 5 items, it will always be an array with 5 items.

Of course, the return value of the function can be something other than 5 items.

```
iex(1)> immutable_array = ["Immutability", "is", "great", "don't", "you", "agree"]
        ["Immutability", "is", "great", "don't", "you", "agree"]
iex(2)> Enum.slice(immutable_array, 4, 2)
        ["you", "agree"]
iex(3)> immutable_array
        ["Immutability", "is", "great", "don't", "you", "agree"]
```

So even though you call `Enum.slice` with `immutable_array` and get back an array with 2 items, `immutable_array` is still what it started as. There are no side effects.

Of course, there's a trick you can play with the data- take the result of the calculation and *immediately reassign it* to the variable you used.

```
iex(1)> immutable_array = ["Immutability", "is", "great", "don't", "you", "agree"]
        ["Immutability", "is", "great", "don't", "you", "agree"]
iex(2)> immutable_array = Enum.slice(immutable_array, 4, 2)
        ["you", "agree"]
iex(3)> immutable_array
        ["you", "agree"]
```

Oh no! Even naming it `immutable_array` didn't stop that disaster! How is this immutable data?

The array that `immutable_array` originally pointed to is still 5 items long. It's just that we told `immutable_array` to point to a new, different array- one that was the result of calling `Enum.slice` on the original `immutable_array`.

Imagine if you could start calling yourself Germany and then the entire country of Germany could no longer be found. Germany would still exist – you didn't mutate it – but everyone that was looking for Germany found you instead.

That's what it's like reassigning a variable.

One way around this is to keep on assigning stuff to new variables:

```
iex(1)> phrase = "boldly going where no man has gone before"
        "boldly going where no man has gone before"
iex(2)> phrase2 = String.split(phrase, " ")
        ["boldly", "going", "where", "no", "man", "has", "gone", "before"]
iex(3)> phrase3 = Enum.join(phrase2, "... ")
        "boldly... going... where... no... man... has... gone... before"
iex(4)> phrase
        "boldly going where no man has gone before"
iex(5)> phrase3
        "boldly... going... where... no... man... has... gone... before"
```

That method, however, can get tedious. That's one reason why the pipe (`|>`) construct is so popular in Elixir- it allows you to pass on the output of a function and use it as the first argument in the next function call, without the bother of naming it

```
iex(1)> phrase =
...(1)> "boldly going where no man has gone before" |>
...(1)> String.split(" ") |>
...(1)> Enum.join("... ")
        "boldly... going... where... no... man... has... gone... before"
iex(2)> phrase
        "boldly... going... where... no... man... has... gone... before"
```

Remember: in the command line we put the pipe at the end of the line, to let the Elixir interpreter know that we have more coming on the next line.

Now let's apply this to `Map.put`:

```
iex(1)> my_map = LearningElixir.my_map |>
...(1)> Map.put("captain", "Picard") |>
...(1)> Map.put("spock replacement", "Data")
        %{"captain" => "Picard", "mission" => "Code Boldly", "name" => "Enterprise",
          "spock replacement" => "Data", "type" => "CodeShip"}
iex(2)> my_map
        %{"captain" => "Picard", "mission" => "Code Boldly", "name" => "Enterprise",
          "spock replacement" => "Data", "type" => "CodeShip"}
```

We were able to feed the results of each line into the first argument of `Map.put`, then assign the whole thing to `my_map` without intermediate steps or explicit reassigns. This approach gives us a syntax as convenient as mutability, but with the stability and long-term simplicity of immutability.

Exercises

What will be the results from running each of the following functions?

```elixir
def one do
  map = %{"hello" => "universe"}
  Map.put(map, "discarded", "data")
  map
end

def two do
  map = %{"hello" => "universe"}
  Map.put(map, "information", "data")
end

def three do
  %{"hello" => "universe"}
  |> Map.put("exploration_style", "bold")
  |> Map.put("starship", "Enterprise")
end

def four do
  %{"hello" => "universe"}
  |> Map.put("exploration_style", "bold")
  |> Map.put("starship", "Enterprise")
  |> Map.get("hello")
end
```

Be sure to check your work by copying the functions into a file and then running them.

Conclusion

In this chapter we introduced our most complex data-type to date: Maps. We learned how to get data from a map, and then how to update the data on a map- through creating a new copy of the map with updated data, because data in Elixir is immutable.

In the next chapter we'll introduce two more data types- Atoms and Tuples. They'll let us expand our range with pattern matching capabilities significantly, especially when combined with the `case` statement.

Atoms, Tuples, and Case

In this chapter we'll cover three more basic Elixir concepts that are used often in Phoenix apps. We'll also combine them (and build on the concepts we've previously learned) in order to discover even more ways of pattern matching.

Atoms

Here's an example of an atom: `:name`.

Atoms are kind of like Strings, except that they can't be manipulated. `"Name"` could be fed into `String.upcase |> String.slice(1, 3)`, which would output `"AME"`. `:name` will always and forever be `:name`, unless it is first explicitly turned into a String. Hence the name: an atom is irreducible.

At first glance, Atoms could be dismissed as a less-capable replacement for Strings… but their power is in their limitations, both in the signaling value to humans (it will not be manipulated) and in the conveniences that it allows.

One popular use of atoms is as the key in a Map. Here's what `my_map` from the last section would look like with a simple replacement:

```
defmodule LearningElixir do
  def my_map do
    %{
      :name => "Enterprise",
      :type => "CodeShip",
      :mission => "Code Boldly"
    }
  end
end
```

Now when we want to access the code, we use an atom instead of a String:

```
iex(1)> LearningElixir.my_map[:name]
        "Enterprise"
iex(2)> LearningElixir.my_map["name"]
        nil
```

But here's the first convenience that atoms unlock:

```
iex(3)> LearningElixir.my_map.name
        "Enterprise"
```

This is much nicer than the clunky `["string"]` and `[:atom]` syntax.

You can also make your Map syntax nicer with atoms:

```
defmodule LearningElixir do
  def my_map do
    %{
      name: "Enterprise",
      type: "CodeShip",
      mission: "Code Boldly"
    }
  end
end
```

That looks much cleaner than what we had before, and works the exact same way when being called as the `:name => "Enterprise"` syntax.

So we've got a bit of convenience and syntax nicety- why else would we use atoms?

The primary benefit of atoms comes in their signaling value. A String can be manipulated- capitalized, split, reversed, etc. Elixir Strings are technically immutable (unchanging), but reassignment and piping make it easy for them to be manipulated without technically being mutated. On the other hand, if an atom starts as `:name`, it will go through the entire system as `:name` (unless explicitly turned into a String and then turned back into an atom).

Captain's log: changing an Atom

I have noticed that these Atoms, though supposedly indivisible, can be turned into a String or a Char List. This is, of course, a terrible idea, and should never be attempted. Not even for a handy plot point! Like the fourth wall, the integrity of atoms is very important to preserve.

Atoms are a perfect fit for keys in a Map construct. They're also great for pattern matching.

Pattern Matching with Atoms

```
defmodule LearningElixir do
  def my_map do
    %{
      name: "Enterprise",
      type: "CodeShip",
      mission: "Code Boldly"
    }
  end
```

```
  def my_map(:voyager) do
    %{
      name: "Voyager",
      type: "Intrepid",
      mission: "Make it back"
    }
  end

  def my_map(:ds9) do
    %{
      name: "Terok Nor",
      type: "Station",
      mission: "Protect Bajor"
    }
  end
end
```

Now we're matching not just on arity (number of arguments), but also on the value of the argument given (for versions with arity of 1).

```
iex(1)> LearningElixir.my_map
        %{mission: "Code Boldly", name: "Enterprise", type: "CodeShip"}
iex(2)> LearningElixir.my_map(:ds9)
        %{mission: "Protect Bajor", name: "Terok Nor", type: "Station"}
```

Pattern matching is common with atoms, but it could be done with any datatype- Strings, numbers, even Maps.

One of the other common pattern-matching data-types is a Tuple.

Tuples

Tuples are simply collections of values, surrounded by curly braces:

```
iex(1)> my_tuple = {"Babylon", 5}
        {"Babylon", 5}
```

As you can see, the values can be of any type- including multiple types within one tuple. They can also be of any length, although lengths of 2 to 4 are most common.

You can access an element within a tuple using the `elem` function:

```
iex(2)> elem(my_tuple, 0)
        "Babylon"
```

As you can see, it's zero-indexed.

You can also decompose a tuple into its component parts:

```
iex(3)> {name, num} = my_tuple
        {"Babylon", 5}
iex(4)> name
        "Babylon"
```

This is a form of pattern matching, where the intent is to destructure the tuple into its component parts (sometimes this specific technique is just called "destructuring").

This is similar to the Map destructuring we covered briefly in the last chapter. The biggest difference is that for a map you can destructure to as many or as few keys as a you like, but in a tuple you have to destructure to the whole tuple. However, there are some conveniences– if you don't need one part of it, you can just use the underscore symbol instead of coming up with a throwaway name:

```
iex(5)> {name, _} = my_tuple
        {"Babylon", 5}
iex(6)> name
        "Babylon"
```

We're essentially throwing away the number by using the underscore. This saves us the trouble of naming it, and also communicates to future readers of our code which parts of the tuple will be used.

Technobabble: Assignment and Pattern Matching

We've called `name = {"Babylon", 5}` an "assignment", but we called `{name, _} = {"Babylon", 5}` "pattern matching". What sets them apart?

The truth is- they are both pattern matching. They're both attempts to make the left side equal to the right side.

When the left side is just one variable, then it's very easy to "match" to it- almost anything will do. This is "assignment" in Elixir and most other languages.

When there's a tuple or on the left side the attempt is more complex, and the attempt could fail, but it's essentially doing the same thing. A successful "destructuring" is when a complex left side of the equation successfully pattern matches with the right side.

It's possible for a destructuring attempt to fail.

```
iex(1)> {name, 5} = {"Babylon", 5}
        {"Babylon", 5}
```

```
iex(2)> name
        "Babylon"
```

In this first example, we're telling Elixir that `"Babylon"` is the variable `name`, and the number `5` is the number `5`. Elixir agrees!

This second example is where things go wrong.

```
iex(1)> {name, "five"} = {"Babylon", 5}
        ** (MatchError) no match of right hand side value: {"Babylon", 5}
            (stdlib) erl_eval.erl:450: :erl_eval.expr/5
            (iex) lib/iex/evaluator.ex:249: IEx.Evaluator.handle_eval/5
            (iex) lib/iex/evaluator.ex:229: IEx.Evaluator.do_eval/3
            (iex) lib/iex/evaluator.ex:207: IEx.Evaluator.eval/3
            (iex) lib/iex/evaluator.ex:94: IEx.Evaluator.loop/1
            (iex) lib/iex/evaluator.ex:24: IEx.Evaluator.init/4
iex(2)> name
        ** (CompileError) iex:2: undefined function name/0
```

We're saying that `"Babylon"` is the variable `name`, and Elixir still agrees. But then we tell Elixir that the number `5` is the string `"five"`, which is false. Therefore Elixir throws an error.

You can see that if one part of the destructuring fails, they all fail. `name` isn't assigned.

Tuples in Function Definitions

Tuples can also be used to pattern match in function definitions:

```
defmodule LearningElixir do
  def take_action({:ok, _}, ship) do
    "Great job, #{ship}"
  end

  def take_action({:error, error_message}, ship) do
    "Problem with #{ship}.  #{error_message}"
  end
end
```

Here both variations on the function have two arguments, the second of which is a ship and the first of which is a tuple. The tuple has two values- an atom and a string. The atom is either `:ok` or `:error`, and the string is used when the action is a failure but thrown away when the action is successful.

```
iex(1)> LearningElixir.take_action({:ok, "Make it so"}, "Enterprise")
        "Great job, Enterprise"
```

```
iex(2)> LearningElixir.take_action({:error, "Shields are at 38 percent!"}, "Enterprise")
        "Problem with Enterprise.  Shields are at 38 percent!"
```

Matching a tuple with an `:ok` or `:error` atom as the first value is very common in Phoenix apps.

Conditionals with case

There are many other ways we could go about coding `take_action`. Probably the most fitting is the `case` statement.

Here's a super simple example of a case statement:

```
def rabbit_counting(number) do
  case number do
    0 -> "none"
    1 -> "one"
    2 -> "two"
    3 -> "three"
    4 -> "four"
    _ -> "many"
  end
end
```

A case statement has two parts- the expression and the clauses. In this case statement, the expression is `number`, and there are six clauses.

Each clause has a head and a body. The head (such as `1` or `_`) is before the arrow `->`, and is what matches against the expression. The body is what's after the arrow `->`. When a head matches, the body of that clause is returned.

So if you pass in `1`, you get `"one"` returned. If you pass in `4`, you get `"four"` returned.

What happens if you pass in `5`? The underscore (`_`) in the head acts as a default, accepting anything. So if we pass in `5`, it will be caught by the underscore and return `"many"`. Be careful about that, though- if you put the underscore as the first clause, then it will catch *everything*, not letting any of the other more specific clauses match.

Now let's apply the case statement to our tuple pattern-matching example:

```
defmodule LearningElixir do
  def take_action(tup, ship) do
    case tup do
      {:ok, _} -> "Great job, #{ship}"
      {:error, error_message} -> "Problem with #{ship}.  #{error_message}"
    end
```

```
  end
end
```

If `{:ok, _}` matches, then `"Great job, #{ship}"` will be returned, with `ship` being filled in with whatever was passed in to the function.

Case statements of this form are very common in Phoenix apps, so you'll have plenty of time to get used to how they work.

Other Conditionals

There are two other common way of doing conditionals in Elixir: `cond` and `if`. However, they aren't used nearly as often as `case` within Phoenix, so if we use them then we'll introduce them at that time.

This may be a bit of a shock to people from other languages who are used to using `if` for everything, but in Elixir `if` is used sparingly. Most of the common usages of `if` can be done better by some sort of functional conditional- usually, but not always, involving pattern matching.

Exercises

1. For the following code:

```
{:ok, phaser_setting, _} = {:ok, "stun", "thank you"}
%{name: my_name} = %{rank: "Captain", name: "Picard"}
```

What is the value of `phaser_setting` and `my_name`?

   Hint: Rememmber the Map destructuring from the previous chapter

2. Type out our final version of the `take_action` function. What happens when you call it as follows? Why?

a. `LearningElixir.take_action({:err_bear, "Doctor, why is tummy glowing? This seems serious."}, "Enterprise")`
b. `LearningElixir.take_action("no tuple here", "Enterprise")`
c. `LearningElixir.take_action({"ok", "I have made it so"}, "Enterprise")`
d. `LearningElixir.take_action({:ok, "I have made it so", "another part of the tuple, how fun"}, "Enterprise")`
e. `LearningElixir.take_action({:ok, "I have made it so"})`
f. `LearningElixir.take_action({:ok, "I have made it so"}, "Enterprise")`

Learning the common error modes is important- better that you do it now while your program is small.

3. Modify `take_action` so that the ship is passed as a third part of the tuple.

```
iex(1)> LearningElixir.take_action({:ok, "I have made it so", "Enterprise"})
        "Great job, Enterprise"
iex(2)> LearningElixir.take_action({:error, "Phasers not set to stun", "Enterprise"})
        "Problem with Enterprise.  Phasers not set to stun"
```

Conclusion

Now, with the introduction of atoms, tuples, and condition statements, we're finally starting to see functions that might look at home in a Phoenix app. In fact, our example for the case statement was inspired by the auto-generated Controller in Phoenix 1.2 and before (they changed the generator in Phoenix 1.3, but it still depends on the concepts we've introduced in this chapter).

Great job! You've already learned quite bit!

End of Part 1

That's most of the Elixir you need to know in order to understand a basic Phoenix app.

There's plenty of Elixir that I've left out, including some stuff that's used within Phoenix. I've left out several commonly-used things like `Enum.map` and anonymous functions because we're going to get pretty far into our app before we start needing them. And of course there's all the advanced topics- Channels, Metaprogramming, OTP, etc.- because while they're really cool, they have no place at the start of a beginner's guide. Finally, I've left out some of the more project-focused stuff- Import, Use, configuration files, various mix commands- because those are better understood while in the context of a full Phoenix app.

What that means is that our crash course in Elixir is finished- it's time to get started building a Phoenix app!

Captain's Log: more Elixir resources.

I have heard tell of great stores of knowledge, locked away in interconnected datapads and inscribed upon dead lumber. Here are the rumors which I hear most strongly:

- Try Elixir, Mixing it Up with Elixir- Interactive beginner's courses at CodeSchool. Try Elixir is free.

- Programming Elixir 1.3- An excellent book for programming language nerds and others who want to geek out on details of the Elixir language.

Were I a younger captain I might go seeking after these sources of knowledge myself, but I am lucky to have discovered even this one document, and in my age I know that deciphering it is much more important than some fool's errand. We must know how the Phoenix-class ships were constructed!
