Pattern Matching

- One of the most important components of elixir is pattern matching
- Pattern matching allows you to have multiple functions of the same name but will choose the correct function based off of the data type
- In other programming languages this = operator is an assignment operator. In Elixir however this is the match operator.

In elixir when we say `x = 1` it also means that `1 = x` this is much different from an assignment of x = 1.

Let's look at this example

```
iex> x = 10
10
iex> 10 = x
10
iex> 1 = x
** (MatchError) no match of right hand side value: 10
```

- `10 = x` is good comparrison but when both sides don't match a MatchError is raised because a variable can only be assigned on the left side. So with `1 = x` it raised a MatchError due to the fact that we aren't assigning 1 to x so it is seeing if x which is equal to 10 matches with 1

- Patern matching becomes very useful for tools such as breaking down complex data types

```
elixir
iex> {a, b, c} = {1, 2, :atom}
{1, 2, :atom}
iex> a
1
iex> b
2
iex> c
:atom
```

In Section 5 we used pattern matching in our list function to use our head and tail of our list

```
defp do_sum([head | tail], sum) do
    IO.puts(sum)
    do_sum(tail, sum + head)
  end
```

This recursive function splits the list into a head and tail and then calls the function again with just the tail.

A nice use of pattern matching in elixir is gaurd clauses
gaurd clauses allow for simpler condition then if else statements which can create layers of complexity. Here is an example of an add function which takes only integer values.

```
iex(3)> defmodule Add do
...(3)> def sum(x,y) when x |> is_integer() and y |> is_integer() do
...(3)> x+y
...(3)> end
...(3)> end

iex(4)> Add.sum(2,3)
5
iex(5)> Add.sum(2,3.0)
** (FunctionClauseError) no function clause matching in Add.sum/2

    The following arguments were given to Add.sum/2:

        # 1
        2

        # 2
        3.0

    iex:4: Add.sum/2
```

lets go back to our fib.ex file and add gaurd clauses so that we only except positive integers because our function will break if we input negaitve integers

```
defmodule Fib do

  def fib(0), do: 0
  def fib(1), do: 1

  def fib(n) when n |> is_integer and n > 1, do: fib(n-1) + fib(n-2)
end
```

Now our function will only except positive integer values otherwise it will raise a no function matching error.