

Notas de Elixir

Shell de Elixir

- Cuando nos equivocamos en una expresión y no permite continuar el shell
- para salir del Shell se puede mediante CTRL+C o escribiendo System.halt
- Para pedir ayuda del Shell teclea "h"

Tipos de datos

- Elixir utiliza el mismo sistema de tipos de Erlang

Numeros

- Los números (numbers) pueden ser enteros o flotantes

Atoms

- 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

Nil

- similar al null de otros lenguajes
- Los átomos 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 de cortocircuito:
 - || -> retorna la primera expresión verdadera
 - && -> retorna la segunda siempre y cuando la primera lo sea también
 - ! -> retorna la negación de la expresión sin importar el tipo de dato
- || -> retorna la primera expresión verdadera

Tuplas

- son como estructuras o registros
- permiten agrupar elementos fijos

Listas

- Manejo dinámico de datos
- Funcionan como listas enlazadas simples

Variables

- Elixir es un lenguaje de programación dinámico
- NO es necesario declarar de manera explícita una variable
- El tipo de dato se determina de acuerdo al valor contenido
- La asignación(=) se conoce como fijación (binding)
- Cuando se inicializa una variable con un valor, la variable se fija con ese valor.

°características de las variables

-El nombre de una variable siempre inicia con un caracter alfabético en minúscula o caracter de subrayado (_)

- La convención es usar solo letras, dígitos y subrayados
- Pueden terminar con los caracteres ? o !
- Por convención el ? se utiliza cuando la función retorna true o false
- El ! se utiliza generalmente en funciones que podrían provocar algún error en tiempo de ejecución

*ejemplos:

```
*variable_valida
*esta_variable_tambien_es_valida
*esta_tambien_1
*estaEsValidaPeroNoRecomendada
*No_es_valida
*nombre_valido?
*claro_que_si!
```

- Los datos en Elixir son inmutables: su contenido no puede cambiarse.
- Las variables pueden ser refijadas (rebound) a un diferente valor

Modulos y Funciones

- Un módulo consta de varias funciones

- Cada función debe estar definida dentro de un módulo
- El módulo IO permite varias operaciones de E/S (I/O),
- la función puts permite imprimir un mensaje en pantalla
- Se utiliza el constructor defmodule para la creación de los módulos
- Dentro del módulo con el constructor def se crean las funciones.
- Una función siempre debe estar dentro de un módulo
- Los nombres de funciones son igual que las variables
- Tanto defmodule como def NO son palabras reservadas del lenguaje, son macros
- Un módulo puede estar dentro de un archivo. Un archivo puede contener varios módulos.
- Reglas de los módulos

– Inicia con una letra mayúscula

– Puede consistir en caracteres alfanuméricos, subrayados y puntos (.).

Regularmente se usa para la organización jerárquica de los módulos.

- Se pueden utilizar funciones privadas con el constructor defp

Aridad (Arity) de funciones

- Es el nombre para el número de argumentos que una función recibe
- Una función se identifica por:

1. el módulo donde se encuentra

2. su nombre

3. su aridad (arity)

Polimorfismo (sobrecarga)

- Dos funciones con el mismo nombre pero con diferente aridad son dos diferentes funciones.

defmodule Rectangulo do

def area(l) do

l * l

end

def area(l1,l2) do

l1 * l2

end

end

Argumentos por defecto

- Se pueden especificar argumentos por defecto mediante el operador \

```
defmodule Calculadora do
```

```
  def suma(n1,n2 \ 0) do
```

```
    n1 + n2
```

```
  end
```

```
end
```

Alias

- Se puede utilizar alias como alternativa a import, permite hacer una referencia a un módulo con otro nombre

- Elixir permite el registro de atributos, que se almacenarán en el archivo binario.

– @moduledoc

– @doc

- Sirven para documentar módulos y funciones

Shell

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.

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

-Techo de un número.

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

```

-|| -> retorna la primera expresión verdadera.

-&& -> 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 elemento 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
```

<pre>I0.puts("1. Este es un mensaje") I0.puts("2. Este es un \n mensaje") I0.puts("3. Este es un \"mensaje\"") I0.puts("4. Este es un \\mensaje\\") I0.puts("5. Este \t es \tun \t mensaje") I0.puts("4. Este es un mensaje")</pre>	<pre>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</pre>
---	---

Sigils:

```
I0.puts(~s("este es un ejemplo de sigil" apuntes de Elixir))
I0.puts("Este \t es \tun \t mensaje")
I0.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:

<pre>defmodule Cadena do def concatenar(cad1, cad2, separador \\ " ") do cad1 <> separador <> cad2 end end</pre>	<pre>iex(1)> l(Elixir.Cadena) {:module, Cadena} iex(2)> Cadena.concatenar("Hola","Mundo") "Hola Mundo" iex(3)> Cadena.concatenar("Hola","Mundo","<->") "Hola<->Mundo"</pre>
--	--

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).

<pre>#Pattern Matching-Funciones defmodule Calculadora 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</pre>	<pre>iex(1)> l(Elixir.Calculadora) {:module, Calculadora} iex(2)> Calculadora.div(5,2) {:ok, "El resultado es: 2.5"} iex(3)> Calculadora.div(5,0) {:error, "No se puede dividir por cero"}</pre>
---	---

Si invertimos el orden de las funciones, es decir:

```
#Pattern-Matching-Funciones
defmodule Calculadora do
  def div(n1,n2) do
    case {n1/n2} do
    end
  end
  def div(_,0) do
    case {0} do
    end
  end
end
```

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

Guardas:

```
#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:

```
#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"
```

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

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

Case:

```
#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

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

Herramienta mix

Mix

Es una herramienta de la línea de comandos (CLI)

Permite:

- Crear proyectos de Elixir
- Mantenerlos

Proporciona tareas para

- Documentar
- Compilar
- Depurar
- Probar (test)
- Manejar dependencias

Crear proyectos

```
C:\>mix new calculadora
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/calculadora.ex
* creating test
* creating test/test_helper.exs
* creating test/calculadora_test.exs
```

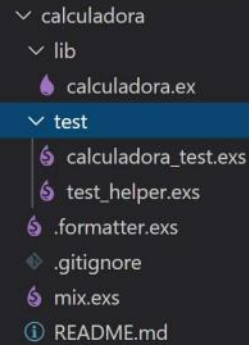
Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

```
cd calculadora
mix test
```

Run "mix help" for more commands.

C:\>

Estructura de directorios de un proyecto



Carga de una aplicación

- Ingresar al directorio donde se creó la nueva aplicación

```
C:\>cd calculadora
C:\calculadora>
```

- Lanzar el shell de Elixir con mix

```
C:\calculadora>iex -S mix
Compiling 1 file (.ex)
Generated calculadora app
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

- Ejecutar la función hello()

```
iex(1)> Calculadora.hello()
:world
iex(2)>
```

Documentación con ExDoc

- Abrir el archivo *mix.exs*
- Modificar las dependencias agregando `{:ex_doc, "~>0.12"}`

```
defp deps do
  [
    {:ex_doc, "~>0.12"}
  ]
end
```

- Ejecutar el comando *mix deps.get*

Doctest

- Se realiza a partir de la documentación de las funciones



```
@doc """
Hello world.

## Examples

iex> Calculadora.hello()
:world

"""
def hello do
  :world
end
```

Test

- Se realiza a partir del script del test

Funciones anónimas

- No tienen nombre
- Se pueden fijar a variables

Ejemplos de funciones anónimas

- Ejemplo 1

Código fuente

```
defmodule Calculadora do
  def suma(n1,n2), do: n1+n2
end
suma_anonima = fn(n1,n2) -> n1 + n2 end

IO.puts(Calculadora.suma(5,4))
IO.puts(suma_anonima.(5,5))

>elixir main.ex
9
10
```

Operador Pipe

- Dada una lista con n numeros, se desea obtener el cuadrado de la suma de los elementos de la cola. Si la lista es [1,2,3,4,5], el resultado es (2+3+4+5)^2
- `csc = cuadrado(suma(2,3,4,5))`

```
#Loops y recursion
#Representan una secuencia de números
#Se definen con un límite inferior y un límite superior
# Son inclusivos
# Se separan con dos puntos (..)
# Son equivalentes a una lista
#Es más eficiente que una lista de números secuenciales, puesto que solo se
  #almacenan dos enteros, el del inicio y el del final
# Son enumerables, cada número se genera conforme se itere sobre el rango
# La función Enum puede usarse con rangos
```

Shell de elixir

Entrar con interactive elixir (iex)

```
C:\Users\HP>iex
```

```
Interactive Elixir (1.14.1) - press Ctrl+C to exit (type h() ENTER for help)
```

```
iex(1)> 5+4
```

```
9 //Retorno
```

```
iex(2)> 5
```

```
5 //Retorno
```

```
iex(3)> 5+
```

```
...(3)> 4
```

```
9 //Retorno
```

//Las funciones de elixir siempre deben retornar algo

```
iex(4)> 5+4;6-3;8*4
```

32 //Cuando hagas varias operaciones, siempre se va a retornar el resultado de la última operación. No significa que no se hayan calculado significa que el valor de sus resultados no fue guardado o retornado

- Aridad de una función (b/1). Son los argumentos que recibe la función
- Elixir es un lenguaje de programación de tipado dinámico
- No es necesario declarar de manera explícita una variable o su tipo de dato
- El tipo de dato se determina de acuerdo al valor del contenido
- La asignación se conoce como fijación (binding)
- Cuando se inicializa una variable con un valor, la variable se fija con ese valor
- Cuando pones un string este se fija como lista, porque los strings no existen como tal en erlang y elixir

Estilos

- El nombre de una variable siempre se inicia con una letra minúscula

- El guion bajo se usa para separar palabras (_)
- El signo de interrogación “?” se recomienda usar para variables booleanas
- El signo de admiración “!” se recomienda usar para warnings
- No es recomendable usar el formato camelCase

Inmutabilidad

- Los datos de elixir son inmutables. Su contenido no puede cambiarse
- Las variables pueden ser refijadas (rebound) a un diferente valor

```
iex(1)> dia_semana =5 //Se fija el valor inicial
```

```
5
```

```
iex(2)> dia_semana //Se verifica
```

```
5
```

```
iex(3)> dia_semana = 7 //Se refija el valor inicial
```

```
7
```

```
iex(4)> dia_semana //Se verifica el efecto de la refijacion
```

```
7
```

//Cuando se refija una variable inmutable lo que hace es que en la memoria crea un nuevo valor para esa variable, el valor anterior se elimina. Cuando es mutable el valor que tiene una variable se modifica pero no se borra nada de la memoria

Módulos

- Un archivo puede tener varios modulos
- Un módulo consta de varias funciones
- Cada funcion debe estar definida dentro de un modulo
- El modulo IO permite varias operaciones de E/S(I/O)-Entrada/Salido-Input/output. La función puts permite imprimir un mensaje en la pantalla. Retorna un “:ok”

Syntaxis general

NombreModulo.nombre_funcion(argumentos)

- El nombre del modulo siempre se escribe en formato CamelCase iniciando con una letra mayúscula

- Se utiliza el constructor `defmodule` para la creación de los modulos
- Dentro del modulo con el constructor `def` se crean las funciones

Funciones

- Una función siempre debe de estar dentro de un modulo
- Los nombres de las funciones se declaran igual que las variables

Elixir

`[h|t]=lista`

`H=head`

`T=tail`

`Lista= [1,2,3]`

`Head =1`

`Tail = [2,3]`

El simbolo “=” en elixir es Binding=fijación, que es el símbolo de pattern matching

Rebiding= vuelves a usar lo que está a la derecha del “=”.

`Lista=[1,2,3]`

`Biding= [uno|lista]=lista`

`Uno=1`

`Lista=[2,3]`

`Rebiding= [dos|lista]=lista`

`Dos= 2`

`Lista=[3]`

`Rebiding= [tres|lista]=lista`

`Tres=3`

`Lista=[]`

Suma de dos listas:

La sobre carga del símbolo de + “++”, te permite unir dos listas.

`Lista1=[1,2,3]`

`Lista2=[4,5,6]`

`Listaconvinada=lista1++lista2`

`Listaconvinada=[1,2,3,4,5,6]`

Resta de dos listas:

Con “-” restas dos listas

```
Lista1=[1,2,3]
```

```
Lista2=[2,3,4]
```

```
Listaconvinada=lista1- lista2
```

```
Listaconvinada=[1]
```

Y si tuviéramos:

```
Lista1[1,3,3]
```

```
Lista2=[2,3,4]
```

```
Listaconvinada=lista1- lista2
```

```
Listaconvinada=[1,3]
```

Poner un valor al inicio de la lista:

```
N=1
```

```
Lista=[2,3]
```

```
Lista=[n | lista]
```

```
Lista=[1,2,3]
```

Datos de elixir:

Todos los datos de elixir, todas las variables son inmutables; NO se pueden cambiar.

El operador pin=”^” protege de que la variable no se pueda modificar = lista^=[n | lista].

La función “i” es para ninspeccionar una lista = i(lista)

```
iex(59)> i(lista)
Term
  [2, 3]
Data type
  List
Reference modules
  List
Implemented protocols
  Collectable, Enumerable, IEx.Info, Inspect, List.Chars, String.Chars
iex(60)>
```

```
Enum.suma(["todo lo que este aquí separado por ‘,’ van a ser sumados"])
```

Para el enum hay que cargar la función “i”, creo.

Es tipo de programación **declarativa**.

Variables:

Elixir es un lenguaje de programación dinámico.

- No es necesario declarar de manera explícita una variable o su tipo de dato.
- El tipo de dato de determina de acuerdo al valor contenido.
- La asignación se conoce como fijación (binding).
- Cuando se inicializa una variable con un valor, la variable se fija con ese valor.

```
iex()> dia_semana = 7 <fija (binds) el valor>
7 <resultado de la última expresión>
iex()> dia_semana <expresion que retorna el valor de la variable>
7 <valor de la variable>
iex()> dia_semana * 2
14
```

Características de las variables.

- El nombre de una variable siempre inicia con un carácter alfabético en minúscula o carácter de subrayado (_).
- Después puede llevar cualquier combinación de estos caracteres.
- La convención es usar solo letras, dígitos y subrayados.
- Pueden terminar con los caracteres “?” o “!”.

```
variable_valida
esta_variable_tambien_es_valida
esta_tambien_1
estaEsValidaPeroNoRecomendada
No_es_valida
nombre_valido?
claro_que_si!
```

Inmutabilidad.

- Los datos en Elixir son inmutables: su contenido no puede cambiarse.
- Las variables pueden ser refijadas (rebound) a un diferente valor.

```
iex()> dia_semana = 5 <se establece el valor inicial>  
5
```

```
iex()> dia_semana <verificación>  
5 <>  
iex()> dia_semana = 7 <se refija el valor inicial>  
7 <>  
iex()> dia_semana <se verifica el efecto de la refijación>  
7 <>
```

Estructura del código

Módulos y Funciones

Módulos.

- Un módulo consta de varias funciones.
- Cada función debe estar definida dentro de un módulo.
- El módulo IO permite varias operaciones de E/S (I/O), la función `puts` permite imprimir un mensaje en pantalla.

```
iex()> IO.puts("Hola Mundo")  
Hola Mundo  
:ok
```

- La sintaxis general es: *NombreModulo.nombre_funcion(args)*.
- Se utiliza el constructor “*defmodule*” para la creación de los módulos.
- Dentro del módulo con el constructor “*def*” se crean las funciones.

Funciones.

Una función siempre debe estar dentro de un módulo.

Los nombres de funciones son igual que las variables:

- El nombre de una variable siempre se inicia con un carácter alfabético en minúscula o carácter de subrayado (`_`).
- Después puede llevar cualquier combinación de estos caracteres.
- La convención es usar solo letras, dígitos y subrayados.
- Pueden terminar con los caracteres “?” o “!”.
- Por convención el “?” se utiliza cuando la función retorna “*true*” o “*false*”.

-El “!” se utiliza generalmente en funciones que podrían provocar algún error en tiempo de ejecución.

-Tanto “*defmodule*” como “*def*” NO son palabras reservadas del lenguaje, son *macros*.

Función sin argumentos (procedimiento):

```
#Función sin argumentos
defmodule HolaMundo do
  def mensaje do
    IO.puts("Hola mundo")
  end
end

iex(1)> l(Elixir.HolaMundo)
{:module, HolaMundo}
iex(2)> HolaMundo.mensaje
Hola mundo
:ok
```

Función con argumentos:

```
#Función con argumentos
#Área de un cuadrado:
defmodule Areas do
  def area_cuadrado(1) do
    1*1
  end
end

iex(3)> l(Elixir.Areas)
{:module, Areas}
iex(4)> Areas.area_cuadrado(8)
64
```

-Un módulo puede estar dentro de un archivo. Un archivo puede contener varios módulos.

Reglas de los módulos:

- Inicia con una letra mayúscula
- Se escribe con el estilo CamelCase
- Puede consistir en caracteres alfanuméricos, subrayados y puntos (.).
- Regularmente se usa para la organización jerárquica de los módulos.

```
defmodule Geometria.Cuadrado do
  def perimetro(1) do
    4*1
  end
end

defmodule Geometria.Rectangulo do
  def perimetro(l1,l2) do
    2*l1 + 2*l2
  end
end
```

También se pueden anidar de la siguiente forma:

```
#Lo anterior también se pueden anidar de la siguiente forma:
defmodule Geometria do
  defmodule Cuadrado do
    def perimetro(1) do
      4*1
    end
  end
  defmodule Rectangulo do
    def perimetro(l1,l2) do
      2*l1 + 2*l2
    end
  end
end
```

Las funciones pueden expresarse de manera condensada:

```
#Las funciones pueden expresarse de manera condensada:
defmodule Geometria do
  def perimetro_cuadrado(1), do: 4*1
  def perimetro_rectangulo(11,12), do: 2*11 + 2*12
end
```

Los paréntesis en los argumentos son opcionales:

```
iex()> Geometria.perimetro_cuadrado 4
16
iex()> Geometria.perimetro_rectangulo 4,3
14
```

Visibilidad de funciones

-Se pueden utilizar funciones privadas con el constructor defp.

-Función Pública y privada.

```
defmodule TestPublicoPrivado do
  def funcion_publica(msg) do #Pública
    IO.puts("#{msg} publico")
  end
  defp funcion_privada(msg) do #Privada
    IO.puts("#{msg} privado")
  end
end

iex(2)> TestPublicoPrivado.funcion_publica("Hola")
Hola publico
Hola privado
:ok
```

Módulo Geometría:

```
#Módulo Geometría:
defmodule Geometria do
  def perimetro1(1), do: cuadrado(1)
  def perimetro2(1), do: Geometria.cuadrado(1)
  defp cuadrado(1), do: 4*1
end
```

Para ejecutar sus privadas hay que usar un **Operador Pipeline**.

```
iex(3)> 4 |> Geometria.perimetro1
16
```

```
#Obtener el cuadrado de la suma de 2 números
#Invocando funciones
```

```
defmodule Operaciones do
  def suma(n1,n2), do: n1 + n2
  def cuadrado(n), do: n * n
end
```

```
Operaciones.cuadrado(Operaciones.suma(4,5))
```

```
{:module, Operaciones}
iex(2)> Operaciones.cuadrado(Operaciones.suma(4,5))
81
```


Capítulo 7. Estructura del código en Elixir

Estructura del código

Aridad (Arity) de funciones

-Es el nombre para el número de argumentos que una función recibe

-Una función se identifica por:

1. el módulo donde se encuentra,
2. su nombre y
3. su aridad (arity)

Polimorfismo (sobrecarga)

-Dos funciones con el mismo nombre, pero con diferente aridad son dos diferentes funciones.

```
#Haciendo que una función dependa de otra de diferente aridad, se podría realizar lo siguiente:
defmodule Calculadora do
  def suma(n) do
    suma(n, 0)
  end
  def suma(n1, n2) do
    n1 + n2
  end
end

#Argumentos por defecto
# -Se pueden especificar argumentos por defecto mediante el operador
# Este módulo genera dos funciones como en el caso anterior
defmodule Calculadora do
  def suma(n1, n2 \\ 0) do
    n1 + n2
  end
end

#Se puede utilizar cualquier combinación de argumentos por defecto:
defmodule Calculadora do
  def funcion(n1, n2 \\ 0, n3 \\ 1, n4, n5 \\ 2) do
    n1 + n2 + n3 + n4 + n5
  end
end
```