Modules and Functions

Let's put some code in a file.

```
# Directory: basics
# File: 00math.ex (create new)

defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

Start the interpreter

```
iex
```

Let's load the file

```
c("00math.ex")      # Reload after every change to file
Math.sum(1, 2)
```

Run as Script

```
# file 01mathscript.exs
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

Execute

```
elixir mathscript.exs
```

Functions in Modules

We've already seen a function in a module. Let's take a look at some other options.

```
# file 02math.exs
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end
```

```
  defp do_sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
IO.puts Math.do_sum(1, 2)
# Run it with elixir 02math.exs
```

The following error is produced:

```
** (UndefinedFunctionError) function Math.do_sum/2 is undefined or private
    Math.do_sum(1, 2)
    02math.exs:12: (file)
    (elixir) lib/code.ex:767: Code.require_file/2
```

Thats because `do_sum/2` is defined as private since it has `defp` but it can be called from within the module.

Pattern Matching in Function Definitions

This migth be new to you if you have not had much experience with functional programming before. Let's take a look

```
# file 03math.exs
defmodule Math do
  def div(_, 0) do
    {:error, "Bad argument"}
  end

  def div(x, y) do
    {:ok, "Value is #{x/y}"}
  end

end

IO.inspect Math.div(1, 0)
IO.inspect Math.div(1, 2)
```

Matching starts from the top and goes down. So, if you move the second `def` before the first, you'll get an error on the first function call.

Also, if you remove the second `def`, you will get an error on the second call – `FunctionClauseError` – because no matching clause will be found for this call.

Guards

Another way of adding conditions to function definitions for specific cases is through guards.

```
# File: 04math.exs
defmodule Math do
  def zero?(0), do: true
  def zero?(x) when is_integer(x), do: false
  def zero?(x), do: "Can't!"
end

IO.puts Math.zero?(0)   # ? is by convention
IO.puts Math.zero?(2)
IO.puts Math.zero?("Me")
```

Anonymous Functions

Now that we've seen basic function definitions, we can take a look at anonymous functions.

Start up the interpreter using `iex`.

```
add = fn (a, b) ->
        a + b
      end

add.(3, 6)
```

This looks weird for several reasons. * instead of `def`, we have an `fn` * instead of `do`, we have a `->` * when calling, we add a `.` before the `(`

This is because anonymous functions are treated a little differently by elixir but for now, these three differences should be sufficient.

Default Arguments

When you want to make certain arguments optional. For this, you use the `\\` operator.

```
# File: 06concat.exs

defmodule Concat do
  def join(a, b, sep \\ " ") do
    a <> sep <> b      # String concatanation
  end
end
```

```
IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

You can also do this using clause matching but let's leave that as an exercise.

The Famous Pipe Operator

Let's consider a usecase in which we want to take a list as input and find the square of the sum of it's tail.

So, `sst([1, 2, 4])  = square(sum(2, 4))`

So, the typical way of writing this would be as follows:

```
a = [1, 2, 4]

a = tail(a)
a = sum(a)
a = square(a)
```

Or, we could write it in one line as:

```
a = square(sum(tail(a)))
```

The problem with the first one is that it's too verbose. It's also going to get a lot messier if we have more arguments that go into the functions.

The second one is just messy since we have to start in the middle and read it backwards.

Let's see the elixir way:

```
# File: 06pipe.exs
defmodule PipeTest do
   def square(x), do: x * x

   def sum(l, acc) do
     acc + Enum.sum(l)
   end


   def sst(the_list) do
      the_list
         |> tl           # Builtin
         |> sum(0)       # Pass the second argument
         |> square
   end

end
```

```
# IO.puts PipeTest.square(5)
# IO.puts PipeTest.sum([1, 2, 5], 0)

IO.puts PipeTest.sst([1, 2, 3])

# Run it ...
# Now try putting IO.inspect between pipes!
```

This will be of extreme importance in any proper Elixir app – especially Phoenix!

"Elixir and Phoenix: Real World Functional Programming"

Video Course by Dr. Nauman

`http://recluze.net/learn`