Los tipos básicos en elixir son enteros, flotantes, booleanos, átomos, cadenas, listas y tuplas.

```
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

Las operaciones básicas son comunes:

```
iex> 1 + 2
3
iex> 2 - 1
1
iex> 5 * 5
25
iex> 10 / 2
5.0
```

Para que al dividir devuelva un entero en vez de un flotante se usa la función div/1 y para el modulo usamos rem/1.

```
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

Elixir le permite quitar los paréntesis al invocar funciones con nombre. Esta característica proporciona una sintaxis más limpia.

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
```

Las flotantes son de doble precisión de 64 bits.

La función «round» devuelve el número entero más cercano a un flotante y la función «trunc» para obtener la parte entera de un flotante.

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

Booleanos

Tenemos las palas claves true y false:

```
iex> true
```

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

También tenemos la función «is_boolean»:

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

Atoms

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

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

Los átomos son iguales si sus nombres son iguales:

```
iex> :apple == :apple
true
iex> :apple == :orange
false
```

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

Los booleanos verdadero y falso también son átomos:

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

Elixir te permite omitir el líder: para los átomos falso, verdadero y nulo.

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
```

Strings

Las cadenas están delimitadas por comillas dobles, y están codificadas en UTF-8:

```
iex> "hellö #{:world}"
"hellö world"
```

Las cadenas pueden tener saltos de línea en ellas. Puedes presentarlos usando secuencias de escape:

```
ex> "hello
...> world"
"hello\nworld"
iex> "hello\nworld"
"hello\nworld"
```

Se puede imprimir una cadena usando la función IO.puts/1 desde el módulo IO:

```
iex> IO.puts "hello\nworld"
hello
world
:ok
```

Podemos ver que la función IO.puts/1 devuelve el átomo :ok después de imprimir.

Las cadenas están representadas internamente por secuencias contiguas de bytes conocidas como binarios:

```
iex> is_binary("hellö")
true
```

Para saber el número de bytes en una cadena:

```
iex> byte_size("hellö")
6
```

Observamos que el número de bytes en esa cadena es 6, a pesar de que tiene 5 caracteres. Esto se debe a que el carácter «ö» toma 2 bytes para ser representado en UTF-8. Podemos obtener la longitud real de la cadena, en función del número de caracteres, utilizando la función String.length/1:

```
iex> String.length("hellö")
5
```

El String module contiene un montón de funciones para operar:

```
iex> String.upcase("hellö")
"HELLÖ"
```

Funciones anónimas

Estas funciones 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 a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> add.(1, 2)
3
iex> is_function(add)
true
```

Las funciones anónimas también se identifican por la cantidad de argumentos que reciben. Podemos verificar si una función usando is_function / 2:

```
iex> is_function(add, 2)
true

iex> is_function(add, 1)
false
```

Definamos una nueva función anónima que usa la función agregar anónimo que hemos definido previamente:

```
iex> double = fn a -> add.(a, a) end
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> double.(2)
4
```

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

```
iex> x = 42
42
iex> (fn -> x = 0 end).()
0
```

```
iex> x
42
```

Listas

Se usan corchetes para especificar una lista de valores. Los valores pueden ser de cualquier tipo:

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

Se pueden concatenar o restar dos listas utilizando los operadores ++ / 2 y – / 2 respectivamente:

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

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. Puede pasar libremente los datos con la garantía de que nadie los mutará en la memoria, solo los transformará.

La cabeza es el primer elemento de una lista y la cola es el resto de la lista. Se pueden recuperar con las funciones hd/1 y tl/1. Asignemos una lista a una variable y recuperemos su cabeza y cola:

```
iex> list = [1, 2, 3]
```

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

Obtener la cabeza o la cola de una lista vacía arroja un error:

```
iex> hd []
** (ArgumentError) argument error
```

A veces creará una lista y 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 el código Erlang existente. Siempre que vea un valor en IEx y no esté seguro de cuál es, puede usar el i/1 para recuperar información al respecto:

```
iex> i 'hello'
Term
  'hello'
Data type
  List
Description
  ...
Raw representation
  [104, 101, 108, 108, 111]
Reference modules
  List
Implemented protocols
  ...
```

Las representaciones con comillas simples y dobles no son equivalentes en Elixir, ya que están representadas por diferentes tipos:

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

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

Tuplas

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 put_elem/3:

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

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

Observamos que put_elem/3 devolve una nueva tupla. La tupla original almacenada en la variable tupla no se modificó. Al igual que las listas, las tuplas también son inmutables. Cada operación en una tupla devuelve una nueva tupla, nunca cambia la dada.

¿Listas o tuplas?

Las listas se almacenan en la memoria como listas vinculadas, 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]

iex> [0] ++ list
[0, 1, 2, 3]

iex> list ++ [4]
[1, 2, 3, 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}
```

Hay que tener 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}
```

Si la ruta dada a File.read/1 existe, devuelve una tupla con el átomo :ok como el primer elemento y el contenido del archivo como el segundo. De lo contrario, devuelve una tupla con :error y la descripción del error.

La mayoría de las veces, Elixir te guiará para hacer lo correcto. Por ejemplo, hay una función elem/2 para acceder a un elemento de tupla, pero no hay un equivalente incorporado para las listas:

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
```

Hasta ahora hemos utilizado 4 funciones de conteo: byte_size/1 (para el número de bytes en una cadena), tuple_size/1 (para el tamaño de la tupla), length/1 (para la longitud de la lista) y

String.length/1 (para el número de grafemas en una cadena). Usamos byte_size para obtener el número de bytes en una cadena, una operación barata. Recuperar el número de caracteres Unicode, por otro lado, usa String.length, y puede ser costoso ya que depende de un recorrido de toda la cadena.

Elixir también proporciona puertos, referencias y PID como tipos de datos (generalmente utilizados en la comunicación de procesos).