Pattern Matching Course

ThinkingElixir.com

Download Reference

by Mark Ericksen

This document is intended as a reference resource after course completion.

For the best learning experience and access to the accompanying download materials, enroll in the course.

Install Elixir

In order to start *learning* Elixir, you need the ability to *play* with Elixir. Please make sure you have Elixir installed on your system before you continue.

Read here for the Official documentation for installing Elixir.

Follow my guide to installing Elixir using asdf-vm .

Asdf is a version manager tool on **MacOS** and **Linux** that installs and manages multiple versions of Elixir and Erlang. I recommend this to ensure you have compatible versions of Elixir and Erlang to give you the best development experience.

With Elixir installed your machine, from a command terminal, you should be able to execute the `-v` or "version" command to see that it is ready to go.

```
$ elixir -v
Erlang/OTP 22 [erts-10.5.5] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Elixir 1.9.4 (compiled with Erlang/OTP 22)
```

The exact version isn't important. Your's will likely be newer than this one which is awesome!

Need a code editor?

Do you need an editor to use for writing your new Elixir code? If you don't already have an editor in mind, check out this post to get you started!

Elixir's Interactive Shell

Elixir has an interactive shell called IEx. This is a powerful and very helpful tool not only during the learning process, but when working with both development and live systems.

IEx allows to you write Elixir statements, execute them, and get the results. It has other great features like auto-completing commands (using the `TAB` key), displaying help and type information.

We Learn By Doing

The most effective way to *learn* Elixir is to start *doing* Elixir. The IEx shell gives you an easy on-ramp to getting started. With Elixir installed on your system, you are ready to go.

In the coming sections, I really encourage you to *play* with the code as you read and learn about it. The code examples are designed to be easy to copy & paste into a terminal to make it that much easier.

Open a terminal on your computer. The command to start IEx is... `iex` .

```
$ iex
Erlang/OTP 21 [erts-10.0.6] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Before we start to play, you need to know how to get out of your new play area. To exit IEx, hit `CTRL+C` and you'll see the BREAK menu:

```
iex(1)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

If you press `c` , you will remain in IEx. If you press `a` , it will abort the shell and exit. The most common way to exit is to use `CTRL+C` again. Start an IEx shell again, now hit `CTRL+C` , `CTRL+C` . That's right, two times in a row. That's the easiest way to exit an IEx shell. Commands like `quit` and `exit` don't exist.

Play Time!

Now that you know how to start and exit an IEx shell, it's time to start playing. Here are some simple commands you can try.

```
iex(1)> 1 + 1
2
iex(2)> b = 12
12
iex(3)> b
12
iex(4)> b + 10
22
iex(5)>
```

Notice that the `iex` prompt includes a line number. It increments automatically. You don't have to worry about that.

Line Continuations

When you write an expression and use an opening character that requires a closing one, it can carry across lines.

```
iex(1)> "abcd
...(1)> efg"
"abcd\nefg"
```

Notice that it included the `\n` new-line character inside the string. Also note that the `...(1)>` was the continuation line.

Sometimes this happens by accident. You accidentally forget to include a matching closing character. In this example, I "forgot" to include the closing parenthesis. Eventually I figure it out, add the closing parenthesis and the function is evaluated.

```
iex(1)> String.downcase("HELLO WORLD"
...(1)>
...(1)>
...(1)> )
"hello world"
```

Auto-Complete in IEx

As mentioned, auto-complete is very handy in IEx. Time to play with it. Let's explore that `String` module that was just used to downcase some text. What else can it do? To find out, type `Str` and hit your `TAB` key. You should see something like this:

```
iex(2)> Str
Stream      String      StringIO
```

It shows modules that match the `str` prefix. Add an `i` and it knows you want "String" over "Stream". Hit `TAB` and let it complete. Now add a `.` and hit `TAB` again to see all the modules and functions available under the `String` module. It should look something like this:

```
iex(2)> String.
Break                 Casing                Chars
Tokenizer             Unicode               at/2
bag_distance/2        capitalize/1          capitalize/2
chunk/2               codepoints/1          contains?/2
downcase/1            downcase/2            duplicate/2
ends_with?/2          equivalent?/2         first/1
graphemes/1           jaro_distance/2       last/1
length/1              match?/2              myers_difference/2
next_codepoint/1      next_grapheme/1       next_grapheme_size/1
normalize/2           pad_leading/2         pad_leading/3
pad_trailing/2        pad_trailing/3        printable?/1
printable?/2          replace/3             replace/4
replace_leading/3     replace_prefix/3      replace_suffix/3
replace_trailing/3    reverse/1             slice/2
slice/3               split/1               split/2
split/3               split_at/2            splitter/2
splitter/3            starts_with?/2        to_atom/1
to_charlist/1         to_existing_atom/1    to_float/1
to_integer/1          to_integer/2          trim/1
trim/2                trim_leading/1        trim_leading/2
trim_trailing/1       trim_trailing/2       upcase/1
upcase/2              valid?/1
iex(2)> String.
```

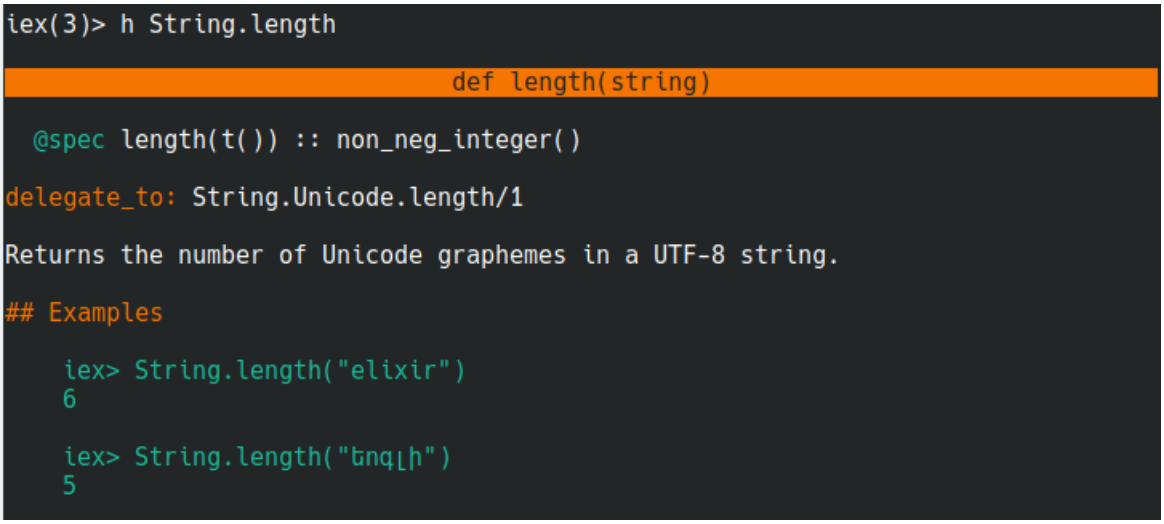
I see “length/1”. That seems obvious enough. I expect it will return the length of a string. Let’s try it.

```
iex(2)> String.length("elixir")
6
```

It returned the number of characters in the string. Note: the `/1` means the function takes 1 argument.

Help in IEx

As mentioned before, IEx has the ability to get “help” on modules, functions and more. Let’s see the help for `String.length`. The help is displayed by putting the `h` command in front of a module name or a module and function name together as in the following example:



You will notice that this help text is the exact same as what is in the online documentation. String.length/1 documentation. This is no coincidence. The documentation is generated from the help text included in the code. Likewise, your code always has the documentation for the exact version of Elixir that you are running. That’s pretty cool! Still works when you are on an airplane.

Command History

Being able to press the `UP` arrow key and bring back the last command can be really helpful. When you start IEx, you can pass in a command to make that available like this:

```
$ iex --erl "-kernel shell_history enabled"
```

This command is passing an Erlang configuration option through IEx using `--erl` . The command will enable shell history. Now, when in an IEx shell, pressing the `UP` arrow brings back the previous command. Hitting the `UP` arrow repeatedly lets you keep going back through the command history.

Entering that command every time you start IEx isn't fun. You can set the `ERL_AFLAGS` environment variable on your system through your shell's profile file to make it always available.

On Unix-like / Bash: (ie: MacOS/Linux)

```
export ERL_AFLAGS="-kernel shell_history enabled"
```

On Windows:

```
export ERL_AFLAGS="-kernel shell_history enabled"
```

On Windows 10 / PowerShell:

```
$env:ERL_AFLAGS = "-kernel shell_history enabled"
```

Don't worry about doing this right now if it's not important to you at the moment. Just know that it can be solved and you have the solution right here when you are ready for it.

Ready, Set, Go!

With Elixir installed and the ability to play in the interactive IEx shell, you are ready to go! Remember that *play* is fun. So now it's time to have fun!

Basic Types Overview

There aren't that many basic data types in Elixir. These are the ones we'll cover here.

- atom
- boolean
- nil
- integer
- float
- string

Thinking Tip: Learn by doing

Remember, we learn best by *doing*. Open an IEx shell and play with each of these types as we cover them.

Atom

An atom is a literal, a constant with name. It is a constant whose name is its value. Atoms are very useful in pattern matching and are used for more than you might expect in Elixir.

Examples of Atoms:

```
:my_atom
:testing
:customer_type
```

An atom begins with a colon and typically is all lowercase letters using an underscore to separate words.

If you need an atom to contain spaces or special characters that otherwise wouldn't be valid, you can express it as a string with the colon in front.

```
:"I'm still an atom"
```

However, writing atoms this way is unusual and only done when needed for specific exceptional instances.

An Atom in Elixir is similar to a *symbol* in Ruby or an *enumeration* in C/C++. Atoms are stored in the "atom table". All references to an atom like " `:ok` " are shared and actually all point to the same atom table entry.

Functions for working with an Atom can be found in the Atom module.

Preventing Denial-of-Service

Atoms are not garbage collected. The important thing to learn from this is you should not allow user provided content to become an atom at run-time in your system. Converting unchecked user data to an atom can expose your system to a Denial of Service (DOS) attack. The attack works like this, a malicious actor causes your system to repeatedly create unique atoms until it consumes all of the available resources on your machine ultimately causing it to crash. It isn't a security flaw, it isn't a "break-in", but it can be abused to cause your system to crash.

You *can* allow user input to become an atom using the `String.to_existing_atom/1` . If you already have an atom defined in the system, it accepts the conversion. If you try to convert to something new, it raises an exception blocking the operation.

```
iex(1)> String.to_existing_atom("ok")
:ok

iex(2)> String.to_existing_atom("whaaaat")
** (ArgumentError) argument error
    :erlang.binary_to_existing_atom("whaaaat", :utf8)
```

Boolean

Examples are: `true` and `false` .

The boolean values are actually implemented as special reserved atoms.

```
false == :false
#=> true

true == :true
#=> true
```

Nil

Nil represents the absence of a value. It is like `null` in Javascript and many other languages.

Interestingly, `nil` is also implemented as a reserved atom.

```
nil == :nil
#=> true
```

In evaluations, `nil` behaves like `false` .

```
if nil || true, do: "True!", else: "False."
#=> "True!"
```

Integer

Represents positive and negative whole numbers. Including 0.

Functions for working with an Integer can be found in the Integer module.

Examples:

```
-10, 0, 1000
```

Integers can be arbitrarily large. It uses as much data is needed to contain it.

```
99999999999999999999999999999999999999999999999999999999
```

Can be represent as Hex and more.

```
0xFF
#=> 255
```

Integers can use _ as a grouping separator. This is functionally ignored but helps with readability.

```
1_000_000
#=> 1000000
```

Float

Represents floating point numbers.

Examples:

```
-12.4, 0.5, 100.67, 83.33333333333
```

Functions for working with a Float can be found in the Float module.

Floats don't support a "Decimal" type for explicit levels of precision. As such, it results in typical floating point rounding issues.

```
0.8 * 3
#=> 2.4000000000000004
```

This is expected with floating point math. For further explanation, see these resources:

- http://0.30000000000000004.com
- What Every Programmer Should Know About Floating-Point Arithmetic

If you need to represent fixed decimal values for something like money, using a Float may not be the most appropriate choice. For example, it may work better to use an integer that represents cents (instead of whole dollars).

Another option is to use a community package called Decimal that provides arbitrary decimal precision.

Scientific Notation

Floats are frequently displayed in their scientific notation. This may not be what you expect or want.

```
1200.0
#=> 1.2e3
```

Note that when working in IEx, it converts the data to a string for display in the shell. This is the same display as Float.to_string/1.

```
Float.to_string(1200.0)
#=> "1.2e3"
```

If you need to convert a float to a string with explicit decimal precision, use the built-in Erlang function `float_to_binary`. In Elixir, you can use any Erlang function. In this example, the function we want is declared in the `erlang` module. To call it, use an atom as the module name like this:

```
:erlang.float_to_binary(1200.0, decimals: 2)
#=> "1200.00"
```

String

Strings are encoded in UTF-8. A string uses the double quote character `"`. Single quoted text is not a string, that's a charlist and behaves differently.

```
"The quick brown fox"
```

The String module has a number of functions for working with them.

```
String.upcase("hello world")
#=> "HELLO WORLD"
```

Concatenation

Elixir strings can be concatenated using the `<>` operator.

```
"one" <> " two"
#=> "one two"

text = "Hey"
text <> " friends!"
#=> "Hey friends!"
```

Interpolation

Elixir strings support interpolation using the `#{...}` characters embedded in a string.

```
name = "Tom"
"Greetings #{name}!"
#=> "Greetings Tom!"
```

Strings are Binary

UTF-8 strings didn't exist in Erlang before Elixir. Because of this, many Erlang functions that take a string don't take an Elixir string. Erlang sees an Elixir string as a `binary` type.

```
is_binary("a string")
#=> true
```

In Erlang, functions often expect a `charlist`.

Charlist

Mostly used for interoperability with Erlang. It is just a list of code points and is created with single quotes.

```
'this is a charlist'
#=> 'this is a charlist'

is_list('testing')
#=> true
```

A charlist can be converted to a string and vise versa using these functions:

- Kernel.to_string/1
- Kernel.to_charlist/1

"Kernel" is Elixir's default environment. It even defines basic math operators like `+` and `-`. Because it is so essential to Elixir, the Kernel module is always included for you. You can just use the functions `to_string/1` and `to_charlist/1` to convert between the types without needing to specify the "Kernel" module explicitly.

```
to_string('hello world')
#=> "hello world"

to_charlist("hello world")
#=> 'hello world'
```

Modules to Manipulate

All of these types are primitives. They are "data". They are not objects with functions attached to them. There are no "objects" in Elixir. There is only "data" and "functions". Modules are a collection or a container for functions.

By convention, when you want to perform some function on a piece of data, you use the type's module for doing that. This is particularly the case for Atom, Integer, Float, and String.

List

How it works

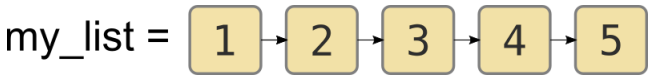
A List uses the `[` and `]` characters to contain the elements of the list separated by commas. It looks like this:

```
my_list = [1, 2, 3, 4, 5]
```

This looks like an Array in other languages. However, a List doesn't *act* like an Array.

A list in Elixir (and other functional programming languages) is actually a recursive data structure. Internally implemented as a linked list.

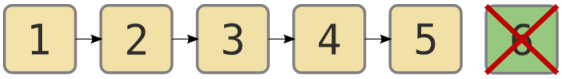
When you realize an Elixir list is an immutable linked list, it's behavior makes sense and how you use it becomes clear.



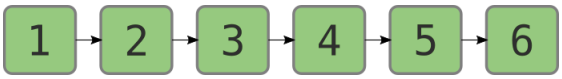
`my_list` is "bound" to the list. The variable isn't "assigned" the value of the list, it is said to be "bound". Variables in Elixir are all "bound" to their values. You can think of "bound" as meaning "is pointing to". So `my_list` is pointing to a specific element in a linked list. Each element points to the next element in the list and separately points to the *value* for the element.

> When you realize an Elixir list is an immutable linked list, it's behavior makes sense and how you use it becomes clear.

In many languages, it is common to add more items to the end of an Array as it is built up. With an immutable List, we can't add items to the end as that would be affecting other variables that are bound to some element earlier in the list.

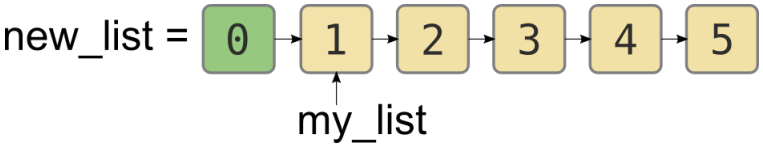


If I really do want to add to the *end* of the list, I can, but a whole new list is created with that value at the end. When a list is very large, this becomes an expensive operation. Two lists can be concatenated together using the `++` operator.




```
my_list ++ [6]
#=> [1, 2, 3, 4, 5, 6]
```

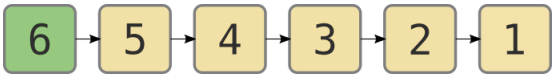
Adding to the *front* of the List, or the "head" is very cheap. It doesn't alter other variables that are already bound to some element in the List.



This makes it very efficient both in speed and memory consumption. To efficiently add to the *front* of a list, you use the `|` pipe character to join the new element and the existing list together.

```
[0 | my_list]
#=> [0, 1, 2, 3, 4, 5]
```

It can be cheaper and faster to build up a large list in reverse, adding to the "head" rather than re-building the whole list with each new addition. Then, once built, perform a single "reverse" of the whole list using Enum.reverse/1.




```
Enum.reverse([6, 5, 4, 3, 2, 1])
#=> [1, 2, 3, 4, 5, 6]
```

Thinking Tip: Immutable Data Insight

Thinking about a list as a “linked list” data structure where it’s made up of pointers to the *actual* data helps to understand how immutable data can work and be so memory efficient. The same kind of thing is happening with the other data structures, but I think it’s easiest to start imagining it with a list.

Working with immutable data may be a different experience for you. It probably feels uncomfortable. However, it is what enables concurrency and immutable data removes a whole category of state related bugs. If it is uncomfortable now, just know that understanding and embracing it will help make you a better programmer. And you’ll end up loving it!

See for yourself!

I described how adding to the *front* of a list is faster than adding to the *end*. The best way to really “get” this is to see and feel it for yourself.

Below I have 2 one-liner functions that exaggerate the differences so you can really feel it. Don’t worry about understanding the details of the code. Conceptually they both build a list of 100,000 items. They both add one number at a time to the list starting with the number 1 and going up to 100,000.

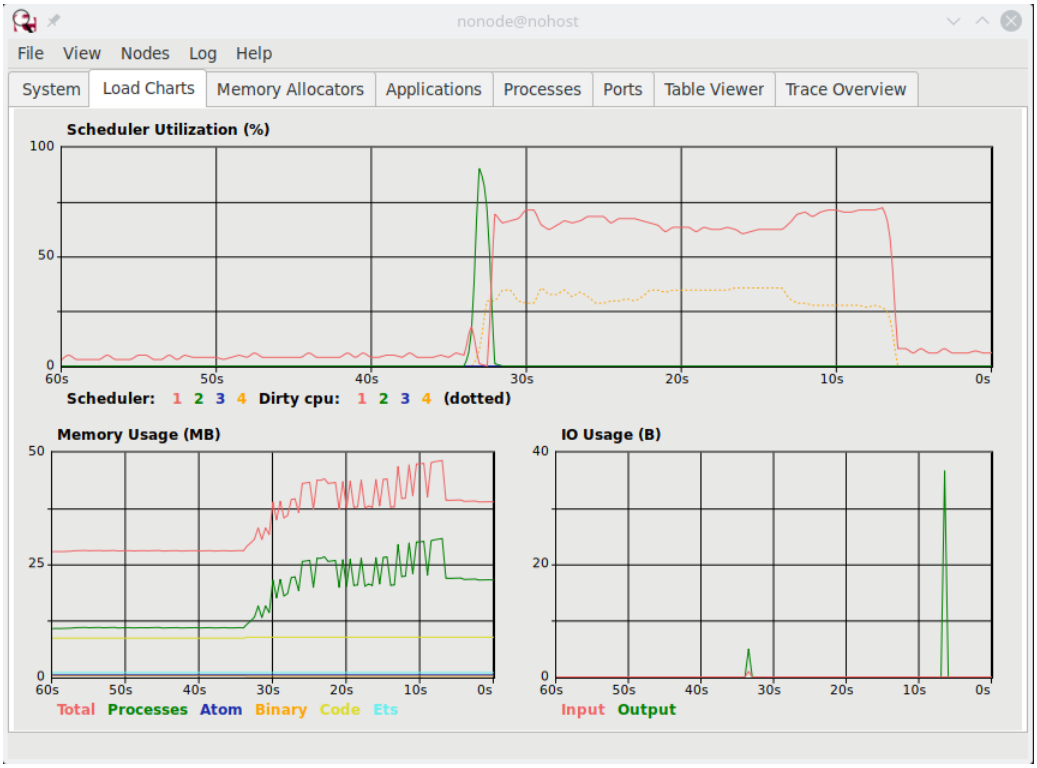
The difference is one function adds the item to the *end* of the list using list concatenation. That’s the code that says `acc + [n]`. It means `list_so_far ++ [next_number]`.

Run this line in IEx and expect it to take some time.

```
Enum.reduce(1..100_000, [], fn(n, acc) -> acc ++ [n] end)
```

Did you see how long that took?

Running Observer (it’s like an activity monitor built in to the BEAM), you can see the load it put on the machine.



Observer showing system impact of 100,000 inefficient list operations.

The top chart shows the CPU impact and how much time it took. The bottom left chart shows the RAM impact. The saw-tooth line is from garbage collection being done.

Why is it so inefficient? With every item added, it re-creates the *entire list* so no other references to the list are affected.

Let's try it again with code adds items to the *front* of the list. The important part of the code is the `[n | acc]` which means `[next_number | list_so_far]`. It adds the single item to the front of the list.

Run this line in IEx and see how long it takes to return.

```
Enum.reduce(1..100_000, [], fn(n, acc) -> [n | acc] end)
```

Did that feel different? Let's see the system impact of the code.

Notice the chart scale is reduced and that the CPU has one spike that stayed below 30%. The RAM also only slightly bumped.

Why is it so much more efficient? As the list is being built, it never gets re-created. Any other variable references to the list made during this operation won't be affected by adding more items to the front. So it can just keep adding items to the front efficiently.

Should I only ever add to the front?

You might see this and ask, "should I only ever add items to the front of a list?" The answer is "no". This example is specially designed to exaggerate the behavior. When building a list of 10, 100, or even 1,000 items, adding to the end of the list probably won't even be noticed. However, it is important when working with much larger lists.

The point here is to understand how lists work in Elixir so you have the right mental model. Lists are internally implemented as linked list, not an array. You can't treat them like an array.

List Contents

Lists don't just contain integers. Lists can contain any data type and can contain different types from one element to the next.

A list's contents are ordered. They remain in the order they are declared.

```
[1, "Hello", 42.0, true, nil]
```

Experiment with Lists

Take some time to experiment with lists in IEx. Try concatenating lists together with `++`, you can also *remove* elements from a list using `--`. Try adding to the front of a list using the `|` pipe separator. Try tweaking the list function examples to use different numbers. Experimentation is playing! Have fun playing!

Tuple

Here are a few key points about the tuple data type.

- A tuple is a collection type. It is a structure that contains a *fixed number* of things.
- A tuple's elements are *ordered* and *fixed* in size.
- A tuple does **NOT** mean **TWO**. Tuples can have many elements. However, it is common to see a tuple with only two elements.
- A common usage in Elixir is to return multiple result values from a function in a tuple.

The following example represent common function return values. Each tuple contains 2 pieces of information or data.

```
{:ok, result}
{:error, reason}
```

The first piece of information is an atom (ie: `:ok` or `:error`) that tells if the operation succeeded or not. If we got an `:ok`, we know it succeeded and the next piece of data is the result of the operation. If we got the atom `:error` then we know the operation failed and the `reason` may tell us why.

Another example of using a tuple to return multiple pieces of information at once might be the result of a function that splits a list of integers into odd and even sets. Imagine that a function named `split_odd_even/1` is available to us. This function leaves the original list unmodified (immutable) and returns the two different result sets through a tuple. It could look like this:

```
{odd_results, even_results} = split_odd_even([1, 2, 7, 12, 15])

odd_results
#=> [1, 7, 15]

even_results
#=> [2, 12]
```

Here is another tuple example that contains multiple pieces of data. In this example the tuple contains a person's name and age:

```
{"Howard", 32}
```

Using the Kernel.elem/2 function, you can extract the value of an element from a tuple.

```
tuple = {:foo, :bar, 3}
elem(tuple, 1)
#=> :bar

elem({:foo, :bar}, 2)
#=> ** (ArgumentError) argument error
```

A cleaner and more natural way to do this is with "pattern matching" which isn't covered here.

You can replace an element using Kernel.put_elem/3. Remember, Elixir isn't changing the tuple, internally it is creating a new tuple where the elements point back to the previous tuple's values, with the exception of the new change.

```
tuple = {:foo, :bar, 3}
put_elem(tuple, 2, :baz)
#=> {:baz, :bar, :baz}
```

You can see that the original `tuple` variable has not been modified:

```
tuple
#=> {:foo, :bar, 3}
```

Tuples are best for a fixed number of elements. If you need a *dynamic* container that preserves order, use a `List` instead.

Map

A map is a collection data structure. An entry is made up of a `key` and a `value`. As noted in the Map documentation:

> Maps are the “go to” key-value data structure in Elixir.
>
> https://hexdocs.pm/elixir/Map.html

The “key” of a map can be anything. However, it is most commonly a string or an atom.

When you receive a web POST with data, it will be represented as a Map with string keys. When you prepare a response structure for JSON, it will likely end up as Map with atom keys.

Let’s look at each.

Make sure to try these things out in an IEx session.

String-Key Version

```
%{"name" => "Howard", "age" => 30}
#=> %{"age" => 30, "name" => "Howard"}
```

You may notice that the keys do not preserve order.

Atom-Key Version

```
%{name: "Howard", age: 30}
#=> %{age: 30, name: "Howard"}
```

An atom key can also be written using the arrow syntax. It is shortened for you.

```
%{:name => "Howard", :age => 30}
#=> %{age: 30, name: "Howard"}
```

Other Key Types

A map can have *any* data structure as a key and a value.

The key can be *any* data type.

```
%{
  1 => "Integer One",
  1.5 => "Float!",
  true => "boolean key",
  {:name, "Daniel"} => "a tuple",
  %{map_as_key: true} => "gracious!",
  [1, 2, 3] => "a list"
}
```

Using *any* data type as a key may not be very practical, but it can come in handy. Just know that this ability exists.

Again, **the most common keys are atoms and strings**.

Nested Values

It is common to have nested values. Take this for example:

```
order = %{
  id: 1,
  customer: %{id: 99, name: "Widgets, Inc."},
  item: %{
    id: "item-11332",
    name: "Sprocket #12",
    price: 12.70,
    quantity: 1
  },
  discounts: [%{code: "SUMMER19"}],
  total: 10.00
}
```

The `customer`, `item`, and `discounts` keys are atoms. The value is another data structure.

Accessing Values in a Map

There are a **lot** of ways to access the values and structure of a map. Here we'll introduce only a few of the basics.

Map.get/3

We can access elements of the map using the `Map.get/3` function. We provide it the map and the key we want the value for. The third argument is the default value to return if the key isn't found in the map. It defaults to `nil`.

```
Map.get(order, :id)
#=> 1
```

Requesting a missing key from the map, we can give the default value we want back.

```
Map.get(order, :missing_key)
#=> nil

Map.get(order, :missing_key, "Hey! It's missing!")
#=> "Hey! It's missing!"
```

Access Behaviour

Another way to access the value of a key is to use the access operator `[]`.

```
order[:id]
#=> 1

order[:missing_key]
#=> nil
```

When the key is an atom, it supports accessing it directly like this:

```
order.id
#=> 1
```

This will now cause an error if trying to access a missing key.

```
order.missing_key
#=> ** (KeyError) key :missing_key not found in: ...
```

Kernel.get_in/2

Another option is Kernel.get_in/2. It supports a handy way of returning data nested deeper in the structure.

```
get_in(order, [:customer, :name])
#=> "Widgets, Inc."
```

Changing an Immutable Map

Data structures in Elixir are **immutable**. Changing an element in a map returns a new map with the desired change. Internally, it is managed by pointers where everything points to the values of the old data structure with the exception of the latest change.

You don't need to worry about the internals though. You can just know that it is efficient memory management and it ensures "immutable" data.

There are several ways to alter a map.

Map.put/3

The Map.put/3 function can add *new* keys to a map and change *existing* ones.

```
person = %{
  name: "Sally Green",
  age: 35,
  position: "Manager",
  division: "Engineering"
}
```

Say we want to add another key to track on the `person`. Now we want to track the corporate region she works in.

```
Map.put(person, :region, "west-1")
#=> %{
#=>   age: 35,
#=>   division: "Engineering",
#=>   name: "Sally Green",
#=>   position: "Manager",
#=>   region: "west-1"
#=> }
```

If we look at the `person` again, we see it is unchanged! It doesn't contain the newly added region.

```
person
#=> %{
#=>   age: 35,
#=>   division: "Engineering",
#=>   name: "Sally Green",
#=>   position: "Manager"
#=> }
```

We didn't modify the original `person` map. It added the key to a new map where the new map's other keys all pointed back to the original `person` map.

If we want to *keep* the change, we can **re-bind** the `person` variable to point to the newly updated map. Alternatively we can create a new variable to bind to the altered map.

```
person = Map.put(person, :region, "west-1")
#=> %{
#=>   age: 35,
#=>   division: "Engineering",
#=>   name: "Sally Green",
#=>   position: "Manager",
#=>   region: "west-1"
#=> }
```

Now the `person` variable points to the updated map.

```
person
#=> %{
#=>   age: 35,
#=>   division: "Engineering",
#=>   name: "Sally Green",
#=>   position: "Manager",
#=>   region: "west-1"
#=> }
```

Thinking Tip: Re-Binding

In Elixir, we don't call it "variable assignment" because we aren't "assigning" a value to a variable. We are "binding" a variable to a value. When you think of how it's working internally with pointers, this makes more sense.

Elixir allows us to **re-bind** a variable. Erlang and other functional programming languages don't even allow that. Instead, I'd have to create a new variable for every change. Elixir makes this easier and it can deceptively *feel* like how it works in other languages.

Kernel.put_in/3

The Kernel.put_in/3 function lets us add new keys and update existing ones in a *deeply nested map*.

Updating a value deeper inside a nested map can be hairy when using `Map.put/3` because you have to update the map at each level. Instead `Kernel.put_in/3` gives us the syntax we need to make a deeper change.

Remember that we don't need to use the module `Kernel` as that is imported for us.

In this example our data is nested 3 levels deep and we want to update the `:value` at the deepest level.

```
data = %{
  name: "level 1",
  value: 100,
  data_1: %{
    name: "level 2",
    value: 200,
    data_2: %{
      name: "level 3",
      value: 300
    }
  }
}
```

Using `Kernel.put_in/3` works well. The second argument is a list of the keys to follow that get us where we want to go. The last argument is the new value to set for the last key in our list.

```
put_in(data, [:data_1, :data_2, :value], 3000)
#=> %{
#=>   data_1: %{
#=>     data_2: %{name: "level 3", value: 3000},
#=>     name: "level 2",
#=>     value: 200
#=>   },
#=>   name: "level 1",
#=>   value: 100
#=> }
```

This updated the value at the deepest level and returned a *new map* with the desired change. Data in Elixir is immutable. Our original `data` variable is still bound to the original map with the unchanged value.

Working with immutable data takes some time to get used to, but it is an important foundation for so many benefits that we want! The benefits are worth the initial discomfort we feel with it.

Special Update Syntax

Maps support a special update syntax that can update *existing* atom keys only.

```
%{person | position: "VP of Engineering", age: 36}
#=> %{
#=>   age: 36,
#=>   division: "Engineering",
#=>   name: "Sally Green",
#=>   position: "VP of Engineering",
#=>   region: "west-1"
#=> }
```

This can set multiple values at one time. This update recognized Sally's promotion and that she is now older. Likewise, it doesn't update the `person` unless we explicitly re-bind the `person` variable to the updated map.

```
person = %{person | position: "VP of Engineering", age: 36}
```

This only works for ***atom*** keys that ***already exist*** on the map. It will error when attempting to set a key that doesn't exist.

```
%{person | salary: 100_000}
#=> ** (KeyError) key :salary not found in: ...
```

Try performing some of your own updates in an IEx session. What do you have to do to keep the changes?

More Resources

For more detailed information on the built-in functionality for working with maps, refer to these resources:

- Map module
- Access module
- Kernel.get_in/2
- Kernel.pop_in/2
- Kernel.put_in/3
- Kernel.get_and_update_in/3

Introducing Modules and Functions

A quick overview and introduction to an Elixir **module** and **function** is needed.

In functional programming, it is just "functions" and "data". Functions are the instructions of how to transform some given data. Modules are containers for functions that also provide a namespace.

Modules

Modules serve both as a container for functions and as a namespace.

A module is defined using the `defmodule` command and the name starts with an uppercase letter.

```
defmodule MyFoo do

  def foo do
    "Hello!"
  end

end
```

In order to execute the `foo` function, I need to provide the namespace to the function. You can copy/paste the above module code into IEx to play with it.

```
MyFoo.foo
#=> "Hello!"
```

Parenthesis are optional in Elixir. The above could also be written as `MyFoo.foo()`. If there is ambiguity about the syntax then the parenthesis are required by compiler.

Functions in IEx

Functions must be defined inside of a module. This makes playing with them less convenient in IEx. Typing and editing module code into IEx is awkward. Here are three ways you can play with modules at this stage:

1. Edit the code in a text file and paste it into IEx
2. Edit the code in a text file and execute the file as a script
3. Start a “mix” project

The first one you can do easily enough on your own. Editing the code and pasting it into IEx will “re-write” the module.

Running an Elixir Script

The second option is to create an Elixir script file. You can execute the script from the command line. This makes it easier to work with slightly more complex code samples.

Thinking Tip

Read “Running an Elixir File as a Script” for a walk-through of how to do it. It includes explanations of what’s happening and how to recognize when you’ve grown out of a script.

Creating a Simple Mix Project

The third option is to create a simple Elixir “mix” project. I recommend this approach for the following reasons:

- Simple to create
- A mix project is very small
- Easily supports organizing your code into multiple files
- Starts you off with a testing framework setup

Thinking Tip

Read "Creating Your First Mix Project" for a walk-through of how to do it. It includes tips on working with your code in a mix project.

For more details on how to do this, please read the post " Creating Your First Mix Project" which explains it in more detail.

Function "Arity"

In Elixir we talk about functions and their " arity". Meaning the number of arguments that a function takes.

Let's add two functions called `greeting` to our `MyFoo` module. You can copy/paste this into IEx.

```
defmodule MyFoo do

  def foo do
    "Hello!"
  end

  def greeting(name) do
    "Hello #{name}!"
  end

  def greeting(name, extra_greeting) do
    "Greetings #{name}! #{extra_greeting}"
  end

end
```

In IEx, I can use auto-completion to see the functions available on the module I just defined. After `MyFoo.` press the `TAB` key to see the auto-complete options.

```
iex> MyFoo.
foo/0         greeting/1    greeting/2
iex> MyFoo.
```

It shows the two different `greeting` functions. One with the `/1` and the other with a `/2` to identify how many arguments they take.

Function Return Values

Elixir uses an "implicit return" for functions. You don't explicitly say "return" this value. Elixir returns the value of the last expression as the function result.

In the " `MyFoo.foo` " example, the return is the string `"Hello!"` because it was the last value expressed in the function. There is no way to *not* return a value. Given that this is Functional Programming, every function returns a value! You may choose to ignore it, but it will return *something*.

In this example, let's create a function that does nothing and returns nothing. What happens when we call it?

```
defmodule MyFoo do

  def do_nothing do

  end

end

MyFoo.do_nothing
#=> nil
```

It has to return *something* so it returns `nil`.

No Early Return?

Given that a function always returns something and the last thing the function does is used as the return value, some people ask, “Why can’t I return explicitly at an earlier point? Why doesn’t Elixir have an explicit return?”

The short answer is, “Erlang doesn’t have explicit returns either and Elixir is built on Erlang.”

The longer answer is, with pattern matching, you have a new tool to solve old problems in a new way. Once you get comfortable with pattern matching in functions, you won’t miss early returns. Your Elixir code becomes more understandable and readable without them. If this is a concern of yours, just know for now that it’ll be okay. Promise.

What if I don’t want to return anything?

There are times when a function does some work or creates a side-effect and there is *nothing* meaningful to return. A common pattern you’ll see in Elixir and Erlang is that those functions return the atom `:ok`.

```
defmodule Testing do

  def do_stuff do
    # do stuff that can't fail or any errors are handled
    :ok
  end

end

Testing.do_stuff
#=> :ok
```

Private Functions

Functions defined with the `defp` macro are “private”. They are not exported from the module. The can be called from within a module, but are not available outside the module.

```
defmodule MyApp do

  def public_do_work(input) do
    private_work(input)
  end

  defp private_work(_input) do
    IO.puts "working!"
  end
end

MyApp.public_do_work(123)
#=> working!
#=> :ok

MyApp.private_work(123)
#=> ** (UndefinedFunctionError) function MyApp.private_work/1 is undefined or private
#=>     MyApp.private_work(123)
```

Passing a Function by Name

We can refer to a specific function by name using it’s arity. We use the `&` to express that we want a reference to the function.

```
say_hello = &MyFoo.greeting/1
#=> &MyFoo.greeting/1
```

I can now execute the function bound to the variable using a `.` to identify that the variable *references* a function to execute and isn't the name of the function itself.

```
say_hello.("Mark")
#=> "Hello Mark!"
```

Using this technique, we can pass a function as a parameter into other functions.

```
defmodule MyFoo do

  def greeting(name) do
    "Hello #{name}!"
  end

  def process_name(name, fun) do
    fun.(name)
  end

end
```

Now we can let the `process_name/1` function combine a piece of data and a function that we provide.

```
MyFoo.process_name("Mark", &MyFoo.greeting/1)
#=> "Hello Mark!"

MyFoo.process_name("Mark", &IO.puts/1)
#=> Mark
#=> :ok

MyFoo.process_name("Mark", &String.to_atom/1)
#=> :Mark
```

Default Arguments

An argument to a function can be given a default value using the `\\` operator. It looks like this:

```
defmodule MyFoo do

  def some_function(value \\ :default) do
    value
  end

end
```

Let's create a new greeting function that also gives a compliment. We'll provide a default compliment.

```
defmodule MyFoo do

  def greeting_with_compliment(name, compliment \\ "You look nice today!") do
    "Greetings #{name}! #{compliment}"
  end

end
```

When I execute the function with only a name given, the default compliment value is used. I can provide an override compliment to use for that specific case.

```
MyFoo.greeting_with_compliment("Tom")
#=> "Greetings Tom! You look nice today!"

MyFoo.greeting_with_compliment("Bill", "That color suits you.")
#=> "Greetings Bill! That color suits you."
```

Multiple Functions are Created

When you give a default value to an argument, Elixir creates 2 versions of the function.

Using auto-completion, I can see that two functions were created.

```
iex> MyFoo.greeting_with_compliment
greeting_with_compliment/1    greeting_with_compliment/2
```

We didn't explicitly create a `greeting_with_compliment/1` function. That one was created for us. If we could see the generated code, it would essentially look like this.

```
def greeting_with_compliment(name) do
  greeting_with_compliment(name, "You look nice today!")
end
```

When the `/1` function is called, it executes the `/2` version passing in the default value for that argument. This is helpful to understand so as you auto-complete your functions and see functions listed that you didn't create, you understand where they are coming from.

Module Names are Atoms Too!

Atoms are a significant part of Elixir. In fact, Elixir modules are atoms!

```
is_atom(MyFoo)
#=> true

Atom.to_string(MyFoo)
#=> "Elixir.MyFoo"

String.to_atom("Elixir.MyFoo")
#=> MyFoo

:"Elixir.MyFoo" == MyFoo
#=> true
```

Behind the scenes, an Elixir module is an atom with "Elixir" as part of the name. This namespaces it and makes it easier to identify internally.

Aliases

As you organize your code into modules and namespaces, it can become pretty long. You can use an "alias" to create a name shortcut for your code. Imagine something like the following module.

```
defmodule MyApp.Customers.Billing.History do

  def compute_for_period(_from_date, _to_date) do
    # compute the value
    103.5
  end

end
```

In order to execute the function from outside the module, the full namespace is needed. This quickly becomes tedious.

```
month_total = MyApp.Customers.Billing.History.compute_for_period(month_start, month_end)
```

An alias can be declared anywhere. However, it is convention to declare the aliases all together at the top of a module.

```
defmodule MyApp.Customers do
  alias MyApp.Customers.Billing.History

  def compute_current_month() do
    # get the start/end dates for the current month
    History.compute_for_period(month_start, month_end)
  end

end
```

Be default, the alias name is the last piece of the namespace. In this case, “History”. Any references to “History” inside the module declaring the alias are a shortcut to the full name.

Thinking Tip: Aliases are not Imports

In other languages, you must “import” or “require” code from another file into your current file before you can call it. That is not the case in Elixir. An alias is not an import. It is only a name shortcut. You can use the full namespace name to execute any code available in your application. All *public* code is available to the application. Code is just a set of instructions. Only private functions are blocked from execution outside of a module.

Override the Alias Name

If the default name would create collisions or be unclear, you can alias it “as” something else to explicitly give it a name.

Let’s define some poorly named modules that will create a name collision when we alias them.

```
defmodule MyApp.Customers.Orders.Process do

  def perform(_order) do
    IO.puts "performing order work"
    :ok
  end

end

defmodule MyApp.Customers.Jobs.Process do

  def perform(_job_info) do
    IO.puts "performing job work"
    :ok
  end

end
```

When I have some code that needs to access **both** of the above modules, declaring the aliases like this won’t work as expected. No error occurs, but the last alias command wins and the alias to “Process” is overwritten.

```
alias MyApp.Customers.Orders.Process
#=> MyApp.Customers.Orders.Process

alias MyApp.Customers.Jobs.Process
#=> MyApp.Customers.Jobs.Process

Process
#=> MyApp.Customers.Jobs.Process
```

An alias that renames the default name might look like this:

```
alias MyApp.Customers.Orders.Process, as: OrderProcessor
alias MyApp.Customers.Jobs.Process, as: JobProcessor

OrderProcessor.perform(123)
#=> performing order work
#=> :ok

JobProcessor.perform(123)
#=> performing job work
#=> :ok
```

I don't recommend module naming like this, but I've seen it enough times now that it's worth mentioning.

Introducing the Struct

A struct is an extension of a map with more strict rules about what keys it can have. A struct's keys are atoms and cannot be strings. Because of the strict definition of a struct, Elixir can provide compile-time checks for the keys. This won't *prevent* you from adding invalid keys to the map, but it works very well for catching typos and when a key is renamed.

A struct is defined *inside* a module and the name of the struct *is* the module itself.

A simple struct:

```
defmodule Player do

  defstruct [:username, :email, :score]

end
```

This defines a Player struct where the data has 3 named attributes. For a new struct, all the attributes will have the default value of `nil`. When you copy/paste that module definition into IEx, you can then auto-complete on "`Player.`" and it completes to `Player.__struct__`. Execute that and you see the struct.

```
Player.__struct__
#=> %Player{email: nil, score: nil, username: nil}
```

The struct gets the name of the module. Since we didn't define any default values for the struct, Elixir assigned `nil` to each of the attributes. For a `Player`, it would make sense that the score should default to 0 instead of `nil`.

Default Values

When declaring the struct, we can provide a keyword list to provide default values. That looks like this:

```
defmodule Player do

  defstruct username: nil, email: nil, score: 0

end
```

When creating a new Player struct, it defaults the score to 0 for us.

```
%Player{}
#=> %Player{email: nil, score: 0, username: nil}
```

Compile Time Checks on Keys

As mentioned before, Elixir can perform compile time checks for valid keys on the struct. This is an example of a runtime error for an invalid key.

```
%Player{lives: 100}
#=> ** (KeyError) key :lives not found
#=>     expanding struct: Player.__struct__/1
```

A Struct is a Map

A struct is a map and can be accessed using normal map functions.

```
gary = %Player{username: "Gary", score: 100}
#=> %Player{email: nil, score: 100, username: "Gary"}

is_map(gary)
#=> true

Map.get(gary, :score)
#=> 100

gary.score
#=> 100
```

Thinking Tip: A struct can be like an OOP class

An Elixir struct is similar to a “class” in Object Oriented Programming languages. If you think about a class as the explicit linking of a data structure and the methods (or functions) that operate on that data, then a well-defined struct/module can do the same thing. The main difference for a struct is that the data structure and the functions are not explicitly *tied* together. We *define* them in the same place as a convenience both to the developer creating the data structure and writing the tests, but also to the developer using the struct. The primary functions for operating on the struct are located in the same namespace.

No Default Access Behaviour

A struct does not implement the Access Behaviour mentioned when talking about Maps. That behaviour (yes, spelled the British English way) allows you to use `[]` to provide a key to access a value in a map. When you try that on a struct it fails.

```
gary[:username]
#=> ** (UndefinedFunctionError) function Player.fetch/2 is undefined (Player does not implement the Acces
s behaviour)
#=>     Player.fetch(%Player{email: nil, score: 100, username: "Gary"}, :username)
#=>     (elixir) lib/access.ex:322: Access.get/3

gary.username
#=> "Gary"
```

Structs have pre-defined atom keys. You can use the key name like `gary.username` and it returns the value.

There’s Much More to Structs

This is only an introduction to Elixir structs so we have enough of a foundation to use them in pattern matching. The combination of structs and pattern matching are powerful and beneficial. They are two features that go great together.

For more information on structs, they are covered in more detail in the Elixir Fundamentals Collection. You can also find information online.

Introduction to Pattern Matching

Meet the Match Operator "="

The " = " sign in most programming languages means "assignment". You are *assigning* some value to a variable. In Elixir, we don't "assign" values to variables, we "bind" a variable to point to a value. That gives us our first hint that things are different here. In Elixir, we call " = " the "Match Operator". Now you *know* it's going to behave differently when it has such a different name!

Similar to operators like " + ", " - ", and " * " there is a left and right side of the Match Operator. The left side of the match operator is the *pattern*. The right side is the *data* being matched.

```
pattern = data
```

What Happens in a Match?

Three things that can *all* happen *at the same time* when a match is performed.

1. Match the **type** of data
2. Match the **shape** of data
3. **Bind values** to variables

Let's see an example of doing those three things the "normal" non-pattern matching way. Let's say the data we want to use is the following map.

```
data = %{name: "Howard", email: "howard@example.com"}
```

Now imagine I have a function that takes a given piece of data that may or may not be a map. I want to verify that it *is* a map, then that the map has the key `:name` and finally, bind a variable called `name` to that value in the map. The code might look something like this:

```
name =                              # result is bound to name
  if is_map(data) do                # match type
    if Map.has_key?(data, :name) do # match shape
      Map.get(data, :name)          # get :name value
    end
  end

name                      # name variable is bound to value
#=> "Howard"
```

I systematically "poke" the data blindly trying to feel out it's shape. Once I know that it *is* a map and it *has* a `:name` key, then I can get that value and bind the `name` variable.

A Pattern Match match does this all in *single statement*. Instead of a series of "pokes" at the data, I say, "This is the type and shape of the data I want. If it matches, bind the value to a variable called `name` ".

```
%{name: name} = %{name: "Howard", email: "howard@example.com"}

name
#=> "Howard"
```

This is "declarative". I state the conditions that must be met for it to match (the pattern) and *if* it matches, the value is bound!

If the **type** matches and the data's **shape** matches, then it "pulls apart" the data by binding values to variables. This is also "destructuring" the data or "unpacking" it for us.

Thinking Tip: Elegance in communication

The imperative version is a series of instructions that accomplishes something similar to the pattern matching version but it is much less clear. The imperative version tells us *how* to do it but a developer reading then code is left to figure out *what* is being done.

Pattern matching gives us an elegance in communicating *what* is happening to the developer. It is clearer, simpler, and so much better!

Pattern matching helps us create code that avoids nested **if** statements. Our code becomes flatter, clearer, easier to read and easier to maintain.

The Simplest Match

The simplest version of a match is this:

```
x = 5
#=> 5

x
#=> 5
```

It doesn't *look* like anything special. It looks like a normal variable assignment you see in other languages. However, the BEAM takes the statement and perform those three pattern matching steps for us:

1. Does the **type** match? No type was specified, so "yes".
2. Does the **shape** match? No shape was specified, so "yes".
3. **Bind** the variable to the value.

After this statement, `x` is bound to the value `5`. It skips right to binding because we didn't provide a pattern to match.

Match Without Binding

You can perform a Pattern Match without binding a variable to a value. This is an example of that:

```
x
#=> 5

5 = x
#=> 5
```

If `x` has the value of 5, and we are matching the statement `5 = x` then the BEAM performs the Pattern Match steps for us like this:

1. Does the **type** match? The left side is an Integer and `x` is bound to an Integer, so "yes".
2. Does the **shape** match? The left side has the shape `5` and `x` is bound to a shape of `5`, so "yes".
3. Nothing to bind. No variables were given on the left side.

The statement `5 = x` is invalid in assignment-based languages but in Elixir, this is a valid match!

You may never have thought of the number `5` as a "shape" for data, but it is! You can think of the number 5 as a specific shape of an Integer.

Match Error

What happens when a match is **not** made? When we write a Pattern Match like `5 = 3`, we are telling the BEAM, "This **has to** match!". The expression doesn't give the BEAM an alternative for what to do when it doesn't match. We are basically *asserting* that it matches with the way the expression is written. When the BEAM has no other option, a Match Error is raised.

Here's an expression that *cannot match* and results in a Match Error.

```
5 = 3
#=> ** (MatchError) no match of right hand side value: 3
```

The BEAM is saying, “You asserted that this matches and it doesn’t. Error!”

We will look at Pattern Match expressions that allow for elegant alternatives to a Match Error. In fact, those other ways are at the heart of what makes pattern matching so awesome!

The Pin Operator: ^

Sometimes we want to use a variable in a match without having the variable become bound to a value. To do this we use the “Pin Operator” `^`. If you are familiar with pointer-based languages, I think of it as a “pointer dereference”. It’s simple enough to think of it as saying, “don’t *bind* me, use the value I *point to* for the match comparison” and it has a little arrow-like character `^` to help visualize it is *pointing back* to the value.

Let’s see what that looks like:

```
x = 5
#=> 5

^x = 6
#=> ** (MatchError) no match of right hand side value: 6

^x = 5
#=> 5
```

The Pin Operator can be used with every data type. It can be very useful in pattern matching.

Exercises

Here are some commands to try out in an IEx session. Play with what makes a match and what doesn’t. Get familiar with the Match Error message because while you are learning Elixir, you will likely encounter it a lot. As you become more comfortable, you will instinctively *know* what will match and you will encounter the error a lot less.

```
a = 1
#=> 1

a = 1 + 5
#=> 6

6 = a
#=> 6

5 = a
#=> ** (MatchError) no match of right hand side value: 6

3 = 2
#=> ** (MatchError) no match of right hand side value: 2

3 = 3
#=> 3

x = 5
#=> 5

^x = 3 + 2
#=> 5

^x = 3 + 5
#=> ** (MatchError) no match of right hand side value: 8

^x = x
#=> 5

:ok = :ok
#=> :ok

:ok = :error
#=> ** (MatchError) no match of right hand side value: :error

[1, 2] = [1, 2]
#=> [1, 2]

[1, 2] = [3, 4]
#=> ** (MatchError) no match of right hand side value: [3, 4]
```

Limits to Matching

Pattern matching is everywhere in Elixir. It is a powerful and central language feature. There is a limit that is important to understand. **You cannot execute a function on the left-hand side of a match.**

```
length([10]) = 1
#=> ** (CompileError) iex:6: cannot invoke remote function :erlang.length/1 inside match

length([10]) == 1
#=> true
```

Note that the " == " or *equality* operator works as you'd expect.

Functions can create side effects. Examples of side effects:

- Making an HTTP call to an external service
- Modifying records in the database
- Creating logging output
- Writing to a file
- Sending a message to another process that creates a side effect like modifying the database

The BEAM is trying to determine *if* the data on the left-side matches what is on the right. *If* it matches, then we may want to intentionally create side effects in our system. **The process of performing a match is not allowed to create side effects.** Because of this, no function calls are allowed on the left side of a Pattern Match statement.

Recognizing a Common Error

If you are coming from an Object Oriented language, you will likely accidentally see this error a lot in the beginning. A common mistake for developers new to immutable data structures is trying to mutate data. The result will be the error `cannot invoke remote function [?] inside match`. Let's see how this happens.

```
user = %{name: "Jim"}
#=> %{name: "Jim"}

user.name
#=> "Jim"

user.name = "Howard"
#=> ** (CompileError) iex:3: cannot invoke remote function user.name/0 inside match
```

There are a few things happening with this code.

1. The developer is trying to use the match operator " = " to perform an "assignment". In effect, they are trying to "assign" the value "Howard" to the map's `name` key. This isn't how pattern matching binds variables.
2. The developer is trying to *mutate* the map. Remember, data in Elixir is **immutable**. The change could be made with `user = Map.put(user, :name, "Howard")` or using the abbreviated update syntax of `user = %{user | name: "Howard"}`.
3. The left side is actually a function `user.name/0`. Functions are not allowed on the left side of a match.

Why did the error say `cannot invoke remote function user.name/0 inside match`? Why did it say we were trying to execute a function on the left-side of the match operator?

In Elixir, the Access behaviour is a convenience that is available to a Map and other data structures. It turns our short-hand access of `user.name` into a function that effectively calls `Map.get(user, :name)` for us. Even though we didn't create or call the `name/0` function, the Access behaviour was executed for us, and *that* is how the function was called. So the left side of the match statement turned into a function call! Couple that with a misuse of the match operator and now you understand the error message.

Hopefully this helps you quickly recognize the error message you'll see as you occasionally forget and fall into an old code habit and misuse the match operator. It's okay. It takes time to establish new habits and new ways of thinking about something as common as the equals sign " = ".

Matching Complex Data Types

"Complex Data Types" refers to collection data structures. They include tuples, maps, structs and lists. Each type can contain other data structures either simple or complex.

Matching to Destructure Data

The Match operator is really powerful for pulling data apart or "destructuring" the data. This is what the "binding" step is doing. The BEAM can bind a variable to a location deep in a data structure just because it matched the pattern we described.

We'll look at some examples of how this works for the different collection data types.

- Map
- Struct
- Tuple
- List

We'll also look at how to do this with the Strings and binary types.

Case Statement

Part of the magic of pattern matching is that when it *doesn't* match our specific pattern, it continues on to try the next pattern. To play with this, we'll use the `case` statement. Pattern matching is everywhere in Elixir and that is true for the `case` statement as well.

Let's look at how a case statement works.

```
case data_being_examined do
  pattern_1 -> code_to_execute_when_matches
  pattern_2 -> code_to_execute_when_matches
  pattern_3 -> code_to_execute_when_matches
end
```

The patterns are checked from top to bottom. If it doesn't match `pattern_1` then it checks `pattern_2` and so on. This looks a lot like a `switch` statement in Javascript.

Three important things to remember:

1. it tests the patterns going from **top to bottom**
2. it breaks or **stops on the first pattern that matches**
3. it has all the power of pattern matching (type, shape and binding)

Matching a Map

With a `case` statement to play with, let's start with a Map. Let's look at an example using code you can paste into IEx.

```
data = %{name: "Howard", age: 35}

case data do
  %{name: "Howard"} -> "Yes sir Mr. Admin!"
  %{name: name} -> "Greetings #{name}!"
  %{age: age} -> "I don't know who you are, but you're #{inspect age} years old!"
  _other -> "Uhh.... what's that?"
end

#=> "Yes sir Mr. Admin!"
```

After pasting that code in, it returns `"Yes sir Mr. Admin!"`. The match was for a map (type), with a `:name` key and a value of `"Howard"` (shape). That's very specific match! Also note, **it stopped at the first match**.

Let's try sending a different piece of data though the `case` statement and see what happens.

```
data = %{name: "Jill", age: 30}

case data do
  %{name: "Howard"} -> "Yes sir Mr. Admin!"
  %{name: name} -> "Greetings #{name}!"
  %{age: age} -> "I don't know who you are, but you're #{inspect age} years old!"
  _other -> "Uhh.... what's that?"
end

#=> "Greetings Jill!"
```

After pasting in the `case` statement again, you should get a different result. The more specific pattern that included the name `"Howard"` in the pattern didn't match the data for `"Jill"`. So the pattern match moved on to the next pattern and that one matched.

The 2nd pattern given is for a map (type) with a `:name` key (shape) but we don't specify what the value must be. We provide a `name` variable to bind the value to and magically it is bound to the string `"Jill"` and available for use in our code!

Now what will happen if the data being matched is the number `5`?

```
data = 5

case data do
  %{name: "Howard"} -> "Yes sir Mr. Admin!"
  %{name: name} -> "Greetings #{name}!"
  %{age: age} -> "I don't know who you are, but you're #{inspect age} years old!"
  _other -> "Uhh.... what's that?"
end

#=> "Uhh.... what's that?"
```

The first 3 patterns we specified are all maps. When it doesn't match any of those, we can specify a "catch all" pattern that will match no matter what. You can think of this like an `else` or `default` in other `switch` -style statements. This pattern doesn't define a type or a shape, so it matches.

Order is Important!

We've already seen that the patterns match from top to bottom. Let's make that *really clear* here because it is *really important*!

Returning to the first data example with Howard, we'll rearrange the patterns.

```
data = %{name: "Howard", age: 35}

case data do
  %{name: name} -> "Greetings #{name}!"
  %{name: "Howard"} -> "Yes sir Mr. Admin!"
  %{age: age} -> "I don't know who you are, but you're #{inspect age} years old!"
  other -> "Uhh.... what's that?"
end

#=> "Greetings Howard!"
```

The less specific pattern that doesn't match on the value of the name is now higher. The pattern matches and the result is `"Greetings Howard!"` even though a more specific and "better" match exists. It stops at the first match!

At the risk of over emphasizing this point, lets rearrange the patterns one more time. Let's move the `other` pattern to the top.

```
data = %{name: "Howard", age: 35}

case data do
  other -> "Uhh.... what's that?"
  %{name: "Howard"} -> "Yes sir Mr. Admin!"
  %{name: name} -> "Greetings #{name}!"
  %{age: age} -> "I don't know who you are, but you're #{inspect age} years old!"
end

#=> "Uhh.... what's that?"
```

In this example, we can see that multiple other patterns are "better" matches for the data than the " `other` " pattern. It doesn't matter though, the first pattern to match gets the data!

Thinking Tip: Be Specific!

Try to start with the most explicit or specific pattern you can. These are often the "edge cases".

Using the underscore to define shape

A significant part of pattern matching is defining the "shape" of the data. Not just the values or attributes, but the very shape itself. To help with this, we can use the underscore character " _ " to help define a shape that lets us ignore the value.

```
%{name: _}
```

This defines a pattern for a map that has an atom key of " :name ". Because of the _ , we skip the "binding" step and we also don't specify a value it should have. This lets us be more specific about the shape of the data that we care about for this pattern.

Named for Developer Clarity

We can give a name to the ignored space that helps for developer clarity. It doesn't impact the usage or value, but it does convey meaning to the developer.

```
%{name: _username} = data
%{name: _company_name} = data
%{name: _pet_name} = data
```

All 3 of the patterns describe the same shape. The difference is in the named placeholder. This doesn't *change* the shape but it is certainly helpful to another developer who comes across the code later!

Tuples and Shape

Tuples are very specific about their shape. A two element tuple is completely different from a three element tuple. Using the _ underscore is very helpful for specifying the shape of the pattern.

```
{_, _, _}
```

This defines the pattern for a 3 element tuple. It doesn't matter about the "values" in the tuple, but it defines the "shape" as being a tuple with 3 elements.

Naming the placeholders for developer clarity is really beneficial here. The following pattern still defines a 3 element tuple. However, it gives much more meaning to the positions!

```
{_year, _month, _day}
```

Deeper Data Matches

If you've ever dealt with pulling data out of a deeply nested structure, then you've felt the pain of deeply nested if statements. With pattern matching, you can handle pulling out deep data much more elegantly. In this example, our data is a map with two deeper nested maps. The value we want in on the very inside map, but only if the important_flag value is set to true .

```
data = %{
  important_flag: true,
  level_1: %{
    other: "stuff",
    level_2: %{
      value: 123,
      more: "stuff"
    }
  }
}

case data do
  %{important_flag: false} -> {:ok, 0}
  %{important_flag: true, level_1: %{level_2: %{value: value}}} -> {:ok, value}
  _other -> {:error, "Invalid data"}
end

#=> {:ok, 123}
```

Try changing the shape of the data and run it against the case statement to see how it behaves.

Nesting Data Types

Pattern matching works as you nest different data types inside each other. Imagine a function that looks up a user by some search criteria. If it finds the user it is returns an `{:ok, user}` tuple.

In this example, we can match the two element tuple (type and shape), the first element is an `:ok` atom (shape), the second element is a map (type), the user `:email` key (shape) and finally bind the email variable to the value. This is a very common pattern!

```
user = %{id: 1, name: "John", email: "john@example.com", active: true}
function_call_result = {:ok, user}

case function_call_result do
  {:ok, %{email: email}} -> "Sending email to: #{email}"
  _other -> "Nothing to do"
end

#=> "Sending email to: john@example.com"
```

When you start to break it down, there's a lot happening here. But I don't have to think about it. I declare the pattern I'm looking for and when it matches, the instructions to perform are executed. It's declarative and clean. The Pattern Match works both for conditional logic and flow control.

Recap

Here we've covered working with maps and tuples in pattern matches. The point worth understanding is the following things apply to *all* data types, not just maps and tuples.

- The `case` statement can match any data type
- Order in pattern matches is *very* important
- The first pattern to match is the one that's used
- The `_` can be used as a placeholder to define a pattern's shape
- Matching and binding to deeply nested data works on all types

Practice Matching a Map

You learn by doing. Here are a few exercises to play with in IEx. The solutions are here but hidden. Find the solution yourself first. Experiment. Have fun!

Exercise #1

Write the left hand side of this statement. Your goal is to *only* match when the map has an atom key of `:amount` and then binds the value for that key to the variable named `value`. What does the left side look like?

```
_your_statement = %{name: "Your Customer, Inc", amount: 142}
```

Showing Solution

```
%{amount: value}
```

The full match statement as:

```
%{amount: value} = %{name: "Your Customer, Inc", amount: 142}
```

The `value` variable should be bound to the value `142`. You can verify:

```
value
#=> 142
```

Exercise #2

Write the left hand side of this statement. Your goal is to *only* match when the map has a string key of `"name"` and then binds the value for that key to the variable named `name`. What does the left side look like?

```
_your_statement = %{"name" => "Your Customer, Inc", "amount" => 142}
```

Showing Solution

```
%{"name" => name}
```

The full match statement as:

```
%{"name" => name} = %{"name" => "Your Customer, Inc", "amount" => 142}
```

The `name` variable should be bound to the expected value. You can verify:

```
name
#=> "Your Customer, Inc"
```

Exercise #3

Write the left hand side of this statement. Your goal is to *only* match when the map has both string keys of `"name"` and `"amount"`. We don't care what the `"name"` value is, but the key should be present. Bind to the amount value. What does the left side look like?

```
_your_statement = %{"name" => "Your Customer, Inc", "amount" => 142}
```

Showing Solution

```
%{"name" => _name, "amount" => amount}
```

The full match statement as:

```
%{"name" => _name, "amount" => amount} = %{"name" => "Your Customer, Inc", "amount" => 142}
```

The `amount` variable should be bound to the expected value. You can verify:

```
amount
#=> 142
```

Practice Matching a Tuple

You learn by doing. Here are a few exercises to play with in IEx. The solutions are here but hidden. Find the solution yourself first. Experiment. Have fun!

Exercise #1

Write the left hand side of this statement. Your goal is to *only* match when the first element of the tuple is `:ok` and then bind the value `1500` to a variable named `answer`. What does the left side look like?

```
_your_statement = {:ok, 1500}
```

Showing Solution

```
{:ok, answer}
```

The full match statement as:

```
{:ok, answer} = {:ok, 1500}
```

The `answer` variable should be bound to the value `1500`. You can verify:

```
answer
#=> 1500
```

Exercise #2

Given a tuple that represents a date used in native Erlang functions, how would you bind a variable to the month value? The format is `{year, month, day}`

```
_your_statement = {2020, 2, 14}
```

Showing Solution

```
{_, month, _}
```

The full match statement as:

```
{_, month, _} = {2020, 2, 14}
```

The `month` variable should be bound to the value `2`. You can verify:

```
month
#=> 2
```

It can also be written using placeholder names. The values are still disregarded, but it communicates what kind of value is in that position.

```
{_year, month, _day} = {2020, 2, 14}
```

Matching a List

A list is a common data structure. You'll have lists of customers, orders, players, etc. You will need to process lists a lot. Because of how often you'll be working with them, let's spend a little more time talking about a different way to *think* about lists.

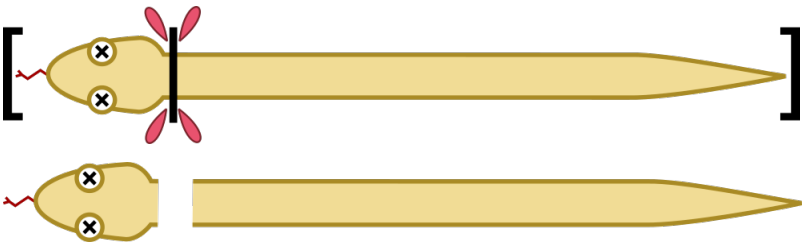
Lists are Freaky Little Snakes

Lists are very different from Arrays in other languages. They look deceptively similar. I think it helps to think about lists as snakes. A normal cute little snake. Not the scary kind at all.

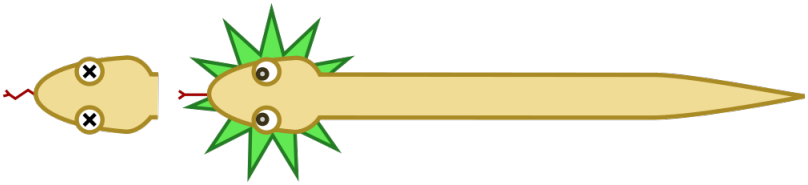
There are 2 parts to this snake. The "head" and the "tail".

Now imagine the snake has consumed a sequence of numbers. The first one eaten is the furthest inside. The last one eaten is still in the head.

I want to process this list and pull off the “head” value. Using a Pattern Match, I can cut off the head of the snake and break it into the two parts “head” and “tail”.



I’ve detached the head from the tail. Now the freaky part! The tail grows a new head!



In this way a list is recursive. When you remove the head, the tail is a new list and so has a head of its own!

Tools of the Trade

The square brackets and pipe characters (ie. `[`, `|`, `]`) are very useful when working with lists.

They can be used to add a new element to the front of a list:

```
existing = [2, 3, 4]
[1 | existing]
#=> [1, 2, 3, 4]
```

The same characters can be used in a Pattern Match to destructure a list and separate the list’s head from the tail.

```
existing = [1, 2, 3, 4]
[head | tail] = existing

head
#=> 1
tail
#=> [2, 3, 4]
```

Here we’re going to focus on using this tool for pattern matching.

Lists Matching

Let’s look at some examples for pattern matching a list.

What if I want to pull off the head of the list but I don’t care about the rest of the list. I only want the head. I might try this:

```
[head] = [1, 2, 3, 4, 5]
#=> ** (MatchError) no match of right hand side value: [1, 2, 3, 4, 5]
```

This won’t work because the pattern I defined on the left says it is a 1 element list. But that’s not what I want. By using the pipe `|` I can separate the head from the tail. Since I don’t *care* about the tail, I’ll use the underscore `_` to ignore it.

```
[head | _tail] = [1, 2, 3, 4, 5]

head
#=> 1
```

With this pattern I'm saying, "Bind the list's head to `head` and ignore the rest".

If I have a list of length 1, what happens? There are no numbers in the tail.

```
[head | tail] = [1]

head
#=> 1
tail
#=> []
```

Remember a list is recursive. Even when there is nothing in the tail, I will receive a new list. In this case, it happens to be empty. This allows me to Pattern Match a list with only 1 element in it too.

If I explicitly want to match against a list with only 1 element in it, I can this way:

```
[first] = [1]

first
#=> 1
```

Multiple Heads?

You are not limited to only putting 1 element in the head portion of the pattern `[head | tail]`. You just need to comma separate them.

```
[a, b, c | rest] = [1, 2, 3, 4, 5]

a
#=> 1
b
#=> 2
c
#=> 3
rest
#=> [4, 5]
```

The pattern `[a, b, c | rest]` says, "With a list of *at least* 3 elements, bind the 1st to `a`, the 2nd to `b`, the 3rd to `c` and whatever remains bind to `rest`." So this pattern will not match a list with two or fewer elements.

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

[a, b, c | rest] = [1]
#=> ** (MatchError) no match of right hand side value: [1]

[a, b, c | rest] = []
#=> ** (MatchError) no match of right hand side value: []
```

Matching an Empty List

If you want to match that a list is actually empty, the pattern to use is an empty list `[]`.

```
[] = []
```

Using the pattern `[head | tail]` with an empty list fails.

```
[head | tail] = []
#=> ** (MatchError) no match of right hand side value: []
```

The pattern is saying, “It must be a list with *at least* 1 element in it. It *can* have more, but must have at least 1.”

Matching to an Exact Sized List

We saw how you can match to “at least” a number by including the `|` and a variable or placeholder in the tail position. You can also match against an exact number. This is a pattern that more explicitly defines the shape of the list.

```
[a] = [1]

a
#=> 1

[a, b] = [1, 2]

a
#=> 1
b
#=> 2

[a, b, c] = [1, 2, 3]

a
#=> 1
b
#=> 2
c
#=> 3
```

Matching Values in the List

We can also be more explicit about the shape of the list by giving a pattern that includes explicit values.

```
[1 | rest] = [1, 2, 3, 4]

rest
#=> [2, 3, 4]
```

This pattern says, “It must be a list with at least 1 element and it starts with a `1`.” This pattern won’t match a list that starts with something other than a 1.

```
[1 | rest] = [2, 3, 4]
#=> ** (MatchError) no match of right hand side value: [2, 3, 4]
```

Pin Operator Matching Values

We can also use the Pin Operator `^` to describe a pattern.

```
head = 12
[^head | rest] = [12, 13, 14, 15]

rest
#=> [13, 14, 15]

head = 0
[^head | rest] = [12, 13, 14, 15]
#=> ** (MatchError) no match of right hand side value: [12, 13, 14, 15]
```

This pattern says, “It must be a list with at least 1 element and the first element must be the value bound to the `head` variable. Bind whatever else is there to the `rest` variable.”

Recap

Quick review of some important properties of pattern matching a List.

- A List is recursive. When we pull off the “head”, the “tail” is itself a list.
- We can specify patterns that a list must match an exact number of elements or “at least” a number of elements.
- We can use explicit value to define the shape of the data.
- We can use the Pin Operator to define the shape of the data.

Understanding how to Pattern Match lists lets us do some pretty cool things. One cool thing is we can recursively process a whole list using pattern matching without using a `for` or `while` style loop.

Practice Matching a List

You learn by doing. Here are a few exercises to play with in IEx. The solutions are here but hidden. Find the solution yourself first. Experiment. Have fun!

Exercise #1

Write the left hand side of this statement. Your goal is to bind the head of the list to a variable named `head` and the rest of the list to a variable named `tail`. What does the left side look like?

```
_your_statement = [1, 2, 3]
```

Showing Solution

```
[head | tail]
```

The full match statement as:

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

Verify that `head` and `tail` has the expected values:

```
head
#=> 1

tail
#=> [2, 3]
```

Exercise #2

Create a match that only matches when a list has a single item.

```
_your_statement = [10]
```

Your solution should **not** match with values on the right like these: `[10, 20]` and `[11, 12, 13]`.

Why would we want to *only match* in very limited cases? We don't want to have the "greediest" match or a match that catches the most it can, we want the *most specific* match that meets our requirement. Pattern matching lets us more explicitly and perfectly match what we are looking for.

Showing Solution

```
[first]
```

The full match statement as:

```
[first] = [10]
```

Verify that `first` has the expected value:

```
first
#=> 10
```

The other list types should result in a match error:

```
[first] = [10, 20]
#=> ** (MatchError) no match of right hand side value: [10, 20]
```

Exercise #3

Create a match that only matches when a list has at least two items.

Create a match that matches and pulls the first 2 items off the list and ignores the rest. Bind the first 2 elements to variables named `a` and `b`.

```
_your_statement = [1, 2]
```

Does your solution work with a list like `[1, 2, 3]`?

Showing Solution

```
[a, b | _rest] or [a, b | _]
```

The full match statement as:

```
[a, b | _rest] = [1, 2, 3]
```

Verify that `a` and `b` have the expected values:

```
a
#=> 1

b
#=> 2
```

Practice Project

A practice project is available to make it easier to play with pattern matching in increasingly advanced scenarios.

Pattern Matching practice project Download

The project has no other dependencies so if you already have Elixir installed then you are ready.

Intro to TDD

TDD stands for "Test Driven Development". The idea with TDD is that you spend some up-front time thinking about what your goal is with the code you will write. You first write a test that says, "given this situation and input, the code should cause this result or return this output." The test is written to "assert" that the desired behavior happened. If the code doesn't, then the test fails.

In this project the tests have already been written for you. The tests assert that a function behaves in a specific, desired way. Initially, all the tests are failing because the functions *don't* behave correctly. The functions haven't been implemented yet and that's *your* job!

The value of the tests is they help validate and check that your solution satisfies the requirements. They are additionally helpful because they make it easy to continue to re-check your solution when you refactor or make other code changes.

Running Tests

Try running the tests for the whole project first. From a command-line terminal, go to the directory location where you downloaded the files and run the following command:

```
mix test
```

You should see a **lot** of errors. That's expected. You are seeing *all* the failing tests for the whole project. Don't worry about that, I'll walk you through where to start working. We'll focus on a single file at a time and within that file we'll focus on just a couple tests at a time.

With the ability to run tests in the practice project, you're ready to start really digging in to pattern matching!

Pattern Matching a Function Body: Intro

We started playing with pattern matching using the `case` statement. This is valid and a very common tool. Now we really have fun when we start pattern matching with functions!

Matching with a Function

Pattern matching using a function declaration is really powerful! A big part of pattern matching is what happens when it *doesn't* match? It goes on to the next pattern to check. So it is with functions. We can define multiple versions of a function where each one defines a different pattern. These are different clauses. You can think of it like overloaded functions, but each version is to handle a specific data type and shape. It makes more sense when we have some code to look at.

```
defmodule Testing do

  def do_work(true) do
    {:ok, "I did some work!"}
  end

  def do_work(false) do
    {:ok, "I refuse to work."}
  end

end
```

Paste that module into an IEx shell. Using tab completion, we can see there is only 1 function declared on our module.

```
Testing.do_work
#=> do_work/1
```

Remember the `/1` means it takes 1 argument.

When we execute the function passing in `true`, pattern matching kicks in and the BEAM looks at the different function clauses we defined. It starts from the top and if the pattern matches, that version of the function gets executed.

```
Testing.do_work(true)
#=> {:ok, "I did some work!"}

Testing.do_work(false)
#=> {:ok, "I refuse to work."}
```

As defining functions goes, a function definition with types, shapes, and values in the function declaration looks weird! Honestly, until you understand pattern matching, even reading Elixir is confusing. I remember looking at Elixir code and not knowing how to mentally parse what it was doing! No where is that more true than with functions. When function declarations include patterns, they just don't even look like "normal" code.

> Until you understand pattern matching, even reading Elixir code is confusing.

Once I understood what was going on and how pattern matching in functions works, then I thought, "Why isn't every language doing this?" The problem I had with reading Elixir code was because I had never seen pattern matching in a language before. That won't be your problem now!

Thinking Tip: Function Overloading?

Is multiple function clauses the same thing as "function overloading"? Not really. Typically function overloading gets evaluated at *compile* time while pattern matching function clauses happens at *runtime*. The BEAM is comparing the data against the various function clauses. Beyond just the *type* of each argument, the BEAM can look deeper into the *shape* of the data for making the match.

Also, function overloading is often used in other languages to define a function with a different number of arguments. Remember, in Elixir functions are very much "arity" based. So a different number of arguments is a different function.

I don't think it hurts to think of it as function overloading if that helps you, however, they are different things.

Single Line Function Clauses

A common thing you will see in Elixir code is writing function clauses as a single line when then are simple. This is what that looks like:

```
defmodule Testing do

  def do_work(true), do: {:ok, "I did some work!"}
  def do_work(false), do: {:ok, "I refuse to work."}

end
```

This becomes very clear and clean!

When writing a function clause this way, please note the `def name(args), do: return_value` syntax. There is no closing `end` for the function. And note the `,` before the `do` and that the `do:` makes it an atom. You will see this code a lot but it has some special little syntax differences to pay attention to.

When a Function Doesn't Match

With the previous `do_work/1` example, what happens if I pass in some data that doesn't match the patterns I defined?

```
Testing.do_work("abc")
#=> ** (FunctionClauseError) no function clause matching in Testing.do_work/1
#=>
#=>     The following arguments were given to Testing.do_work/1:
#=>
#=>         # 1
#=>         "abc"
#=>
#=>     iex:3: Testing.do_work/1
```

It isn't a `MatchError` because we aren't using the Match Operator. It's a `FunctionClauseError`. We defined a `do_work` function with two clauses. The error tells us that **none** of the clauses we defined match the data. The error is helpful because it shows us what the data looked like that it couldn't match against.

Your Flip-the-Lid Clause

Our function clauses couldn't handle the data `"abc"` because they were explicitly matching on the values `true` and `false`. What if you want the ability to catch everything else? For an `if` statement, we're talking about the `else` clause. In Elixir, it's the "everything else comes here" function clause.

You already know how to bind to a variable without specifying a type or shape for the pattern. It's the same thing here! In this case, if we are given some data that we don't know how to process, we'll treat it as an error. Let's see what that looks like:

```
defmodule Testing do

  def do_work(true), do: {:ok, "I did some work!"}
  def do_work(false), do: {:ok, "I refuse to work."}

  def do_work(other) do
    {:error, "I don't know what to do with #{inspect other}"}
  end

end
```

The addition of a new function clause that doesn't define a pattern works perfectly for handling all the other data that can be thrown at our function without causing a `FunctionClauseError`. Let's throw some data at it and see what it does!

```
Testing.do_work("abc")
#=> {:error, "I don't know what to do with "abc""}

Testing.do_work(1)
#=> {:error, "I don't know what to do with 1"}

Testing.do_work(%{a_map: true})
#=> {:error, "I don't know what to do with %{a_map: true}"}

Testing.do_work([1, 2, 3])
#=> {:error, "I don't know what to do with [1, 2, 3]"}
```

Order Matters!

The order of the clauses really matters! The pattern matching stops when the first match is found. The data is "captured" by the function. Even if a *better* or *more exact* match exists, it won't be used if another function clause matches it first.

Let's see what happens when we put the "flip-the-lid" version first...

```
defmodule Testing do

  def do_work(other) do
    {:error, "I don't know what to do with #{inspect other}"}
  end

  def do_work(true), do: {:ok, "I did some work!"}
  def do_work(false), do: {:ok, "I refuse to work."}

end
```

The first thing you'll notice is we get 2 compiler warnings for the more specific function clauses.

```
warning: this clause cannot match because a previous clause at line 3 always matches
  iex:7

warning: this clause cannot match because a previous clause at line 3 always matches
  iex:8
```

The compiler can detect when a function clause will *never* be reached because the first clause doesn't define a pattern. Try executing the function passing in `true`.

```
Testing.do_work(true)
#=> {:error, "I don't know what to do with true"}
```

When we call the function with data that has a perfect match, it doesn't get executed! It is captured by the *first* match encountered. So order is very important. The order is checked goes top down.

Also note when a function has multiple arguments, it is possible to define function clauses where the compiler can't *tell* it isn't what we actually want. It won't warn us in every situation. It is up to us to pay attention to the order in which we define our clauses.

Code Without Pattern Matching

You could easily write equivalent code without using pattern matching. That version of it might look like this.

```
defmodule Testing do

  def do_work(value) do
    if value == true do
      {:ok, "I did some work!"}
    else
      if value == false do
        {:ok, "I refuse to work."}
      else
        {:error, "I don't know what to do with #{inspect value}"}
      end
    end
  end

end

Testing.do_work(true)
#=> {:ok, "I did some work!"}

Testing.do_work(false)
#=> {:ok, "I refuse to work."}

Testing.do_work("abc")
#=> {:error, "I don't know what to do with "abc""}
```

While this code may *feel* familiar, I strongly encourage you to avoid the temptation to fall back into this style. The "Elixir way" is to prefer the use of a pattern matching over `if` conditionals. The code is flatter and easier to read. It values the *pattern* of the data

over the explicit process of “poking” the data to figure out its shape.

Thinking Tip: See IF as an anti-pattern

Saying the Elixir way prefers pattern matching over `if` conditionals does not mean you can never use an `if` statement! There are valid cases for their use and they are part of the language! However, consider the use of `if` statements (especially an excessive use) as an anti-pattern in Elixir.

Compare the `if` conditional version above code to the pattern matching version. The Elixir-way is declarative, direct, clear, and much easier to reason about. At least it’s easier to reason about now that you understand pattern matching and function clauses!

Ready, Set, Go!

With the basics of pattern matching for the different data types covered and an introduction to using pattern matching with function clauses, you are ready to jump in and have fun!

Pattern Matching a Function Body: Tuples

The following exercises use the Pattern Matching project. Here you will focus on making a single test pass at a time.

The tests we are focusing on are in `test/tuples_test.exs`. Running the following command will execute *all* the tests in this file. Running all the tests now will show they all fail.

```
$ mix test test/tuples_test.exs

[...]

Finished in 0.07 seconds
7 tests, 7 failures

Randomized with seed 866762
```

I’ll spare you the display of the failing tests. Run it for yourself and you can see it! We’re going to tackle them one at a time. For the first one, I’ll walk you through it more step-by-step.

Exercise #1 – Tuples.day_from_date/1

In this exercise, we want to write a function clause that matches on a 3 element tuple and returns one of the values from it. In Erlang, dates are represented as a tuple like this: `{2019, 5, 30}`. The function you need to write will match on this type of date and return the day number from it.

First, run the test for this one by itself and see it fail and the message. You can do this by putting a `:` and the line number after the filename. The line number needs to be any line number that’s inside the test. I just chose line 10.

```
$ mix test test/tuples_test.exs:17
```

Running this the first time will include a lot of compiler warnings about unused variables. Disregard those for now.

```
$ mix test test/tuples_test.exs:17
Excluding tags: [:test]
Including tags: [line: "17"]



  1) test day_from_date/1 returns the day number from an erlang date (PatternMatching.TuplesTest)
     test/tuples_test.exs:9
     Assertion with == failed
     code:  assert 5 == Tuples.day_from_date({2018, 9, 5})
     left:  5
     right: nil
     stacktrace:
       test/tuples_test.exs:10: (test)



Finished in 0.07 seconds
8 tests, 1 failure, 7 excluded

Randomized with seed 164689
```

The error is nicely laid out for us. In the console it is colored for improved readability.

With a failing test ready that demonstrates the correct working behavior, it's your turn to write the code in the file being tested. In your editor, open `lib/pattern_matching/tuples.ex`. Find the function we want to write the implementation for.

```
defmodule PatternMatching.Tuples do
  @moduledoc """
  Fix or complete the code to make the tests pass.
  """

  def day_from_date(erl_date) do

  end

  [...]
end
```

Modify the argument to make it a Pattern Match and modify the body to the return the desired value. After making a change to the file, re-run the same test (hit the `up` arrow in the console to repeat the previous command). Does your test pass? If not, correct any mistakes and try again. If you have a passing test or you are totally stuck, check out the solution below.

Showing Solution
The function argument is a tuple. We only care about the `day` variable so the others are placeholders with a leading `_`. The function body returns the variable we care about, `day`.

```
  def day_from_date({_year, _month, day}) do
    day
  end
```

Exercise #2 – Tuples.has_three_elements?/1

The test we want to focus on is in `test/tuples_test.exs`. If the line numbers haven't changed, execute the failing test using:

```
mix test test/tuples_test.exs:25
```

Using a text editor, make changes to the function `has_three_elements?` in the file `lib/pattern_matching/tuples.ex`.

Make the test pass by using pattern matching in the function declaration. Create a second function clause that handles when it *doesn't* match the tuple. Keep re-running the `mix test` command for that specific test as you test your solutions.

Showing Solution
Two function clauses are used. One to match when the argument is a three element tuple and the other for anything else.

```
def has_three_elements?({_, _, _}) do
  true
end

def has_three_elements?(_tuple) do
  false
end
```

Exercise #3 – Tuples.major_us_holiday/1

The test we want to focus on is in `test/tuples_test.exs`.

```
mix test test/tuples_test.exs:40
```

Using a text editor, make changes to the function `major_us_holiday` in the file `lib/pattern_matching/tuples.ex`.

Make the test pass by using pattern matching in the function declaration. Create multiple function clauses to handle the different cases.

For this example, we only care about 3 specific holiday months.

- If the month is 12, return “Christmas”.
- If the month is 7, return “4th of July”
- If the month is 1, return “New Years”
- For any other month value, return the string “Uh...”

Showing Solution

```
def major_us_holiday({_, 12, _}), do: "Christmas"
def major_us_holiday({_, 7, _}), do: "4th of July"
def major_us_holiday({_, 1, _}), do: "New Years"

def major_us_holiday(_erl_date) do
  "Uh..."
end
```

Note that the order of the 3 functions matching the number is not important. Any order is valid. However, the “Uh...” response does need to be the last one as it will match with *any* value given it.

Exercise #5 – Tuples.greet_user/1

The first test to focus on is the “happy path” or “success” version of the function. The tests for this function are broken out into two tests. The success and failure cases.

```
mix test test/tuples_test.exs:52
```

The function should be altered to return the a greeting with the user’s name when it is passed in using an `{:ok, username}` tuple. A specific error text should be returned if given an `{:error, reason}` tuple.

Once you have the success version passing, write the failure handling version. That test is:

```
mix test test/tuples_test.exs:56
```

Showing Solution

```
def greet_user({:ok, name}), do: "Hello #{name}!"
def greet_user({:error, _reason}), do: "Cannot greet"
```

Exercise #6 – Tuples.add_to_result/1

This one is also broken out into two different test cases. The first is the “success” or “happy path” version. The second test is how it handles data it can’t operate on. Here are the two tests to focus on.

```
mix test test/tuples_test.exs:65
mix test test/tuples_test.exs:70
```

This represents a common pattern in Elixir. Receiving an `:ok` tuple, operating on the data and returning a new `:ok` tuple with the modified result.

Iterate editing the file and running tests until they both pass.

Showing Solution

```
  def add_to_result({:ok, result}), do: {:ok, result + 10}
  def add_to_result(error), do: error
```

All Tests Passing!

All the tests in this file should be passing now! Run the tests for the full file (not any specific line number). You are ready to move on.

```
$ mix test test/tuples_test.exs
........

Finished in 0.07 seconds
8 tests, 0 failures

Randomized with seed 57431
```

Pattern Matching a Function Body: Maps

Maps are your go-to key-value collections. You are likely to use them a lot. When building a web application, the request data submitted by a client is represented as a map with string keys. Before we jump into practicing how we pattern match maps in functions, this is a good time to talk about a code pattern you will see in Elixir.

In Pieces and Whole

There is a common code pattern we use in Elixir that is worth exploring here. Let’s look at a code example and talk about it.

```
params = %{"name" => "John", "email" => "john@example.com"}

defmodule Testing do
  def do_work(%{"email" => email} = params) do
    IO.inspect email
    IO.inspect params
    "Sent an email to #{email} addressed to #{params["name"]}"
  end
end

Testing.do_work(params)
#=> "john@example.com"
#=> %{"email" => "john@example.com", "name" => "John"}
#=> "Sent an email to john@example.com addressed to John"
```

In this code, we have a string-key map called `params` similar to what you’d get when handling a web request. The `do_work` function takes the params in and does a pattern match on some of the data. It looks like this: `do_work(%{"email" => email} = params)` . What’s special here is that we are pattern matching for some data we care about *and* still binding everything passed in as `params` .

This is a common code pattern. You can use this same approach everywhere you pattern match. It works in `case` statements and even in a straight forward match expression. Check this out:

```
%{"email" => email} = params = %{"name" => "John", "email" => "john@example.com"}

email
#=> "john@example.com"
params
#=> %{"email" => "john@example.com", "name" => "John"}
```

Yes, this may look strange, but when you break it down you can see what's happening. The match operator performs a Pattern Match of the left and right sides. When we say `params` by itself, we haven't defined a type or a data shape so it just gets bound.

This is effectively what we're doing with the map match in the function. We still bind everything to the variable `params` but we *also* define the pattern (data type and shape) which also pulls out the data we care most about.

This can be very helpful like in my hypothetical function below. I want to Pattern Match and access a portion of the data when it is passed in, but I still want the full `customer` data because I pass it on to another function called `notify_customer`.

```
defmodule Billing do
  def apply_charge(%{id: customer_id} = customer, charge) do
    record_charge(customer_id, charge)
    notify_customer(customer, charge)
  end
end
```

Still Naming for Clarity

You've seen how we can name an argument for developer clarity and documentation reasons using the underscore prefix like `_customer`. We can do that when matching map arguments. When I *don't* need the full data structure, it still adds value to name it for developer clarity and expressing the intent of the function.

```
defmodule Testing do
  def do_work(%{"email" => email} = _customer_params) do
    # use `email`
  end
end
```

Having the argument named `_customer_params` conveys meaning for what the developer was *expecting* to be passed in. That meaning may not be clear from just looking at a function declaration that says, " `do_work(%{"email" => email})` do". A map with an `email` string key could be for a user, customer, a login, or something else.

Be kind to your future self an any other developers that work on your code base. Name your variables even when the code doesn't need it.

Binding to a Nested Map

Maps are the go-to data structure for key-value data. They also elegantly handle nested data. When pattern matching, it is important to understand that you can also easily bind to a *nested* map. This can be very handy when working with web requests, and actually gets used frequently.

An example will help demonstrate this. First, we need some nested data to pattern match against. Remember, in this example we are focusing on the *function declaration* and the pattern match it uses.

```
params = %{
  "customer" => %{
    "id" => 123,
    "name" => "Willy Wonka Chocolates",
    "bonuses" => %{
      "employees" => %{
        "Oompa 1" => 1_000,
        "Oompa 2" => 2_000,
        "Hillary" => 1_500,
        "Oompa 3" => 500
      },
      "total" => 5_000
    }
  }
}

defmodule NestedBinding do

  def award_bonuses(%{"customer" => %{"bonuses" => %{"total" => bonus_total} = bonuses}} = _params) do
    IO.inspect bonus_total, label: "TOTAL TO VALIDATE"
    IO.inspect bonuses, label: "BONUSES"
    # TODO: validate intended total and employee amounts
    :ok
  end
end

NestedBinding.award_bonuses(params)
#=> TOTAL TO VALIDATE: 5000
#=> BONUSES: %{
#=>   "employees" => %{
#=>     "Hillary" => 1500,
#=>     "Oompa 1" => 1000,
#=>     "Oompa 2" => 2000,
#=>     "Oompa 3" => 500
#=>   },
#=>   "total" => 5000
#=> }
#=> :ok
```

The pattern is where we are focusing right now:

```
%{"customer" => %{"bonuses" => %{"total" => bonus_total} = bonuses}} = _params
```

Notice that we can bind to a deeply nested map to get the `bonus_total` *and* bind a variable to the full `bonuses` map at the same time!

The ability to bind to a nested piece of data works on *all* data types. However, it is most helpful when working with maps, primarily because maps tend to have nested data more than other data types.

Practice Exercises

The following exercises continue using the Pattern Matching project. We will continue focusing on making a single test pass at a time.

The tests we are focusing on are in `test/maps_test.exs`. Running the following command will execute *all* the tests in this file. Running all the tests now will show they all fail.

```
$ mix test test/maps_test.exs

[...]

Finished in 0.05 seconds
9 tests, 9 failures

Randomized with seed 958439
```

You should be more comfortable with the TDD (Test Driven Development) approach here. Let's get started with an easy warm-up.

Exercise #1 – Maps.return_name/1

The test to focus on is in `test/maps_test.exs`. Given a map with a `:name` key, return the value from the function.

```
mix test test/maps_test.exs:20
```

Showing Solution

```
  def return_name(%{name: name}) do
    name
  end
```

Exercise #2 – Maps.has_sides?/1

There are two tests to focus on. The first is the "success" path and the second is handing non-matching data. This is focusing on validating the shape of the data with less focus on the values and binding variables.

Refer to the tests for examples of the input data.

```
mix test test/maps_test.exs:30
mix test test/maps_test.exs:40
```

Showing Solution

```
  def has_sides?(%{sides: _num}), do: true
  def has_sides?(_value), do: false
```

Exercise #3 – Maps.net_change/1

The test we want to focus on is in `test/maps_test.exs`. It is under "net_change/1" and the test is "subtracts beginning balance from ending balance". If you haven't modified the file causing the line numbers to change, then using the following mix command will work.

There are two tests for this function. This function takes a map, performs a calculation and returns the result in an `{:ok, result}` tuple. The result is the difference of `:ending_balance` and `:initial_balance`.

```
mix test test/maps_test.exs:50
mix test test/maps_test.exs:60
```

Iterate on the code changes to make the tests pass. Use pattern matching in the function declaration.

Showing Solution

```
def net_change(%{initial_balance: initial, ending_balance: ending}) do
  {:ok, ending - initial}
end

def net_change(_value) do
  {:error, "Missing balance information"}
end
```

Exercise #4 – Maps.classify_response/1

This exercise is one of my favorites! This is based on several real-world experiences of mine. At some point, you will have a project that integrates with some other external service. That external service may return XML or JSON but not in any standardized, clean way. You are left dealing with whatever they give you. Even APIs returning JSON can be messy and not follow good REST practices. This exercise is inspired by those experiences. You don't get to control the data you have to react to. Your goal is to get the meaning and result you need from the given data.

When I was learning Elixir and encountered a situation like this, I created my first-pass solution. It was a traditional imperative step-by-step solution. Looking at my working solution, I wasn't satisfied. I thought, "That doesn't feel very Elixir-like yet". I went back, looked at more Elixir code, tried again. My second attempt still didn't feel quite "right" yet. After my third attempt, I ended up with a solution that "felt right". It was elegant and I was elated!

I stress this point because you need to know that it's okay to take multiple passes at a solution.

The test defines 4 sample responses from an external service. We want to be able to classify the response and identify when a request was successful and extract the desired value. There are 3 examples of how the request might fail. We want to be able to tell them apart as they will influence how the application responds.

The response data we need to look at is spread out throughout a nested map. Pattern matching lets you create an elegant solution.

There are 4 tests for this function. One is the happy-path solution and the other 3 are for handling the different error conditions we expect.

```
mix test test/maps_test.exs:86
mix test test/maps_test.exs:92
mix test test/maps_test.exs:96
mix test test/maps_test.exs:102
```

Matching Success

First, let's detect when it succeeds. Our function should return that it succeeded with an `{:ok, result}` tuple which include the extracted token. The test case include a comment that explains what about the data defines a success. The `setup` for the test cases includes sample response data.

```
mix test test/maps_test.exs:88
```

Make the test pass by using pattern matching in the function declaration.

Showing Solution

```
def classify_response(%{"success" => true, "token" => token} = _response) do
  {:ok, token}
end
```

Matching When Invalid

Next, let's add to our existing solution by moving on to the next test. Again, the test case includes a comment that defines when the response is "invalid".

```
mix test test/maps_test.exs:92
```

Showing Solution

```
  def classify_response(%{"success" => false, "messages" => %{"general" => %{"result_code" => -1}}} =
_response) do
    {:error, :invalid}
  end
```

Matching When Retry

Continue building on your work to add support for detecting when we should “retry” our request with the server. Again, the test case includes a comment that defines when the response tells us to “retry”.

```
mix test test/maps_test.exs:96
```

Showing Solution

```
  def classify_response(%{"success" => false, "messages" => %{"general" => %{"result_code" => 3}}} =
_response) do
    {:error, :retry}
  end
```

Matching When Account is Frozen

Conclude this exercise by building on your work to add support for detecting when a request failed because an account is “frozen”. Again, the test case includes a comment that defines when we can tell an account is “frozen”.

```
mix test test/maps_test.exs:102
```

Showing Solution

```
  def classify_response(%{"success" => false, "account" => %{"status_code" => "3001"}} = _response) do
    {:error, :frozen}
  end
```

Summary

All the tests for `classify_response/1` should be passing!

```
$ mix test test/maps_test.exs
Compiling 1 file (.ex)
.........

Finished in 0.05 seconds
9 tests, 0 failures

Randomized with seed 955953
```

Take a look at the full solution (hidden below) and think how the same solution would look if done imperatively using nested `if` statements! I love pattern matching! It makes the code declarative and clear. I feel like I’m a better programmer because of it.

Showing Full Solution

```
  def classify_response(%{"success" => true, "token" => token} = _response) do
    {:ok, token}
  end
  def classify_response(%{"success" => false, "messages" => %{"general" => %{"result_code" => -1}}} =
_response) do
    {:error, :invalid}
  end
  def classify_response(%{"success" => false, "messages" => %{"general" => %{"result_code" => 3}}} =
_response) do
    {:error, :retry}
  end
  def classify_response(%{"success" => false, "account" => %{"status_code" => "3001"}} = _response) do
    {:error, :frozen}
  end
```

When code is written this way, it is “declarative”. The code just “declares” the shape of the data it cares about. It basically says, “When the data matches this specific shape, it means we retry.”

When the same problem is solved without pattern matching (like when using nested `if` statements), reading it becomes a exercise of mentally parsing code to figure out what it’s doing. Instead of applying a single pattern definition for a match statement, it is step-by-step poking the data to “feel” what the shape is. “Does it have this key? Yes? So then does it have this next key?”

As you become more comfortable writing code this way, you won’t want to go back to the old way. Code becomes elegant. Your solutions become a declarative expression of what the data means. It’s easier to understand, easier to refactor, and more fun to write!

Pattern Matching a Function Body: Struct

Matching with a struct is just like matching with a map. The main differences with structs are:

- the keys are *always* atoms
- the compiler tells us if we get a key wrong
- the struct type gives us another thing to match against

In fact, you can write the pattern matching solutions to the tests using only maps! With the test code available, this is a good chance to explore and understand how this works and why there is value in using structs when you can.

The project defines two structs for us to play with here. Feel free to check them out.

- lib/pattern_matching/customer.ex
- lib/pattern_matching/user.ex

You can also see a description of the structs in IEx. Start the IEx using `iex -S mix` to load the project into the IEx session. Using the IEx helper `t()`, it can describe a type for us.

```
$ iex -S mix
Erlang/OTP 21 [erts-10.0.6] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> t PatternMatching.Customer
@type t() :: %PatternMatching.Customer{
        active: boolean(),
        contact_name: nil | String.t(),
        contact_number: nil | String.t(),
        location: nil | String.t(),
        name: String.t(),
        orders: list()
      }

iex(2)> t PatternMatching.User
@type t() :: %PatternMatching.User{
        active: boolean(),
        admin: boolean(),
        age: nil | integer(),
        gender: nil | :male | :female,
        hair: nil | String.t(),
        name: String.t(),
        points: integer()
      }
```

There are some similarities between the Customer and User structs. They both have `:name` and `:active` keys. All the other fields are different.

Compile Time Checks

A benefit of using struct types in Elixir is that you get compile-time checks that the keys are correct.

When we use IEx, the Elixir commands we enter are interpreted at runtime, they aren't compiled at build time. We still get errors for incorrect structs, but they look different. Let's see an example of both:

Example: IEx – Interpreted. Started using `iex -S mix`.

```
alias PatternMatching.User
user = %User{car: "Toyota"}
#=> ** (KeyError) key :car not found
#=>     (pattern_matching) expanding struct: PatternMatching.User.__struct__/1
#=>     iex:2: (file)
```

This is a runtime `KeyError` trying to use a key that doesn't exist on the struct.

Example: Compiled. Assumes you modify the code to be invalid and then try to start the project, which compiles it in the process.

```
$ iex -S mix
Erlang/OTP 21 [erts-10.0.6] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Compiling 1 file (.ex)

== Compilation error in file lib/pattern_matching/structs.ex ==
** (CompileError) lib/pattern_matching/structs.ex:9: unknown key :car for struct PatternMatching.User
    lib/pattern_matching/structs.ex:9: (module)
```

This is helpful in catching problems ranging from a simple typo to changing a struct definition where a key was renamed or removed.

However, if I perform the match as a map instead of the struct, I have no protections or guarantees. This alone is a significant benefit. It's the difference of including the struct name or not.

```
# This version is compile-time checked for Customer keys.
def do_work(%Customer{name: name}) do
  # work
end

# This version cannot be checked.
def do_work(%{name: name}) do
  # work
end
```

Guarantee it is the Struct

When a struct is part of the Pattern Match, it is a guarantee that the data coming in *is* that struct. Not just a map with similar keys, but it *is that struct*. All the code inside that function clause can be written confidently knowing it can't be something else with a similar structure.

It is totally valid to define a function clause that matches the struct just to have that assurance and protection. Here's an example:

```
def do_work(%Customer{} = customer) do
  # work
end
```

Inside this function clause I am guaranteed to have a Customer struct. This is opposed to a similar function declaration without it.

```
def do_work(customer) do
  # work
end
```

This function calls the argument `customer` but there is no guarantee it is a customer. The code would likely be written with the assumption it is a Customer struct but it implicitly depends on all callers to always only pass in a customer struct. When this function is passed something other than a customer, it will likely result in runtime errors.

Protects from Misspellings

Explicitly using a struct in a Pattern Match helps protect against misspelled keys. In the following example, I don't use the struct type but instead use a map. I've accidentally misspelled `name` as `nam`.

```
def do_work(%{nam: name} = customer) do
  # Will never match a %Customer{}!
end
```

Calling this function and passing in a Customer struct won't error, it will just never match! I've described a pattern that cannot match a Customer. When the Customer struct type is included, I get a compilation error for the invalid key.

Keep in mind that it is completely acceptable to *intentionally* use a map to match a struct. In this way my code can be somewhat polymorphic. I match on attributes of the data that multiple types can share. However, if I'm *expecting* the data to be Customer type, then it is preferable to explicitly declare that.

Practice Exercises

The following exercises continue using the Pattern Matching project. We will continue focusing on making a single test pass at a time.

The tests we are focusing on are in `test/structs_test.exs`. Running the following command will execute *all* the tests in this file. Running all the tests now will show they all fail.

Please review the solutions below even if you feel confident with your answer. You may find additional insight in the explanations.

```
$ mix test test/structs_test.exs

[...]

Finished in 0.05 seconds
9 tests, 9 failures

Randomized with seed 958439
```

Exercise #1 – Structs.get_name/1

In this exercise you write the function `get_name/1` that takes both a `User` and `Customer` struct.

There are 2 tests for this function. One is the happy-path solution and the other handles when some other data type is given.

```
mix test test/structs_test.exs:25
mix test test/structs_test.exs:30
```

Make the tests pass by using pattern matching in the function declaration.

Showing Solution
You can treat the data more generically as just a map. This lets you treat it somewhat polymorphically.

```
  def get_name(%{name: name}), do: {:ok, name}
  def get_name(_other), do: {:error, "Doesn't have a name"}
```

Exercise #2 – Structs.create_greeting/1

In this exercise you write the function `create_greeting/1` that handles receiving a `User` and `Customer` struct differently.

There are 2 tests for this function. One is the happy-path solution that handles creating a customized greeting for a `User` or `Customer` struct. The other handles when the `User` or `Customer` being greeted is inactive.

```
mix test test/structs_test.exs:38
mix test test/structs_test.exs:45
```

Make the tests pass by using pattern matching in the function declaration.

Showing Solution
The first Pattern Match handles when the subject to be greeted is inactive. It matches both a `User` and a `Customer`. The other two function clauses match the struct type to return an appropriate and customized greeting.

```
  def create_greeting(%{active: false}), do: {:error, "Recipient is inactive"}
  def create_greeting(%Customer{name: name}), do: {:ok, "Howdy customer #{name}!"}
  def create_greeting(%User{name: name}), do: {:ok, "Greetings user #{name}!"}
```

Exercise #3 – Structs.deactivate_user/1

In this exercise you write the function `deactivate_user/1` that handles receiving a `User` struct, modifying it and returning the modified struct.

There are 2 tests for this function. One is the happy-path solution that handles creating an updated `User` struct. The other handles when something other than a `User` is passed in.

```
mix test test/structs_test.exs:55
mix test test/structs_test.exs:60
```

Make the tests pass by using pattern matching in the function declaration.

Showing Solution

The first function clause matches on a `User` struct. It returns a new `User` struct with the value changed. The second function clause matches when anything other than a `User` struct is passed in and returns an error.

```
  def deactivate_user(%User{} = user) do
    # {:ok, Map.put(user, :active, false)}
    # {:ok, Map.merge(user, %{active: false})}
    {:ok, %User{user | active: false}}
  end

  def deactivate_user(_other), do: {:error, "Not a User"}
```

There are multiple ways this function can be written to make the tests pass. Other possible solutions are included as comments. The solution used here has the benefit of using the struct's type to perform the update. Doing this gives compile-time checks on the keys being used. The other options will not. Play around with it get comfortable with the way this works.

Pattern Matching a Function Body: Lists

Pattern matching a list with function clauses is the foundation of recursively processing a list of data. We aren't going to go into recursion here, but instead focus on getting comfortable with the different ways we can Pattern Match a list with function clauses.

Practice Exercises

The following exercises continue using the Pattern Matching project. We will continue focusing on making a single test pass at a time.

The tests we are focusing on are in `test/lists_test.exs`. Running the following command will execute *all* the tests in this file. Running all the tests now will show they all fail.

```
$ mix test test/lists_test.exs

[...]

Finished in 0.1 seconds
7 tests, 7 failures

Randomized with seed 423605
```

Exercise #1 – Lists.is_empty?/1

In Elixir, a function can have the question mark character `?` as part of the name. By convention, this is used to convey that it returns a boolean result. This function works this way as well. If given an empty list, `true` is returned. For anything else it returns `false`.

```
mix test test/lists_test.exs:25
```

Showing Solution

```
  def is_empty?([]), do: true
  def is_empty?(_list), do: false
```

Exercise #2 – Lists.has_1_item?/1

This function also returns a boolean result because of the `?` in the name. If given a list with exactly 1 item, return `true`. For anything else it returns `false`.

```
mix test test/lists_test.exs:35
```

Showing Solution

```
  def has_1_item?([_]), do: true
  def has_1_item?(_list), do: false
```

Exercise #3 – Lists.at_least_one?/1

This function also returns a boolean result because of the `?` in the name. If the list is not empty, return `true`. For anything else it returns `false`.

```
mix test test/lists_test.exs:40
```

Showing Solution

```
  def at_least_one?([_ | _rest]), do: true
  def at_least_one?(_list), do: false
```

Exercise #4 – Lists.return_first_item/1

If the list is not empty, return the first item. If the list is empty, return the atom `:error`.

```
mix test test/lists_test.exs:50
```

Showing Solution

```
  def return_first_item([head | _rest]), do: head
  def return_first_item(_list), do: :error
```

Exercise #5 – Lists.starts_with_1?/1

If the list starts with a value of `1`, then return `true`. Any other initial value returns `false`.

```
mix test test/lists_test.exs:60
```

Showing Solution

```
  def starts_with_1?([1 | _rest]), do: true
  def starts_with_1?(_list), do: false
```

Exercise #6 – Lists.sum_pair/1

If the list has exactly two items, add them together and return the result. If the list doesn't have exactly two items, return the atom `:error`.

```
mix test test/lists_test.exs:70
```

Showing Solution

```
  def sum_pair([first, second]), do: first + second
  def sum_pair(_list), do: :error
```

Exercise #7 – Lists.sum_first_2/1

Given a non-empty list, take the first two elements from it, sum them together and make the summed value be the new head of the list. If the list doesn't have at least two items in it, return the original passed in value.

```
mix test test/lists_test.exs:80
```

Showing Solution
This function uses the `rest` of the list to build the result. All of our other exercises here have ignored the tail of the list. This is part of how we handle recursion. We pull off the head of the list, perform some work and then recursively pass on the `rest` of the list.

```
  def sum_first_2([first, second | rest]), do: [first + second | rest]
  def sum_first_2(list), do: list
```

Pattern Matching a Function Body: Binaries

Elixir has the ability to Pattern Match on binaries. If you recall, a string in Elixir is implemented as a binary type. This lets us do some interesting things. It is also important to understand some of the limitations it has so we can choose the best approach for our problem.

Matching a String Prefix

A string is a binary type. We can match on the beginning of a string like this:

```
defmodule StringTests do

  def match_greeting("Hello " <> subject), do: {:hello, subject}
  def match_greeting("Greetings " <> subject), do: {:greetings, subject}
  def match_greeting("Good morning!"), do: {:morning, nil}
  def match_greeting(_other), do: :unknown

end

StringTests.match_greeting("Hello Tom")
#=> {:hello, "Tom"}

StringTests.match_greeting("Greetings Jane")
#=> {:greetings, "Jane"}

StringTests.match_greeting("Good morning!")
#=> {:morning, nil}

StringTests.match_greeting("Buenos dias")
#=> :unknown
```

This works when we know the *exact* prefix we are looking for. It is case sensitive and *everything* after the match gets bound to the variable. This can be useful when working with a text-based protocol over TCP or UDP. For example, a command strings like the following can be matched and parsed quickly.

- "GET /url/endpoint"
- "SAY Hey guys! How's it going?"
- "POKE friend_user_name"

Matching the Middle or End?

You may think about having the pattern match for the *end* of the string. Let's try that:

```
greeting <> "Tom" = "Hello Tom"
#=> ** (ArgumentError) the left argument of <> operator inside a match should be always a literal binary
as its size can't be verified, got: greeting
```

As you can see, matching at the end isn't allowed. For a binary pattern match to work, it must know the *size* of the pieces being matched, at least in the front. It can match on a known sized beginning and catch the rest of an unknown size at the end.

Matching a Fixed Size String

Binary pattern matching works well for matching and parsing a fixed size string where the structure is already known. Imagine that you have some date values stored in the format `"YYYYMMDD"`. The values are stored as a string. You want to display the date as `"MM/DD/YYYY"`. Binary pattern matching works great here!

Let's look at this example and then we'll break it down.

```
defmodule Formatting do

  def date(<< year::binary-size(4), month::binary-size(2), day::binary-size(2) >>) do
    "#{month}/#{day}/#{year}"
  end

end

Formatting.date("20181230")
#=> "12/30/2018"
```

The pattern match uses the `<< >>` characters to indicate it is a binary type. In this example we define a pattern that breaks the data into 3 chunks using the variable names `year`, `month`, and `day`. With each specifying how large it is.

Defining a pattern like this lets the BEAM perform fast matches. It also lets us "declare" the pattern we want which allows us to elegantly and quickly parse the data. It's just so cool!

Practice Exercises

The following exercises continue using the Pattern Matching project. We continue focusing on making a single test pass at a time.

The tests we are focusing on are in `test/binaries_test.exs`. Running the following command will execute *all* the tests in this file. Running all the tests now will show they all fail.

Remember to focus on the test file as a specification for what the code should do and what the sample inputs look like.

```
$ mix test test/binaries_test.exs

[...]

Finished in 0.06 seconds
7 tests, 7 failures

Randomized with seed 905586
```

Exercise #1 – Binary.identify_command/1

In this exercise you write the function `identify_command/1` that takes a string where the start of the text contains a text-based command. This sort of thing *actually exists* like with the Hypertext Transfer Protocol v1.1 (HTTP 1.1) specification. This is, of course, a dramatically simplified usage and test case.

There are 2 tests for this function. One is the ability to correctly identify the commands we care about and the other handles unsupported commands.

```
mix test test/binaries_test.exs:18
mix test test/binaries_test.exs:22
```

Make the tests pass by using pattern matching in the function declaration.

Showing Solution
We only care about the basic categorization of two commands, "SAY" and "WAVE". Anything else is an unsupported command and results in an error.

```
def identify_command("SAY " <> text), do: {:say, text}
def identify_command("WAVE " <> username), do: {:wave, username}
def identify_command(_other), do: {:error, "Unrecognized command"}
```

Exercise #2 – Binary.format_phone/1

In this exercise you write the function `format_phone/1` that takes a string containing a US-based phone number with no formatting or special characters.

There are 2 tests for this function. One is the ability to correctly parse the input value and return a correctly formatted string. The other handles inputs that don't match.

```
mix test test/binaries_test.exs:30
mix test test/binaries_test.exs:35
```

Make the tests pass by using pattern matching in the function declaration.

Showing Solution
The order doesn't matter for the two versions that match on different sized phone numbers. The pattern is specific enough to only match one or the other. Of course the "I match anything" version must be last.

```
def format_phone(<< area::binary-size(3), three::binary-size(3), four::binary-size(4) >>) do
  "(#{area}) #{three}-#{four}"
end

def format_phone(<< three::binary-size(3), four::binary-size(4) >>) do
  "#{three}-#{four}"
end

def format_phone(other), do: other
```

Exercise #3 – Binary.image_type/1

Binary data can be something *other* than a string. Image files often have a standardized header that describes the file. We can use pattern matching to identify the header data of a file and classify it for us.

In this exercise you write the function `image_type/1` that takes a binary (not a Unicode string), containing some image file signatures.

There are 3 tests for this function. The first two handle correctly identifying a PNG and JPG files. The other test deals with unsupported file signatures.

```
mix test test/binaries_test.exs:48
mix test test/binaries_test.exs:52
mix test test/binaries_test.exs:58
```

Make the tests pass by using pattern matching in the function declaration.

Showing Solution
The order doesn't matter for the two versions that match on specific file signatures. The pattern is specific enough to only match one or the other. Of course the "I match anything" version must be last.

You could also write the binary pattern directly into the function clause. However, using a "Module Attribute" (the `@png_signature`) is helpful to define them all in one place and then reference it in a function clause.

Note that Module Attributes are private to a module. They aren't quite like declaring a constant that can be referenced outside the module. Also, there are some module attributes with special meaning and compiler direction. However, that topic is outside the scope of pattern matching.

```
@png_signature <<137::size(8), 80::size(8), 78::size(8), 71::size(8), 13::size(8), 10::size(8),
                 26::size(8), 10::size(8)>>

@jpg_signature <<255::size(8), 216::size(8)>>

def image_type(<< @png_signature, _rest::binary >>), do: :png
def image_type(<< @jpg_signature, _rest::binary >>), do: :jpg
def image_type(_other), do: :unknown
```

Just to clarify, when you see `<<137::size(8)>>`, the `137` is an exact integer value for the match. There are a lot of options for how to define the binary pattern match. Refer to the documentation for details.

Recap

If you are interested in going deeper on binary pattern matching, then there are many great resources and examples online. The Elixir documentation for defining a bitstring is an excellent reference. There are other great examples of binary pattern matching by others as well. For example, if you want to learn about matching on PNG headers, you can check out articles like this one.

The important thing to remember with binary pattern matching is where it works well and where it doesn't. It works great in these situations:

- matching on a command-style prefix
- matching and unpacking a fixed size string (like for formatting)
- matching and unpacking data from a fixed size binary structure (like a header)

If you want to parse data from the middle or end of a string and there isn't a predictable location for it, then you probably want a Regular Expression. Luckily, you have that available in RegEx! You just can't do that much work in a pattern match function clause. Remember, when pattern matching, the BEAM is trying to answer the question, "Should I execute *this* function clause?"

Binary pattern matching is awesome! Just keep this tool in mind and know that this tool is available to you when you have a suitable problem.

Guard Clauses – Additional Level of Matching

There is an additional level of pattern matching we haven't touched on yet. A "guard clause" can be used in a function clause to further define the pattern for a match.

To define a guard clause, we use the keyword `when`. This is how the function declaration is defined and where the guard clause goes.

```
def function_name(arg1) when guard_clause do
  # function body
end
```

Let's look at a guard clause that checks if the value passed in is an integer.

```
defmodule Testing do
  def greet_integer(value) when is_integer(value), do: "Hello #{value}"
  def greet_integer(_other), do: "meh"
end

Testing.greet_integer(123)
#=> "Hello 123"
Testing.greet_integer("Jim")
#=> "meh"
```

You should note that the `is_integer(value)` guard clause includes the bound variable `value` from the argument. A guard clause can be used on any bound variable to help define more about the pattern you want.

A guard clause expression must evaluate to `true` or `false`. It is being used to determine if the function should be executed.

Guard Clauses and Pattern Matching

Remember there are 3 parts to pattern matching:

1. Match the data type
2. Match the data shape
3. Bind variables to values

Guard clauses can help us with parts 1 & 2.

Guard Clauses Can Help Match *Type*

A guard clause can be used to help match the data's type. Let's say I want to write a function called `to_string/1` that converts a value to a string. Thankfully that already exists, but if we wanted to implement something like it, how could we do that with pattern matching? The problem comes when trying to tell the difference between `1`, `"1"`, `:one`, and `1.0`. Without a guard clause, we can't define a pattern that says an argument must be an integer, string, atom, float, or some other general data type.

This is where a guard clause can help us. Let's look at our `to_string/1` function we could create.

```
defmodule Testing do

  def to_string(value) when is_binary(value), do: value
  def to_string(value) when is_integer(value), do: Integer.to_string(value)
  def to_string(value) when is_atom(value), do: Atom.to_string(value)
  def to_string(value) when is_float(value), do: Float.to_string(value)

end

Testing.to_string("123")
#=> "123"
Testing.to_string(123)
#=> "123"
Testing.to_string(:one)
#=> "one"
Testing.to_string(12.3)
#=> "12.3"
```

Notice that the first pattern match tests for `is_binary(value)`. An Elixir string is a binary, so this test determines there is nothing to do and returns the value as-is.

The other function clauses test the data type using guard clauses. They use `is_integer/1`, `is_atom/1`, and `is_float/1`.

The full list of supported types can be found on the Kernel module. They are the functions that start with `is_*`. Here's a shorter list to give you a convenient idea.

- is_atom(term)
- is_binary(term)
- is_boolean(term)
- is_float(term)
- is_function(term)
- is_integer(term)
- is_list(term)
- is_map(term)
- is_nil(term)
- is_number(term)
- is_tuple(term)

Let's do some practice exercises to play with guard clauses and matching data types.

Type Practice Exercises

The following exercises continue using the Pattern Matching project. We will continue focusing on making a single test pass at a time.

The tests we are focusing on are in `test/guard_clauses_test.exs`. This first set should go pretty fast for you.

Exercise #1 – GuardClauses.return_numbers/1

Given a variety of data inputs, return the value when it is a number. If not a number, return the atom :error .

```
mix test test/guard_clauses_test.exs:20
```

Showing Solution

```
  def return_numbers(value) when is_number(value), do: value
  def return_numbers(_value), do: :error
```

Exercise #2 – GuardClauses.return_lists/1

In previous exercises, you matched lists where they had various patterns like empty, a single item, at least 1 item, etc. There isn't a way to just match "if it is a list" without guard clauses. Now you can!

Given a variety of data inputs, return the value when it is a list. If not a list, return the atom :error .

```
mix test test/guard_clauses_test.exs:35
```

Showing Solution

```
  def return_lists(value) when is_list(value), do: value
  def return_lists(_value), do: :error
```

Exercise #3 – GuardClauses.return_any_size_tuples/1

In previous exercises, matching tuples was very specific to the number of elements in the tuple. Using guard clauses you can now match if it's a tuple of any size!

Given a variety of data inputs, return the value when it is a tuple. If not a tuple, return the atom :error .

```
mix test test/guard_clauses_test.exs:45
```

Showing Solution

```
  def return_any_size_tuples(value) when is_tuple(value), do: value
  def return_any_size_tuples(_value), do: :error
```

Exercise #4 – GuardClauses.return_maps/1

Given a variety of data inputs, return the value when it is a map. If not a map, return the atom :error .

TIP: After you have your solution, **make sure to check out the hidden solution as it has an extra tip on how this can be done!**

```
mix test test/guard_clauses_test.exs:60
```

Showing Solution
In keeping with the focus on guard clauses, this is that version of the solution.

```
  def return_maps(value) when is_map(value), do: value
  def return_maps(_value), do: :error
```

Alternate Solution: Maps are somewhat special in that you can specify a pattern match like the following. It effectively says, "It's a map, but I'm not specifying any keys."

```
  def return_maps(%{} = value), do: value
  def return_maps(_value), do: :error
```

Both approaches are equally valid!

Exercise #5 – GuardClauses.run_function/1

In Elixir, functions are first-class data types. It is common to pass functions as arguments. Using guard clauses you can check that an argument is a function.

Given a variety of data inputs, if the argument is a function, execute the function and return the function's result. If not a function, return the atom `:error` .

```
mix test test/guard_clauses_test.exs:75
```

TIP: When a function is passed as an argument, it is treated the same as an anonymous function. This example code shows how to declare an anonymous function with no arguments that returns the value "Hello!" when executed.

Note: to execute an anonymous function, you use a period after the name. You also must include the parenthesis.

```
greet = fn -> "Hello!" end
greet.()
#=> "Hello!"
```

Showing Solution

```
  def run_function(fun) when is_function(fun), do: fun.()
  def run_function(_fun), do: :error
```

Guard Clauses Can Help Match *Shape*

Guard clauses can also help match the *shape* of your data. A guard clause can be very helpful in defining a less specific shape for a pattern. We've looked at many examples where the pattern defines a shape like matching a specific value.

```
%User{active: true} = data
```

Some operators are safe for guard clauses and let us define a *range* of values to accept in our pattern. Here's a shortened list of some of the most common and helpful operators.

Safe for Guard Clauses

Let's look at a reduced set of the functions and operators you *can* use in guard clauses. We already looked at the "is_*" functions for data types.

- ==
- !=
- +
- -
- *
- /
- <
- <=
- >
- >=
- and
- or
- not
- in

For the full list, see the "Guards" section of Kernel Module documentation.

A particularly interesting operator is the `in` operator. In guard clauses it can only work with ranges (ie `1..5`) and lists. Here's an example of how this can be used in an interesting way.

```
defmodule TestingGuard do
  def place_order(%{status: status} = order) when status in ["pending", "cart"] do
    "Placing order for #{order.customer_id}!"
  end
  def place_order(_order) do
    "Not placing order"
  end
end

order_1 = %{status: "pending", total: 100, customer_id: 10}
order_2 = %{status: "cancelled", total: 75, customer_id: 12}

TestingGuard.place_order(order_1)
#=> "Placing order for 10!"
TestingGuard.place_order(order_2)
#=> "Not placing order"
```

Keep in mind that if you had a list with 100,000 items in it, this could impact your application as the `in` operator stops when the first match is found, but if the value *isn't* in the list, it's an exhaustive search to determine that the function *doesn't* match. But it works great with small, bounded sets.

Likewise, the `not` operator can be very helpful. Expressions like `value not in [1, 2, 3]` and `not is_nil(value)` can be very helpful.

Thinking Tip: Why only a limited set of functions allowed?

Only a reduced set of operators and functions are allowed to be guard clauses. The BEAM will not allow a function call in a guard clause that creates side-effects. Just imagine side-effects like creating a database record, writing to a file, or making an HTTP call to an external service. Those side-effects would be created while trying to decide *if* a function clause should be executed. The function clause may not match, the function body doesn't execute, but the side-effect remains! That would be a *horrible buggy* system!

To be safe, the BEAM only permits a small set of "known safe" functions to be used in guard clauses. This limited set of functions can still do a lot of powerful work for you.

Practice Exercise #6 – GuardClauses.classify_user/1

Let's practice using guard clauses to define a pattern for the *shape* of the data we want.

Given a variety of User structs, we need to classify the user as a legal adult or a minor. For this example we'll use the US definition of age 18 and older to be a legal adult. A User between the ages of 0 and less than 18 is a minor. We should return an error if given a non-user data type, a `nil` age or a negative age. The unit tests cover all these scenarios.

```
mix test test/guard_clauses_test.exs:85
mix test test/guard_clauses_test.exs:92
mix test test/guard_clauses_test.exs:98
mix test test/guard_clauses_test.exs:105
mix test test/guard_clauses_test.exs:110
```

Make sure to check out the solution below after you have your own working code!

Showing Solution
The age `18` has special a business logic meaning here. Because it is referenced multiple times, it makes sense to declare it once as a module attribute and use that for the references.

```
@adult_age 18

def classify_user(%User{age: age} = _user) when is_nil(age) do
  {:error, "Age missing"}
end

def classify_user(%User{age: age} = _user) when age >= @adult_age do
  {:ok, :adult}
end

def classify_user(%User{age: age} = _user) when age >= 0 and age < @adult_age do
  {:ok, :minor}
end

def classify_user(%User{age: age} = _user) when age < 0 do
  {:error, "Age cannot be negative"}
end

def classify_user(_user) do
  {:error, "Not a user"}
end
```

Note that the first function clause matches a `nil` age. The solution here uses a guard clause to do this because that's the topic we're covering here. However, it is more direct and clearer to use a `nil` value in the pattern match. In that version it would look like this:

```
def classify_user(%User{age: nil} = _user) do
  {:error, "Age missing"}
end
```

Custom Guard Clauses

Now is a good time to introduce how you can create your own custom guard clauses. Remember that we are limited in the functions and operations we can use, but we can combine those things together to create helpful, reusable solutions.

Elixir provides a helper command for creating our own custom guard clauses. The `defguard` command looks like this:

```
defguard clause_name(arg1) when guard_clause
```

Let's return to the `GuardClauses.classify_user/1` example and see how we can improve our solution.

```
@adult_age 18

defguard is_adult?(age) when age >= @adult_age
defguard is_minor?(age) when age >= 0 and age < @adult_age

def classify_user(%User{age: age} = _user) when is_adult?(age) do
  {:ok, :adult}
end

def classify_user(%User{age: age} = _user) when is_minor?(age) do
  {:ok, :minor}
end
```

After defining the guard clause, we can use it in a pattern match!

Thinking Tip: When to use custom guard clauses?

The value of a custom guard clause is to prevent duplicating business logic in multiple places.

Custom guard clauses help to clarify the intent of a pattern match. The greatest benefit is when it helps define a pattern that has business logic meaning in your application. *Especially* when you would be repeating that logic in multiple places! If you aren't reusing a guard clause in multiple places, then it may not be adding value.

There is also a `defguardp` command that works similar to `defp`, creating a **private** guard clause that is only available in the defining module.

Import Guard Clauses to Reuse

When you have guard clauses that help define a business logic pattern, you want to be able to reuse them in your application! To do this, you create a module that defines the guard clauses. In our case, like this:

```
defmodule PatternMatching.User.Guards do
  @defmodule """
  Define guard clauses for working with Users.
  """

  @adult_age 18

  defguard is_adult?(age) when age >= @adult_age
  defguard is_minor?(age) when age >= 0 and age < @adult_age
end
```

All that's left is to **import** this module into the modules where we want to use it. Those guard clauses are now available for use in our functions!

```
defmodule PatternMatching.GuardClauses do
  alias PatternMatching.User
  import PatternMatching.User.Guards

  def classify_user(%User{age: age} = _user) when is_adult?(age) do
    {:ok, :adult}
  end

  def classify_user(%User{age: age} = _user) when is_minor?(age) do
    {:ok, :minor}
  end
end
```

Guard Clauses can Match Other Arguments

One powerful aspect of guard clauses is that we can match a piece of data from one argument to another argument's data. Without guard clauses, we couldn't do this in a function clause pattern match. Let's look at an example to help visualize what we are talking about.

```
defmodule Testing do
  def compare_args(arg1, arg2) when arg1 == arg2, do: :equal
  def compare_args(arg1, arg2) when arg1 > arg2, do: :greater_than
  def compare_args(arg1, arg2) when arg1 < arg2, do: :less_than
end

Testing.compare_args(1, 1)
#=> :equal
Testing.compare_args(2, 1)
#=> :greater_than
Testing.compare_args(1, 2)
#=> :less_than
```

Any variable we bind in one argument can be compared to any value bound for another argument! Without guard clauses, this would be done inside a function using an `if` statement. With guard clauses, we are able to keep the filtering of the data on the boundary of the function clause. This lets us keep the function body straight-forward and clear.

Practice Exercise #7 – GuardClauses.award_child_points/3

In this final exercise, we will conditionally award a user additional points if they are within a desired age range. If the user matches the age pattern, increase their points and return an updated user struct. If the user does not match the pattern, return the user unmodified.

There are two tests to focus on.

```
mix test test/guard_clauses_test.exs:125
mix test test/guard_clauses_test.exs:130
```

Showing Solution
The `max_age` is passed in and is compared to the passed in user's age.

```
  def award_child_points(%User{age: user_age} = user, max_age, points) when user_age <= max_age do
    %User{user | points: user.points + points}
  end

  def award_child_points(user, _max_age, _points) do
    user
  end
```

Recap

Guard clauses add another powerful layer to pattern matching in Elixir. We covered a lot here, it's worth taking a moment to mention some of the highlights to keep in mind:

- Guard clauses further define a pattern for data *type* and *shape*.
- Custom guard clauses make reusable business patterns easy to use.
- Guard clauses allow us to define a pattern that combines multiple function arguments.

Additional Resources on Guard Clauses

You can find more resources on guard clauses here:

- https://hexdocs.pm/elixir/master/guards.html – Higher-level documentation on how guards work, what can be used in them, how they fail, etc.
- https://hexdocs.pm/elixir/master/Kernel.html#guards – Kernel module documentation on guard clauses. This is the list of functions and operations defined on the Kernel Module that are safe to be used in guard clauses.
- https://hexdocs.pm/elixir/Kernel.html?#defguard/1 – Documentation on `defguard` command

Pattern Matching Summary

Congratulations on completing the Pattern Matching Course!

Putting it all together

Pattern matching is everywhere in Elixir. If thinking in patterns is a new experience for you, then out of habit you will still be writing code that is more imperative. That's okay! As you spend more time in Elixir, you will see more opportunities to refactor your code to make it more declarative and use patterns more effectively.

A good exercise is to look at some code you just wrote and ask, "Does this *feel* like Elixir code?" If not, look at some examples of a good use of pattern matching and try a small refactor. Ask yourself again, "Does this *feel* like Elixir code?" After just a few iterations it can really begin to change! What really needs to change is not the code, it's the way you *think* about your code. Pattern matching is a new tool you have to solve problems. You need to start thinking about your application as **data**, **functions**, and the **patterns** in your data.

Having gone through the exercises and practice code, you have a good foundation to build on. Let's review some of the things we covered.

- The *pattern* goes on the left of the Match Operator. The data goes on the right.
- A Pattern Match can match the data's *type*, *shape*, and *bind* variables to values all in a single statement.
- A Match Error occurs when no match could be made.
- Pattern matching goes from top to bottom. If the first pattern doesn't match, the next pattern is checked and so on.
- The first pattern to match wins and takes the data.
- Make your top patterns more specific.
- The " ^ " Pin Operator lets you reference the *value* of a variable in a pattern.
- The " _ " lets you define *shape* without binding to the value.
- A nested `if` statement is an anti-pattern
- Lists are "linked lists" and it is cheaper to add to the front than it is to add to the end like an array.
- Lists are recursive, a list is made up of a "head" element and a tail that is itself a list.
- Strings can be pattern matched.
- Guard clauses are another level of a pattern match. They can be also be used to match *type* and *shape*.
- Guard clauses allow you to compare bound variables to each other in a pattern match.
- Pattern matching is most effective when you think differently about your code.

Pattern matching is awesome!

Pattern matching is an *incredible* tool! As with every new tool, there is a learning curve. By completing this coverage of pattern matching, you have dramatically sped up your learning! You have hands-on experience solving problems using this new tool. More important than learning the *mechanics* of pattern matching, you have learned how to *think* about your data and the patterns in your application!

You are on excellent footing now for continued building and growth. You are better able to read and understand Elixir code because pattern matching truly is *everywhere* in Elixir.

I love Elixir because I feel I'm a better developer when I'm working in it. Tools like pattern matching that we covered here feel like a super power! When I work in other languages where these tools don't exist, I really miss it. The more you work with Elixir and pattern matching, the more natural it becomes.

You are ready for the next step and I can't wait to share it with you!

Download reference resource

Now that you have completed the course, as a special "thank you", I want you to have a ready, portable, handy reference as a resource of everything we covered together. This is a PDF download of the course information. It is indexed and searchable so you can easily jump around and find something you want to refer back to.

The Next Step

After learning how pattern matching works in Elixir, it's time to see how pattern matching changes the way we control the ***flow*** of our application code. The Code Flow course covers how **Pattern Matching**, **Immutability** and **Functional Programming** impacts the foundation of programming:

- Branching logic
- Looping
- Error handling

There are new control flow patterns to learn, new language features available, and new ways to do familiar things.

More than learning *what* the new things are and *how* to use them, you also learn *when* to use a technique and *why* to choose one approach over another. You get hands-on experience with the concepts using a downloadable project with practice exercises.

You gain experience, understanding, and confidence in your ability to make meaningful contributions to a project and doing it the "Elixir way". Even more than now, you will be "Thinking Elixir".

> See the available courses and keep your momentum going!
