Curso Elixir

Sección 7

Módulos y funciones

En Elixir un módulo es la agrupación de varias funciones. Para poder crear nuestros propios módulos utilizaremos el macro defmodule y def para crear nuevas funciones dentro del módulo.

```
defmodule Math do
  def mul(x, y) do
    x * y
  end

  def sum(x, y), do: x + y
end
```

Compilación

Normalmente es conveniente escribir módulos y funciones en archivos para poder compilarlos y reutilizarlos.

Crea un archivo math.ex

```
defmodule Math do
  def mul(x, y) do
    x * y
  end

  def sum(x, y), do: x + y
end
```

Ahora compila el archivo con elixirc

```
$ elixirc math.ex
```

Si no existen errores al compilar el archivo, se generará un archivo .beam (Elixir.Math.beam), al entrar de nuevo a iex (estando en la misma ruta del archivo .beam) podremos usar el modulo definido en math.ex

```
iex> Math.sum(2, 9)
```

Los proyectos en elixir comúnmente están divididos en 3 directorios:

- ebin (compilados)
- lib (código elixir)
- test (comunmente archivos .exs)

Modo secuencia de comandos

Además de la extensión .ex, elixir también soporta .exs, elixir trata esas dos extensiones exactamente igual al diferencia es el uso, los archivos con extensión .ex se compilan y las .exs son secuencias de comandos

Crea un archivo math.exs

```
defmodule Math do
  def mul(x, y) do
    x * y
  end

  def sum(x, y), do: x + y
end

IO.inspect(Math.sum(2, 2))
```

ejecútalo

```
$ elixir math.exs
```

Funciones con nombre

- def/2 y defp/2

  Dentro de los módulos podemos definir funciones con def o funciones privadas con defp, la diferencia de estas dos es que las funciones defp solo pueden ser invocadas localmente.

```
defmodule Math do
  def mul(x, y) do
    x * y
  end

  defp sum(x, y), do: x + y
end

IO.inspect(Math.mul(2, 2))  #=> 4
IO.inspect(Math.sum(2, 2))  #=> ** (UndefinedFunctionError)
```

- Guardias

  La declaración de funciones puede tener guardias y varias clausulas

```
defmodule Math do
  def greater_than_zero?(0), do: false
  def greater_than_zero?(x) when x > 0, do: true
end

Math.greater_than_zero?(4)
Math.greater_than_zero?(0)
Math.greater_than_zero?(-1) #=> ** (FunctionClauseError) no function clause matching in Math.greater_than_z
```

El signo de interrogación final en la definición de una función significa que regresara un valor boolenano.
Dando un argumento que no coincida con ninguna cláusula se lanzara un error.

Captura de funciones

Inicia iex corriendo math.ex

```
$ iex math.ex
iex> Math.mul(1, 2)
iex> fun = &Math.sum/2
&Math.sum/2
iex> is_function(fun)
true
iex> fun.(3, 4)
7
```

Recuerda que Elixir hace una distinción entre funciones anónimas y funciones con nombre, donde las primeras deben invocarse con un punto entre el nombre de la variable y los paréntesis.

Tenga en cuenta que la sintaxis de captura también se puede utilizar como acceso directo para crear funciones:

```
iex> fun = &(&1 + &2)
&:erlang.+/2
iex> fun.(2, 1)
3
```

&1 y &2 representa a los parámetros que pasarán a la función, esto sería igual

```
iex> fun = fn x, y -> x + y end
```

Para capturar la función de un módulo, lo puedes hacer:

```
iex> fun = &Enum.count(&1)
iex> fun.([1,2,3,4,5])
5
```

Argumentos por defecto

Las funciones nombradas soportan argumentos por defecto

```
defmodule Math do
  def mul(x, y \\ 0) do
    x * y
  end
end

iex> Math.mul(9)
0
iex> Math.mul(9, 1)
9
```

Si una función con valores predeterminados tiene varias cláusulas, es necesario crear un encabezado de función (sin un cuerpo real) para declarar valores predeterminados:

```
defmodule Helper do
  def concat(x, y \\ nil, s \\ " ")
  def concat(x, y, _s) when is_nil(y), do: x
  def concat(x, y, s), do: x <> s <> y
end

iex> Helper.concat("uno", "dos")
```

```
iex> Helper.concat("uno", "dos", ",")
iex> Helper.concat("uno")
```

Actividad 4

Corrige la función Math.greater_than_zero?(-1) #=> ** (FunctionClauseError) no function clause matching in Math.greater_than_zero?/1