Extracted from:

Programmer Passport: Elixir

The Pragmatic Bookshelf

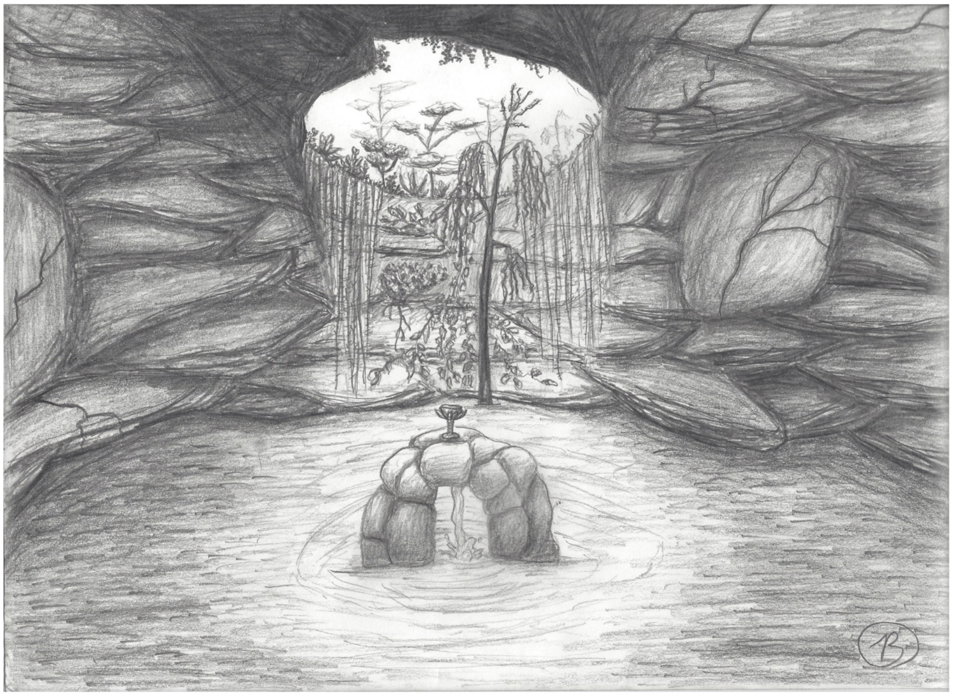
Raleigh, North Carolina

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

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.