Curso Elixir

Sección 2

Tipos básicos

En esta sección aprenderemos más sobre los tipos básicos de Elixir: enteros, flotantes, booleanos, átomos, cadenas, listas y tuplas. Algunos tipos básicos son:

```
iex> 1          # integer
iex> 0x1F       # integer
iex> 1.0        # float
iex> true       # boolean
iex> :atom      # atom / symbol
iex> "elixir"   # string
iex> [1, 2, 3]  # list
iex> {1, 2, 3}  # tuple
```

Aritmética básica

```
iex> 1 + 2
3
iex> 5 * 5
25
iex> 10 / 2
5.0
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

Elixir también admite anotaciones de acceso directo para ingresar números binarios, octales y hexadecimales

```
iex> 0b1010
10
iex> 0o777
511
iex> 0x1F
31
```

Los números flotantes requieren un punto seguido de al menos un dígito y también admiten e para notación científica:

```
iex> 1.0
1.0
iex> 1.0e-10
1.0e-10

iex> round(3.58)
4
iex> trunc(3.58)
3
```

Identificación de funciones y documentación

Las funciones en Elixir se identifican tanto por su nombre como por su arity. La arity de una función describe el número de argumentos que toma la función.

```
iex> h round/1
                              def round()

Rounds a number to the nearest integer.

iex> h <>/2
                             defmacro left <> right
```

Booleanos

Elixir soporta los valores boleanos true y false

```
iex> true
true
iex> true == false
false
```

Elixir proporciona un montón de funciones de predicado para verificar un tipo de valor. Por ejemplo, is_boolean/1

```
iex> is_boolean(true)
true
iex> is_boolean(1)
false
```

Otras funciones similares: is_integer/1, is_float/1 or is_number/1

Atoms

Un átomo es una constante cuyo valor es su propio nombre. Algunos otros idiomas llaman a estos símbolos. A menudo son útiles para enumerar valores distintos, como:

```
iex> :apple
:apple
iex> :orange
:orange
iex> :watermelon
:watermelon
```

A menudo se utilizan para expresar el estado de una operación, mediante el uso de valores como :ok y :error

los boleanos true y false también son atoms

```
iex> true == :true
true
iex> is_atom(false)
true
iex> is_boolean(:false)
true
```

Elixir te permite omitir ":" para los atoms false,, true y nil

Finalmente, Elixir tiene una construcción llamada alias que exploraremos más adelante. Los alias comienzan en mayúsculas y también son átomos:

```
iex> is_atom(Hello)
true
```

Funciones anónimas

Las funciones anónimas nos permiten almacenar y pasar código ejecutable como si fuera un entero o una cadena. Están delimitados por las palabras clave fn y end

```
iex> add = fn x, y -> x + y end
iex> add.(5, 3)
iex> is_function(add)
```

Los argumentos entre paréntesis después de la función anónima indican que queremos que se evalúe la función, no solo que se devuelva su definición. Tenga en cuenta que se requiere un punto (.) Entre la variable y los paréntesis para invocar una función anónima. El punto asegura que no haya ambigüedad entre llamar a la función anónima que coincide con una variable o una función con nombre add / 2.

Finalmente, las funciones anónimas también son cierres y, como tales, pueden acceder a variables que están dentro del alcance cuando se define la función.

```
iex> double = fn a -> add.(a, a) end
iex> double.(2)
```

Una variable asignada dentro de una función no afecta a su entorno:

```
iex> x = 7
iex> (fn -> x = 0 end).()
iex> x
7
```

Lists

Elixir usa corchetes para especificar una lista de valores. Los valores pueden ser de cualquier tipo:

```
iex> [1, 2, true, 3]
iex> length [1, 2, 3]
```

Dos listas se pueden concatenar o restar usando los operadores ++/ 2 y -- /2 respectivamente

```
iex> [1, 2, 3] ++ [4, 5, 6]
iex> [1, true, 2, false, 3, true] -- [true, false]
```

Los operadores de listas nunca modifican la lista existente. Concatenar o eliminar elementos de una lista devuelve una nueva lista. Decimos que las estructuras de datos de Elixir son inmutables. Una ventaja de la inmutabilidad es que conduce a un

código más claro.

Tail y Head, son funciones muy comunes para el acceso a listas

```
iex> list = [1, 2, 3]
iex> hd(list)
1
iex> tl(list)
[2, 3]
```

En ocasiones al crear una lista se devolverá un valor entre comillas simples. Por ejemplo:

```
iex> [11, 12, 13]
'\v\f\r'
iex> [104, 101, 108, 108, 111]
'Hello'
```

Cuando Elixir ve una lista de números ASCII imprimibles, Elixir lo imprimirá como una lista de caracteres (literalmente, una lista de caracteres).
Las listas de caracteres son bastante comunes al interactuar con códigos Erlang.

```
iex> i 'hello'
```

Las comillas simples son charlists, las comillas dobles son cadenas.

```
iex> 'hello' == "hello"
false
```

Tuples

Elixir usa llaves para definir tuplas. Al igual que las listas, las tuplas pueden contener cualquier valor:

```
iex> {:ok, "hello"}
:ok, "hello"}

iex> tuple_size {:ok, "hello"}
2
```

Las tuplas almacenan elementos contiguos en la memoria. Esto significa que acceder a un elemento de tupla por índice u obtener el tamaño de la tupla es una operación rápida. Los índices comienzan desde cero:

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}

iex> elem(tuple, 1)
"hello"

iex> tuple_size(tuple)
2
```

También es posible poner un elemento en un índice particular en una tupla con

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}

iex> put_elem(tuple, 1, "world")
{:ok, "world"}

iex> tuple
{:ok, "hello"}
```

Cada operación en una tupla devuelve una nueva tupla, nunca cambia la dada.

List o Tuples

Las listas se almacenan en la memoria como listas enlazadas, lo que significa que cada elemento de una lista mantiene su valor y apunta al siguiente elemento hasta llegar al final de la lista. Esto significa que acceder a la longitud de una lista es una operación lineal: debemos recorrer toda la lista para determinar su tamaño.

Del mismo modo, el rendimiento de la concatenación de listas depende de la longitud de la lista de la izquierda:

```
iex> list = [1, 2, 3]
#Esto es rápido ya que solo necesitamos pasar por `[0]` para agregar 0
iex> [0] ++ list
[0, 1, 2, 3]
#Esto es lento ya que necesitamos atravesar todo `list` para agregar 4
iex> list ++ [4]
```

Las tuplas, por otro lado, se almacenan contiguamente en la memoria. Esto significa que obtener el tamaño de tupla o acceder a un elemento por índice es rápido. Sin embargo, actualizar o agregar elementos a las tuplas es costoso porque requiere crear una nueva tupla en la memoria:

```
iex> tuple = {:a, :b, :c, :d}
iex> put_elem(tuple, 2, :e)
{:a, :b, :e, :d}

iex> Tuple.append(tuple, :e)
{:a, :b, :c, :d, :e}
```

Tenga en cuenta que esto solo se aplica a la tupla en sí, no a su contenido. Por ejemplo, cuando actualiza una tupla, todas las entradas se comparten entre la tupla antigua y la nueva, excepto la entrada que ha sido reemplazada. En otras palabras, las tuplas y las listas en Elixir son capaces de compartir sus contenidos. Esto reduce la cantidad de asignación de memoria que necesita realizar el idioma y solo es posible gracias a la semántica inmutable del idioma.

Esas características de rendimiento dictan el uso de esas estructuras de datos. Un caso de uso muy común para las tuplas es usarlas para devolver información adicional de una función. Por ejemplo, File.read/1 es una función que se puede usar para leer el contenido del archivo. Devuelve una tupla

```
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> File.read("path/to/unknown/file")
{:error, :enoent}
```

Actividad 2

- Ejecutar los comandos vistos en esta sección.
- Crear función anónima para calcular el cuadrado de un número