1. Installing Elixir:
   https://elixir-lang.org/install.html
   o usar el IDE online:
   https://www.tutorialspoint.com/execute_elixir_online.php
2. Para empezar Elixir interactive shell
   Escribir iex

Hello World

3. Para poner ("put") el string en la salida de la consola:
   `IO.puts "Hello world"`
   O con paréntesis:
   `IO.puts("Hello world")`

Bases

4. Comentarios:

   Comentarios de una línea -> #
   `#This is a comment in Elixir`

   Comentarios de varias líneas -> no existen

5. Finales de línea:

   No es obligatorio poner ningun caracter como fin de línea pero se pueden poner varias sentencias en la misma línea usando `;`.
   `IO.puts("Hello"); IO.puts("World!")`

6. Identificadores:

   Se empieza con letras minúsculas y luego se pueden poner numeros, _ o letras mayúsculas.
   `var1 variable_2 one_M0r3_variable`

   Un valor que no debe ser usado se asigna a una variable que empiece con _. Es decir esa variable no se usará nuevamente, pero es necesario asignarla a algo. También se puede hacer con el nombre de una función privada.
   `_some_random_value = 42`

7. Palabras reservadas:

```
after and catch do inbits inlist nil else end
not or false fn in rescue true when xor
__MODULE__ __FILE__ __DIR__ __ENV__ __CALLER__
```

8. Tipos de dados:
    a. integers.
    b. floats: Elixir admite precisión de 64 bits para números flotantes. También se pueden definir utilizando un estilo de exponenciación. Ejemplo, 10145230000 se puede escribir como 1.014523e10.

    c. octal: para definir un número en la base octal se usa el prefijo '0o'. Ejemplo: 0o52 en octal equivale a 42 en decimal.

    d. hexadecimal: para definir un número en base decimal se usa el prefijo '0x'.
    Ejemplo: 0xF1 en hex es equivalente a 241 en decimal.

    e. binario: para definir un número en base binaria se usa el prefijo '0b'.
    Ejemplo: 0b1101 en binario es equivalente a 13 en decimal.

    f. atoms: son constantes cuyo nombre es su valor. Se crean usando el símbolo (:).
    Ejemplo: `:hello`

    g. booleanos: elixir admite **true** y **false** como booleanos. Ambos valores están de hecho unidos a átomos `:true` y `:false` respectivamente.

    h. Strings: Se insertan entre comillas simples.
    `"Hello world"`
    Y para definir múltiples strings:
    ```
    """
    Hello
    World!
    """
    ```

    i. Binarios: los binarios se usan principalmente para manejar datos relacionados con bits y bytes.
    `<< 65, 68, 75>>`

    j. Listas: Elixir usa corchetes para especificar una lista de valores. Los valores pueden ser de cualquier tipo.
    `[1, "Hello", :an_atom, true]`

    k. Tuplas: Elixir usa corchetes para definir tuplas. Al igual que las listas, las tuplas pueden contener cualquier valor.
    `{ 1, "Hello", :an_atom, true`

	Organizado como	Insertar	Eliminar	Acceder
Listas	linked list	very fast	very fast	
Tuplas	bloques de memoria continua			very fast

9. Declaración de variables:
   Una variable debe declararse y asignarse a un valor al mismo tiempo.
   ```
   life = 42
   ```

   Si queremos reasignar a esta variable un nuevo valor, podemos hacer esto usando la misma sintaxis.
   ```
   life = "Hello world"
   ```

10. Operadores:
    a. Operadores aritméticos:

Supongamos que la variable A es 10 y la variable B es 20.

Operador	Descripción	Ejemplo
+		A + B da 30
-		A + B da -10
*		A * B da 200
/	Divide el primer número desde el segundo. Esto arroja los números en flotadores y da un resultado de flotación	A / B da 0.5
div	Esta función se usa para obtener el cociente en la división.	div(10, 20) da 0
rem	Esta función se usa para obtener el resto en la división.	rem(A, B) da 10

    b. Comparación de operadores:

Supongamos que la variable A es 10 y la variable B es 20.

Operador	Descripción	Ejemplo
==		A == B da false
!=		A != B da true

===	Comprueba si el tipo de valor a la izquierda es igual al tipo de valor a la derecha, en caso afirmativo, verifique lo mismo para el valor.	A === B da false
!==	Lo mismo que arriba, pero verifica la desigualdad en lugar de la igualdad.	A !== B da true
>		A > B da false
<		A < B da true
>=		A >= B da false
<=		A <= B da true

c. Operadores lógicos:

Suponer que la variable A es verdadera y la variable B es 20.

Operador	**Descripción**	**Ejemplo**
and	Comprueba si los dos valores proporcionados son verdaderos, si es así, devuelve el valor de la segunda variable.	A and B da 20
or	Comprueba si el valor proporcionado es verdadero. Devuelve el valor que sea verdadero. Else devuelve falso.	A or B da true
not	Operador unario que invierte el valor de la entrada dada.	not A dará falso
&&	No es un and estricto. Funciona igual pero no espera que el primer argumento sea un booleano.	B && A da 20
||	No es un or estricto. Funciona igual pero no espera que el primer argumento sea booleano.	B || A da true
!	No es un no estricto. Funciona igual que pero no espera que el argumento sea booleano.	!A da false

d. Operadores bit a bit:

Elixir proporciona módulos bit a bit como parte del paquete Bitwise. Para usarlo, ingrese el siguiente comando en su shell:

```
use Bitwise
Supongamos que A es 5 y B es 6 para los siguientes ejemplos
```

Operador	Descripción	Ejemplo
&&&	Copia un bit para verificar si existe en ambos operandos.	A &&& B da 4
|||	Copia un bit para verificar si existe en cualquiera de los operandos.	A ||| B da 7
>>>	Desplaza los primeros bits del primer operando a la derecha por el número especificado en el segundo operando.	A >>> B da 0
<<<	Desplaza los primeros bits del primer operando a la izquierda por el número especificado en el segundo operando.	A <<< B da 320
^^^	Copia un poco para verificar si es diferente en ambos operandos.	A ^^^ B da 3
~~	Invierte los bits en el número dado.	~~~A da -6

La coincidencia de patrones

Un match tiene 2 partes principales, un lado izquierdo y un lado derecho. El lado derecho es una estructura de datos de cualquier tipo. El lado izquierdo intenta hacer coincidir la estructura de datos en el lado derecho y unir las variables de la izquierda a la subestructura respectiva de la derecha.

```
[var_1, _unused_var, var_2] = [{"First variable"}, 25, "Second variable" ]
```

```
[_, [_, {a}]] = ["Random string", [:an_atom, {24}]]
```

```
a = 25
b = 25
^a = b
```

Si tenemos un conjunto que no coincidente de lado izquierdo y derecho, el operador de coincidencia genera un error. Por ejemplo, si tratamos de hacer coincidir una tupla con una lista o una lista de tamaño 2 con una lista de tamaño 3, se mostrará un error.

Toma de decisiones

11. if else statement:

    Una instrucción if puede ser seguida por una instrucción else opcional.

```
a = false
if a === true do
   IO.puts "Variable a is true!"
else
   IO.puts "Variable a is false!"
end
IO.puts "Outside the if statement"
```

12. unless else statement:

    Una sentencia unless tiene el mismo cuerpo que una instrucción if. El código dentro de la declaración unless se ejecuta solo cuando la condición especificada es falsa.

```
a = false
unless a === false do
   IO.puts "Condition is not satisfied"
else
   IO.puts "Condition was satisfied!"
end
IO.puts "Outside the unless statement"
```

13. cond statement:
    Una declaración de cond se usa cuando queremos ejecutar código en base a varias condiciones. Funciona como un if ... else if ... .En muchos otros lenguajes de programación.

```
guess = 46
cond do
   guess == 10 -> IO.puts "You guessed 10!"
   guess == 46 -> IO.puts "You guessed 46!"
   guess == 42 -> IO.puts "You guessed 42!"
   true        -> IO.puts "I give up."
end
```

14. case statement:

    Se puede considerar como un reemplazo de la declaración switch. Case toma una variable / literal y aplica la coincidencia de patrones con diferentes casos. Sí cualquier caso coincide, Elixir ejecuta el código asociado con ese caso y sale del case statement. Si no se encuentra ninguna coincidencia, sale de la instrucción con un CaseClauseError que muestra que no se encontraron cláusulas coincidentes. Siempre debe tener un caso con _ que coincida con todos los valores. Esto ayuda a prevenir el error mencionado anteriormente. Esto es comparable al caso default.

```
case 3 do
   1 -> IO.puts("Hi, I'm one")
   2 -> IO.puts("Hi, I'm two")
   3 -> IO.puts("Hi, I'm three")
```

```
    _ -> IO.puts("Oops, you dont match!")
end
```

String

15. Interpolación de cadenas

```
x = "Apocalypse"
y = "X-men #{x}"
IO.puts(y)
```

16. Concatenación de cadenas:
    Se hace con el operador <>.

```
x = "Dark"
y = "Knight"
z = x <> " " <> y
IO.puts(z)
```

17. Concatenación de cadenas:

```
x = "Dark"
y = "Knight"
z = x <> " " <> y
IO.puts(z)
```

18. Longitud de la cadena:

```
IO.puts(String.length("Hello"))
```

19. Invertir una cadena:

```
IO.puts(String.reverse("Elixir"))
```

20. Comparación de cadenas:

```
var_1 = "Hello world"
var_2 = "Hello Elixir"
if var_1 === var_2 do
   IO.puts("#{var_1} and #{var_2} are the same")
else
   IO.puts("#{var_1} and #{var_2} are not the same")
end
```

21. Coincidencia de cadenas:

```
IO.puts(String.match?("foo", ~r/foo/))
IO.puts(String.match?("bar", ~r/foo/))
```

    o usando el operador =~

```
IO.puts("foo" =~ ~r/foo/)
```

Más funciones de strings: https://www.tutorialspoint.com/elixir/elixir_strings.htm

Lista de caracteres

Mientras que las comillas dobles representan una cadena (es decir, un binario), las comillas simples representan una lista de caracteres (es decir, una lista).

```
IO.puts('Hello')
IO.puts(is_list('Hello'))
```

Se puede convertir una lista de caracteres en una cadena y volver.

```
IO.puts(is_list(to_char_list("hełło")))
IO.puts(is_binary(to_string ('hełło')))
```

List

Cuando Elixir ve una lista de números ASCII imprimibles los imprime como una lista de caracteres.

```
IO.puts([104, 101, 108, 108, 111])
```

22. Concatenación y Resta:
    Dos listas se pueden concatenar y restar usando los operadores ++ y --.

    ```
    IO.puts([1, 2, 3] ++ [4, 5, 6])
    IO.puts([1, true, 2, false, 3, true] -- [true, false])
    ```

23. Cabeza y cola de una lista:

    ```
    list = [1, 2, 3]
    IO.puts(hd(list))
    IO.puts(tl(list))
    ```

Más funciones de listas: https://www.tutorialspoint.com/elixir/elixir_lists_and_tuples.htm

Tuplas

24. Longitud de una tupla:
    ```
    IO.puts(tuple_size({:ok, "hello"}))
    ```

25. Agregar un valor:
    ```
    tuple = {:ok, "Hello"}
    Tuple.append(tuple, :world)
    ```

26. Insertar un valor

```
tuple = {:bar, :baz}
new_tuple_1 = Tuple.insert_at(tuple, 0, :foo)
new_tuple_2 = put_elem(tuple, 1, :foobar)
```

Lista de palabras clave

En Elixir, tenemos dos estructuras de datos asociativas (estructuras de datos que pueden asociar un valor o múltiples valores a una clave): keywords y maps.

```
list_1 = [{:a, 1}, {:b, 2}]
list_2 = [a: 1, b: 2]
IO.puts(list_1 == list_2)
```

Maps

Cuando se necesite almacenar un par key-value se usan mapas.

27. Crear un mapa:
    Un mapa es creado usando la sintaxis %{}

    ```
    map = %{:a => 1, 2 => :b}
    ```

    Los mapas permiten cualquier valor como clave.

28. Acceder a una llave:

    ```
    map = %{:a => 1, 2 => :b}
    IO.puts(map[:a])
    IO.puts(map[2])
    ```

29. Insertar una llave:

    ```
    map = %{:a => 1, 2 => :b}
    map = Dict.put_new(map, :new_val, "value")
    IO.puts(map[:new_val])
    ```

30. Actualizar un valor:

    ```
    map = %{:a => 1, 2 => :b}
    map = %{ map | a: 25}
    IO.puts(map[:a])
    ```

31. La coincidencia de patrones:

    ```
    %{:a => a} = %{:a => 1, 2 => :b}
    IO.puts(a)
    ```

32. Mapas con llave tipo Atom
    Cuando todas las claves en un mapa son de tipo atom, puede usar la siguiente.

```
map = %{:a => 1, 2 => :b}
IO.puts(map.a)
```

Módulos

Agrupamos varias funciones en módulos.

```
defmodule Math do
   def sum(a, b) do
      a + b
   end
end
IO.puts(Math.sum(1, 2))
```

33. Módulo anidado:

```
defmodule Foo do
   #Foo module code here
   defmodule Bar do
#Bar module code here
   end
end
```

    El ejemplo anterior definirá dos módulos: Foo y Foo.Bar.

Alias

```
alias String, as: Str
IO.puts(Str.length("Hello"))
```

Los alias son válidos sólo dentro del alcance léxico en el que son llamados. Es decir, si tiene 2 módulos en un archivo y crea un alias dentro de uno de los módulos, ese alias no será accesible en el segundo módulo.

Require

```
require Integer
Integer.is_odd(3)
```

Require también tiene un alcance léxico

Import

Usamos la directiva import para acceder fácilmente a funciones o macros de otros módulos.

```
import Integer, only: :macros
```

: only es opcional, se recomienda su uso para evitar la importación de todas las funciones de un módulo determinado dentro del espacio de nombres. También se puede poner :except.

Funciones

34. Funciones anónimas: Estas funciones a veces también se llaman lambdas. Se utilizan asignándoles a variables. Usan las palabras claves fn y end.

```
sum = fn (a, b) -> a + b end
IO.puts(sum.(1, 5))
```

Usando el operador de captura:

```
sum = &(&1 + &2)
IO.puts(sum.(3, 2))
```

Funciones de coincidencia de patrones:
Podemos usar la coincidencia de patrones para hacer que nuestras funciones sean polimórficas. Por ejemplo, declararemos una función que puede tomar 1 o 2 entradas (dentro de una tupla) e imprimirlas en la consola:

```
handle_result = fn
   {var1} -> IO.puts("#{var1} found in a tuple!")
   {var_2, var_3} -> IO.puts("#{var_2} and #{var_3} found!")
end
handle_result.({"Hey people"})
handle_result.({"Hello", "World"})
```

35. Funciones nombradas:
Las funciones con nombre se definen dentro de un módulo usando la palabra clave def. Las funciones con nombre siempre se definen en un módulo.

```
defmodule Math do
   def sum(a, b) do
      a + b
   end
end

IO.puts(Math.sum(5, 6))
```

Funciones privadas
Se puede acceder desde el módulo en el que están definidas. se usa **defp** en lugar de **def.**

```
defmodule Greeter do
   def hello(name), do: phrase <> name
   defp phrase, do: "Hello "
end

Greeter.hello("world")
```

Si intentamos correr: Greeter.phrase() function, va a lanzar un error.

Argumentos predeterminados
Si queremos un valor predeterminado para un argumento, usamos la sintaxis de **argumento \\ valor**.

```
defmodule Greeter do
   def hello(name, country \\ "en") do
      phrase(country) <> name
   end

   defp phrase("en"), do: "Hello, "
   defp phrase("es"), do: "Hola, "
end

Greeter.hello("Ayush", "en")
Greeter.hello("Ayush")
Greeter.hello("Ayush", "es")
```

Recursiones

```
defmodule Math do
   def fact(res, num) do
   if num === 1 do
      res
   else
      new_res = res * num
      fact(new_res, num-1)
      end
   end
end

IO.puts(Math.fact(1,5))
```

Estructuras

```
#Definir una estructura

defmodule User do

   defstruct name: "John", age: 27

end


#Las estructuras toman el nombre del módulo en el que están definidas

john = %User{} #john is: %User{age: 27, name: "John"}

ayush = %User{name: "Ayush", age: 20}


#Para acceder a name y age de john,

IO.puts(john.name)

IO.puts(john.age)
```

```
#Para acceder a name y age de ayush,

IO.puts(john.name)

IO.puts(john.age)


#Y para actualizar el nombre de john

john = %{john | name: "Meg"}
```

Protocolos

Un protocolo se puede pensar esencialmente como lo mismo que una interfaz.

```
#Defining the protocol
defprotocol Blank do
   def blank?(data)
end

#Implementing the protocol for lists
defimpl Blank, for: List do
   def blank?([]), do: true
   def blank?(_), do: false
end

#Implementing the protocol for strings
defimpl Blank, for: BitString do
   def blank?(""), do: true
   def blank?(_), do: false
end

#Implementing the protocol for maps
defimpl Blank, for: Map do
   def blank?(map), do: map_size(map) == 0
end

IO.puts(Blank.blank? [])
IO.puts(Blank.blank? [:true, "Hello"])
IO.puts(Blank.blank? "")
IO.puts(Blank.blank? "Hi")
```

Procesos

- En Elixir, todo el código se ejecuta dentro de los procesos.
- Los procesos están aislados unos de otros, se ejecutan simultáneamente y se comunican a través del envío de mensajes.

- Los procesos de Elixir no deben confundirse con los procesos del sistema operativo.
- Los procesos en Elixir son extremadamente livianos en términos de memoria y CPU (a diferencia de los hilos en muchos otros lenguajes de programación). Debido a esto, no es raro tener decenas o incluso cientos de miles de procesos ejecutándose simultáneamente.

36. La función Spawn
    La forma más fácil de crear un nuevo proceso es usar la función spawn. Spawn acepta una función que se ejecutará en el nuevo proceso.

```
pid = spawn(fn -> 2 * 2 end)
Process.alive?(pid)
```

    ya que los códigos Elixir se ejecutan dentro de los procesos. Si ejecuta la función self, verá el PID para su sesión actual:

```
pid = self
Process.alive?(pid)
```

37. Paso de mensajes
    Podemos enviar mensajes a un proceso con **send** y recibirlos con **receive**.

    Ejemplo:
    Pasemos un mensaje al proceso actual y lo recibimos en el mismo.

```
send(self(), {:hello, "Hi people"})

receive do
   {:hello, msg} -> IO.puts(msg)
after
   1_000 -> "nothing after 1s"
end
```

    Cuando se envía un mensaje a un proceso, el mensaje se almacena en el buzón del proceso. El bloque de recepción pasa por el buzón de proceso actual buscando un mensaje que coincida con cualquiera de los patrones dados.

    Si no hay ningún mensaje en el buzón que coincida con ninguno de los patrones, el proceso actual esperará hasta que llegue un mensaje coincidente. También se puede especificar un tiempo de espera con `after`.

38. Enlaces

    Supongamos este programa

    ```
    spawn fn -> raise "oops" end
    ```

    cuando el programa de arriba corre produce es siguiente error.

    ```
    [error] Process #PID<0.58.00> raised an exception
    ** (RuntimeError) oops
       :erlang.apply/2
    ```

    Se ha registrado un error pero el proceso de generación aún se está ejecutando. Esto se debe a que los procesos están aislados. Si queremos que la falla en un proceso se propague a otro, debemos vincularlos.

    ```
    spawn_link fn -> raise "oops" end
    ```

    Cuando se ejecuta el programa anterior, produce el siguiente error:

    ```
    ** (EXIT from #PID<0.41.0>) an exception was raised:
       ** (RuntimeError) oops
          :erlang.apply/2
    ```

Los procesos y enlaces juegan un papel importante cuando se construyen sistemas tolerantes a fallas. En las aplicaciones de Elixir, a menudo vinculamos nuestros procesos a los supervisores, que detectarán cuándo un proceso falla e inicia un nuevo proceso en su lugar. Esto solo es posible porque los procesos están aislados y no comparten nada por defecto. Y dado que los procesos están aislados, no hay forma de que una falla en un proceso bloquee o corrompa el estado de otro. Mientras que otros idiomas requerirán que capturemos / manejemos excepciones; en Elixir, estamos de acuerdo en dejar que los procesos fallen porque esperamos que los supervisores reinicien nuestros sistemas correctamente.

Filtros

```
import Integer
IO.puts(for x <- 1..10, is_even(x), do: x)
```

Cuando un valor filtrado devuelve falso o nulo, se excluye de la lista final. En el ejemplo de arriba utilizaremos la función `is_even` del módulo Integer para verificar si un valor es par o no.

Podemos usar filtros múltiples en la misma comprensión agregando otro filtro después del filtro `is_even` separado por una coma.

Tipos personalizados

Es conveniente definir tipos personalizados cuando sea apropiado. Esto se puede hacer con la directiva @type.

Ejemplo:

```
defmodule FunnyCalculator do
   @type number_with_joke :: {number, String.t}

   def add(x, y), do: {x + y, "You need a calculator to do that?"}

   def multiply(x, y), do: {x * y, "It is like addition on steroids."}
end

{result, comment} = FunnyCalculator.add(10, 20)
IO.puts(result)
IO.puts(comment
```

Los tipos personalizados definidos a través de @type se exportan y están disponibles fuera del módulo en el que están definidos. Si desea mantener un tipo personalizado privado, puede usar la directiva @typep en lugar de @type.