Tipos de datos:

Elixir utiliza el mismo sistema de tipos de Erlang.

Números:

```
iex> is_number(3)
true
iex> 3.5
3.5
iex> is_number(3.5)
true
```

Los números (numbers) pueden ser enteros o flotantes.

Integer y float:

```
iex> is_integer(3)
true
iex> is_float(3)
false
```

```
iex> is_integer(3.5)
false
iex> is_float(3.5)
true
```

Notación científica:

```
iex> 3.25555e-3
0.00325555
iex> 3.25555e3
3255.55
iex>i 3.25555e3
Term
  3255.55
```

Operaciones aritméticas:

```
iex(1)> 5*4
20
iex(2)> 5*4/3
6.666666666666667
iex(3)> 3+2-5
0
iex(4)> 5*4/3+2-5
3.666666666666668
iex(5)> 5/5
1.0
iex(6)> 5/4
1.25
iex(7)> div(5,5)
1
iex(8)> rem(5,5)
0
iex(9)>
```

-Piso de un número flotante.

-Techo de un número.

```
iex(9)> trunc(5/2)
2
iex(10)> floor(5/2)
2
iex(11)>
```

```
iex(11)> round(5/2)
3
iex(12)> ceil(5/2)
3
iex(13)>
```

-Números Binarios.

(0b es para señalar la

conversión binaria).

```
iex(13)> 0b10101001111
1359
iex(14)> 010101001111
10101001111
```

-Números Octales.

(0o es para señalar la conversión

octal).

```
iex(15)> 0o74754
31212
iex(16)> 0o2541
1377
```

-Números Hexadecimales. (0x es para señalar la conversión hexadecimal).

```
iex(17)> 0xFFFF
65535
```

-Azúcar Sintáctica para los números. // ¿Por qué azúcar?

```
iex(18)> 1_000_000
1000000
iex(19)> 1_000_000.123
1000000.123
```

Atoms (son constantes):

-Constantes literales nombradas.

-Es una constante cuyo nombre es su propio valor.

-Inician con :(dos puntos).

-Seguidos de caracteres alfanuméricos y/o subrayados.

-Se pueden usar espacios en blanco si se ponen entre comillas

```
iex(24)> :atom
:atom
iex(25)> is_atom(:atom)
true
iex(26)> is_atom(:es_un_atom)
true
iex(27)> is_atom(:"es un atom")
true
iex(28)> i :ok
Term
  :ok
Data type
  Atom
Reference modules
  Atom
Implemented protocols
  IEx.Info, Inspect, List.Chars, String.Chars
iex(29)>
```

Un atom consta de dos partes:

```
iex(31)> var_atom = :atom
:atom
iex(32)> var_atom
:atom
iex(33)> :atom = var_atom
:atom
iex(34)> :atom
:atom
iex(35)>
```

-Texto: el que se pone después de los dos puntos.

-Valor: es la referencia a la tabla de atoms.

Un atom se puede nombrar con mayúscula inicial.

```
iex(35)> is_atom(Un_atom)
true
iex(36)> Un_atom = Elixir.Un_atom
Un_atom
iex(37)> is_atom = Elixir.Un_atom
Un_atom
iex(38)> is_atom
Un_atom
iex(39)> Un_atom
Un_atom
```

(Sin necesidad de los ":" al parecer).

Atomos como booleanos

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

(Los valores booleanos son atoms).

Atoms and, or y not:

```
iex> true and true
true
iex> true and false
false
iex> true or true
true
iex> true or false
true
```

```
iex(40)> not false
true
iex(41)> not true
false
iex(42)> not not true
true
iex(43)> not not false
false
```

Nil:

El null del lenguaje Elixir.

```
iex> is_atom(nil)
true
iex> is_atom(:nil)
true
iex> nil == :nil
true
```

Los atoms nil y false son tratados como valores falsos, mientras que todo lo demás es tratado como un valor de verdad.

Esta propiedad es útil con los operadores corto circuito:

```
iex> false || nil || 5 || true
5
iex> false || nil || 5 || false || true
5
iex> false || nil || false || false || true || 5
true
```

-|| -> retorna la primera expresión verdadera.

```
iex> false && 5
false
iex> nil && 5
nil
iex> true && 5
5
iex> true && true
true
iex> 5 && true
true
```

-&& -> retorna la segunda siempre y cuando la primera lo sea también.

```
iex> !true
false
iex> !false
true
iex> !5
false
iex> !nil
true
iex> not !4
true
iex> !(5+4)
false
iex> not(5+4)
** (ArgumentError) argument error
    :erlang.not(9)
```

-! -> retorna la negación de la expresión sin importar el tipo de dato.

Tuplas:

Son como estructuras o registros.

Permiten agrupar elementos fijos.

```
iex>persona = {"Alex", 49}
{"Alex", 49}
```

```
iex> nombre = elem(persona, 0)
"Alex"
iex> nombre
"Alex"
iex> edad = elem(persona,1)
49
```

-Para extraer elementos se usa la función *elem*.

Las tuplas son inmutables, por lo que no se modifica.

Para modificar un elemento se usa la función put_elem. Almacenando el cambio en otra variable, o en la misma si ya no se desea conservar los valores.

```
iex> persona = put_elem(persona,0,"Alexander")
{"Alexander", 49}
iex> persona
{"Alexander", 49}
```

Listas:

-Manejo dinámico de datos.

-Funcionan como listas enlazadas simples.

```
iex> numeros_pares = [2,4,6,8,10]
[2, 4, 6, 8, 10]
```

```
iex> length(numeros_pares)
5
```

Obtener un elemento de la lista mediante la función *Enum.at*.

```
iex> Enum.at(numeros_pares,4)
10
iex> Enum.at(numeros_pares,5)
nil
```

```
iex> 2 in numeros_pares
true
iex> 12 in numeros_pares
false
```

Se puede saber si x elemento pertenece a una lista con operador *in*.

Módulo List:

-Modificar o reemplazar un elemento de la lista.

```
iex> numeros_pares
[2, 4, 6, 8, 10]
iex> nuevos_pares = List.replace_at(numeros_pares,4,12)
[2, 4, 6, 8, 12]
```

Insertar un elemento.

```
iex> numeros_pares
[2, 4, 6, 8, 12]
iex> numeros_pares = List.insert_at(numeros_pares,4,10)
[2, 4, 6, 8, 10, 12]
iex> numeros_pares = List.insert_at(numeros_pares,-1,14)
[2, 4, 6, 8, 10, 12, 14]
```

Concatenar dos listas.

```
iex> numeros_naturales = [1,2,3,4] ++ [5,6,7,8]
[1, 2, 3, 4, 5, 6, 7, 8]
iex> numeros_naturales
```

Recursión sobre listas.

```
iex> []
[]
iex> [1|[]]
[1]
iex> [1|[2|[]]]
[1, 2]
iex> [1|[2|[3|[]]]]
[1, 2, 3]
iex> [1|[2|[3|[4|[]]]]]
[1, 2, 3, 4]
iex> [1|[2,3,4]]
[1, 2, 3, 4]
```

-El formato de una lista es [head | tail].
-Head puede ser de cualquier tipo.
-Tail siempre es una lista.
-Si tail es una lista vacía [], indica que es el final de la lista.

Funciones head (hd) y tail (tl).

```
iex> numeros = [1,2,3,4,5]
[1, 2, 3, 4, 5]
iex> hd(numeros)
1
iex> tl(numeros)
[2, 3, 4, 5]
```

```
iex(45)> [head | tail] = numeros
[1, 2, 3, 4, 5]
iex(46)> head
1
iex(47)> tail
[2, 3, 4, 5]
```

Agregar elementos a una lista:

```
iex> numeros = [0 | numeros]
[0, 1, 2, 3, 4, 5]
```

Mapas:

Par llave-valor, pueden ser cualquier término.

```
iex(48)> persona = %{:nombre => "Eduardo", :edad => 19, :trabajo => "No tiene"}
%{edad: 19, nombre: "Eduardo", trabajo: "No tiene"}
iex(49)> persona
%{edad: 19, nombre: "Eduardo", trabajo: "No tiene"}
iex(50)>
```

Llave = :nombre, valor = “Eduardo”.

Otra forma de representar los mapas:

```
iex(50)> %{nombre: "Eduardo", edad: 19, trabajo: "No tiene"}
%{edad: 19, nombre: "Eduardo", trabajo: "No tiene"}
```

Acceder a un elementro a través de su llave:

```
iex(51)> persona = %{:nombre => "Eduardo", :edad => 19, :trabajo => "No tiene"}
%{edad: 19, nombre: "Eduardo", trabajo: "No tiene"}
iex(52)> persona[:nombre]
"Eduardo"
iex(53)> persona[:apellido]
nil
iex(54)> persona[:edad]
19
```

Ventajas de usar atoms como llave:

```
iex(55)> persona.nombre
"Eduardo"
iex(56)> persona.edad
19
```

Insertar un nuevo llave-par:

```
iex(57)> Map.put(persona, :apellido, "Gonzalez")
%{apellido: "Gonzalez", edad: 19, nombre: "Eduardo", trabajo: "No tiene"}
iex(58)>
iex(58)> Map.get(persona, :nombre)
"Eduardo"
iex(59)> persona.nombre
"Eduardo"
iex(60)> persona[:nombre]
"Eduardo"
```

Obtener el valor de una llave con Map.7

Binaries:

```
iex(14)> <<1,2,3,4,5>>
<<1, 2, 3, 4, 5>>
iex> <<255>>
<<255>>
iex> <<256>>
<<0>>
iex> <<257>>
<<1>>
iex> <<512>>
<<0>>
```

-Un binary es un grupo de bytes.
-Cada número representa un valor que corresponde a un byte.
-Cualquier valor mayor a 255 se trunca al valor en byte.

Strings (Binary Strings):

-No existe un tipo String dedicado.

-Los Strings se representan como binary o list.
-Lo más sencillo es usar dobles comillas.
-Se pueden insertar expresiones en las cadenas (interpolación de cadenas) mediante #{}.

```
iex> "El cuadrado de 2 es #{2*2}"
"El cuadrado de 2 es 4"
```

Secuencias de escape:
– "
– \"
– \t

```
IO.puts("1. Este es un mensaje")
IO.puts("2. Este es un  \n mensaje")
IO.puts("3. Este es un \"mensaje\"")
IO.puts("4. Este es un \\mensaje\\")
IO.puts("5. Este \t es \tun \t mensaje")
IO.puts("4. Este
es un
mensaje")
```

```
1. Este es un mensaje
2. Este es un
 mensaje
3. Este es un "mensaje"
4. Este es un \mensaje\
5. Este          es     un       mensaje
4. Este
        es un
        mensaje
```

Sigils:

```
IO.puts(~s("este es un ejemplo de sigil" apuntes de Elixir))
IO.puts("Este \t es \tun \t mensaje")
IO.puts(~S("Este \t es \tun \t mensaje"))
```

```
"este es un ejemplo de sigil" apuntes de Elixir
Este          es     un       mensaje
"Este \t es \tun \t mensaje"
```

Concatenar cadenas:

```
defmodule Cadena do
  def concatenar(cad1, cad2, separador \\ " ") do
    cad1 <> separador <> cad2
  end
end
```

```
iex(1)> l(Elixir.Cadena)
{:module, Cadena}
iex(2)> Cadena.concatenar("Hola","Mundo")
"Hola Mundo"
iex(3)> Cadena.concatenar("Hola","Mundo","<->")
"Hola<->Mundo"
```

Pattern Matching:

```
iex> ^x = 5
5
iex> ^x = 10
** (MatchError) no match of right hand side value: 10

iex> 10 = x
** (MatchError) no match of right hand side value: 5
```

Operador pin: ^.
Evita la refijación (rebind).

```
#Pattern Matching-Funciones
defmodule Calculadoraa do
  def div(_,0) do
    {:error, "No se puede dividir por cero"}
  end
  def div(n1,n2) do
    {:ok, "El resultado es: #{n1/n2}"}
  end
end
```

```
iex(1)> l(Elixir.Calculadoraa)
{:module, Calculadoraa}
iex(2)> Calculadoraa.div(5,2)
{:ok, "El resultado es: 2.5"}
iex(3)> Calculadoraa.div(5,0)
{:error, "No se puede dividir por cero"}
```

Si invertimos el orden de las funciones, es decir:

```elixir
#Pattern Matching-Funciones
defmodule Calculadoraa do
  def div(n1,n2) do
    {:ok, "El resultado es: #{n1/n2}"}
  end
  def div(_,0) do
    {:error, "No se puede dividir por cero"}
  end
end
```

```
warning: this clause for div/2 cannot match because a previous clause at line 10 always matches
  apuntes.ex:13
```

Guardas:

```elixir
#Guardas
defmodule Numero do
  def cero?(0), do: true
  def cero?(n) when is_integer(n), do: false
  def cero?(_), do: "No es entero"
end
```

```
iex(2)> Numero.cero?(0)
true
iex(3)> Numero.cero?(5)
false
```

```
iex(4)> Numero.cero?(5.2)
"No es entero"
```

Condicionales

If:

```elixir
#Condicionales
#if
defmodule Personal do
  def sexo(sex) do
    if sex == :m do
      "Masculino"
    else
      "Femenino"
    end
  end
end
```

```
iex(2)> Personal.sexo(:m)
"Masculino"
iex(3)> Personal.sexo(:g)
"Femenino"
```

```elixir
defmodule Persona2 do
  def sexo(sex) do
    if sex == :m do
      "Masculino"
    else if sex == :f do
      "Femenino"
    else
      "Sexo desconocido"
    end
    end
  end
end
```

```
iex(2)> Persona2.sexo(:m)
"Masculino"
iex(3)> Persona2.sexo(:f)
"Femenino"
iex(4)> Persona2.sexo(:y)
"Sexo desconocido"
```

Case:

```elixir
#Case
defmodule Persona3 do
  def sexo(sex) do
    case sex do
      :m -> "Masculino"
      :f -> "Femenino"
      _ -> "Sexo Desconocido"
    end
  end
end
```

```
iex(5)> Persona3.sexo(:m)
"Masculino"
iex(6)> Persona3.sexo(:f)
"Femenino"
iex(7)> Persona3.sexo(:t)
"Sexo Desconocido"
```

Match con funciones:

Ejemplo 1

```elixir
#Match con funciones
#Ejemplo 1
defmodule Persona4 do
  def sexo(sex) when sex == :m do
    "Masculino"
  end
  def sexo(sex) when sex ==:f do
    "Femenino"
  end
  def sexo(_sex) do
    "Sexo desconocido"
  end
end
```

```
iex(1)> l(Elixir.Persona4)
{:module, Persona4}
iex(2)> Persona4.sexo(:m)
"Masculino"
iex(3)> Persona4.sexo(:f)
"Femenino"
iex(4)> Persona4.sexo(:o)
"Sexo desconocido"
```

Ejemplo2

```elixir
#Ejemplo 2
defmodule Persona5 do
  def sexo(sex) when sex == :m, do: "Masculino"
  def sexo(sex) when sex == :f, do: "Femenino"
  def sexo(_sex), do: "Sexo Desconocido"
end
```

```
iex(3)> Persona5.sexo(:m)
"Masculino"
iex(4)> Persona5.sexo(:f)
"Femenino"
iex(5)> Persona5.sexo(:q)
"Sexo Desconocido"
```

Cond:

```elixir
#Cond
defmodule Persona6 do
  def sexo(sex) do
    cond do
      sex == :m -> "Masculino"
      sex == :f -> "Femenino"
      true -> "Sexo desconocido"
    end
  end
end
```

```
iex(1)> l(Elixir.Persona6)
{:module, Persona6}
iex(2)> Persona6.sexo(:m)
"Masculino"
iex(3)> Persona6.sexo(:f)
"Femenino"
iex(4)> Persona6.sexo(:d)
"Sexo desconocido"
```