Básicos

En esta sección se expresan algunos temas básicos en el uso de `Julia`.

Tipos de números

En `Julia` hay diferentes tipos de números. Por default se almacenan en 64 bits (si el ordenador es de 64 bits).

- `int`: números enteros.
- `float`: números con punto flotante.
- `complex`: números con parte imaginaria. Pueden ser int o float.

Veamos a los números enteros:

```
1 using PlutoUI
```

```
1 print(5)
```

```
5
```

Un número con punto flotante puede escribirse en notación científica con la notación habitual:

```
5.0e2 # 500.0
```

Los números escritos en notación científica siempre son `float`.

```
1 print(5.0e2)
```

```
500.0
```

Los números complejos se escriben como un número imaginario sumado a otro número.

```
2 + 3im # Imaginario
```

```
print(2+3im)
```

```
2 + 3im
```

💡 En Julia los números siempre tienen la prioridad dependiendo de qué tipo de números involucran. Por ejemplo sumar un `int` y un `float` devuelve `float`.

Operaciones ariméticas

Las operaciones aritméticas funcionan de manera habitual. Sea `a=10` y `b=5`:

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 % 5 = 0
```

💡 Una multiplicación por una constante puede escribirse de manera contigua, sin usar el operador `*`.

```
2a = 20
```

División

Hay tres tipos de divisiones. La **división exacta** se consigue con el operador `/`:

```
10 / 5 = 2.0
```

La **división entera** se consigue con el operador `\div = ÷`:

```
10 ÷ 5 = 2
```

Los **números racionales** también pueden expresarse para realizar operaciones aritméticas siempre terminando en la expresión mínima.

```
1//2 + 1//3 = 5//6
```

Potenciación

La operación con exponente se consigue con el operador `^`:

```
2 ^ 4 = 16
```

Algunas operaciones pueden dar lugar al tipo `Inf`:

```
1 / 0 = Inf
1 // 0 = 1//0
0^-1 = Inf
```

Cadenas de expresiones

Una expresion se puede separar mediante `;`. Siempre se toma en cuenta el último resultado.

8

```
1 2; 4; 8
```

Bloques

Para escribir varias líneas de código se utiliza el formato:

```
begin
    code
end
```

Lógica

Las expresiones lógicas del tipo `AND` y `OR` utilizan los siguientes operadores en *corto circuito*:

```
&& # AND
|| # OR
```

Para evaluar expresión bit a bit se utiliza:

```
& # AND
| # OR
```

Los operadores lógicos funcionan habitualmente. Con `a = 10` y `b = 5`:

```
a == b = false
a != b = true
a < b = false
a ≤ b = false
a > b = true
a ≥ b = true
```

El operador `==` compara el valor de los operandos. Para hacer una comparación *exacta* se hace uso del operador `===`:

```
5.0 == 5 = true
5.0 === 5 = false
```

Ciclo `while`

El ciclo while ocupa la sintáxis:

```
while (bool)
    code
end
```

```
begin
    i_1 = 1
    while i_1 <= 5
        print("$i_1 ")
        i_1 = i_1 + 1
    end
end
```

```
1 2 3 4 5
```

💡 En Julia, las identaciones no tienen un significado real, pero hacen legible el código. Además, un salto de línea es equivalente a `;`, pero tampoco tienen significado real.

Sentencia `if`

La sintáxis para la sentencia `if` es:

```
if bool1
    ...
elseif bool2
    ...
```

```
else
```

Por ejemplo:

```
"0 es par"
```

```
begin
    if x % 2 === 0
        "$x es par"
    else
        "$x es impar"
    end
end
```

```
@bind x Slider(0:10)
```

Arreglos

Los arreglos son colecciones de valores. Los arreglos pueden ser:

- Vectores: una dimensión.
- Matrices dos dimensiones.
- n-dimensionales.

> 💡 Los arreglos siempre favorecen la unicidad de los tipos de datos que contienen.

Vectores

Los vectores son arreglos de una dimensión. P. ej. un vector $\vec{v}$ con tres elementos:

```
v = [1.0, 2.0, 3.0]
```

```
v = [1,2.0,3]
```

Para recuperar los valores del vector, indexamos `v` con corchetes. La palabra `end` refiere al último elemento del arreglo.

> 💡 En Julia los índices de un arreglo comienzan en 1.

```
v[1] = 1.0
v[end] = 3.0
```

Los arreglos también pueden contener arreglos:

```
v2 = [1, [2.0, 3.0], 4.0]
```

Podemos indexar al arreglo dentro del arreglo:

```
v2[2][2] = 3.0
```

💡 Para obtener la longitud de un arreglo utilizamos la función `length`.

```
length(v2) = 3
```

Rangos

Para crear una secuencia de números utilizamos la sintáxis:

```
a:b # Rango desde a hasta b, inclusive, con paso 1
a:c:b #Rango desde a hasta b, inclusive, con paso c
```

Los rangos ocupan poco espacio en memoria. Para crear un arreglo con los rangos, utilizamos la función `collect`.

```
v3 = [10, 8, 6, 4, 2, 0]
```

```
1 v3 = collect(10:-2:0)
```

Con los rangos, podemos indexar elementos no contiguos de un arreglo.

```
v3[10:-3:0] = [0, 6]
```

Matrices

Las matrices son arreglos de dos dimensiones. Es posible ingresar una matriz como una tabla utilizando espacios para separar tokens:

```
[a11 a12 ... a1n
 a21 a22 ... a2n
 ..............
 am1 am2 ... amn]
```

Sea A la matriz

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
A = 2×2 Matrix{Int64}:
 1  2
 3  4
```

```
A = [1 2
     3 4]
```

Para recuperar los valores de la matriz utilizamos dos índices:

```
A[2,2] = 4
```

Para recuperar una fila o columna completa utilizamos el rango `:`

```
A[1,:] = [1, 2]
```

> 💡 Los datos en memoria se almacenan de manera contigua. El *column major order* indica que los datos se leen renglon por columna.

```
2
```

```
A[3]
```

Indexar arreglos con arreglos

Es posible indexar un arreglo con un arreglo o número, que reemplazan los índices originales. P. ej., para acceder a la primera columna de `A`

```
[1, 3]
```

```
A[[1,2], 1]
```

También podemos indexar con un arreglo único, que tiene la misma forma del arreglo que se desea obtener.

```
array[subarray]
```

Cada entrada de `subarray` representa un índice según el *column major order*.

```
2×2 Matrix{Int64}:
 4  2
 3  1
```

```
A[[4 3
   2 1]]
```

Concatenaciones

En Julia la combinación de vectores da lugar a concatenaciones verticales u horizontales:

- Vertical: se consigue con un `salto de línea` o con `;`. La forma explícita es `vcat`.
- Horizontal: se consigue con `espacio en blanco`. La forma explícita es `hcat`.

Por ejemplo una concatenación horizontal más vertical:

```
2×2 Matrix{Int64}:
 6  7
 8  9
```

```
[[6 7]; [8 9]]
```

Un caso especial es la concatenación horizontal de dos vectores. Julia convierte los vectores en *vectores columna*, y los concatena horizontalmente para formar una matriz:

```
2×2 Matrix{Int64}:
 6  8
 7  9
```

```
[[6, 7] [8, 9]]
```

Tuplas

Las tuplas son similares a los vectores, salvo que se consideran de distinto tipo y *no pueden ser modificadas*.

```
(a, b, ...) # Tupla
```

```
(1, 2, 3, 4)
```

```
(1, 2, 3, 4)
```

> 💡 Una tupla de 1 elemento debe ser declarada con una coma: `(x,)`.

Pertenencia

Para comprobar si un elemento pertence a una colección, utilizamos el operador ∈, o en su defecto, su negación ∉. Comprobemos si 3 pertenece a A:

```
true
```

```
3 ∈ A
```

Comprobemos si 5 *no pertenece a* A:

```
true
```
```
1 5 ∉ A
```

Cadenas y caractéres

En Julia se distinguen los tipo `string` y `char`:

- `string`: ocupan doble comilla `(" ")`.
- `char`: ocupan comilla simple `(' ')`.

Char

En Julia, todo símbolo UNICODE puede ser un `Char`. Los caractéres tienen *algunas* características de número, pero no lo son.

> 💡 Los caractéres en Julia tienen un orden secuencial, por lo que es posible calcular la *distancia entre ellos*, sumar números para obtener un carácter posterior, etc.

Veámos si `'A' < 'a'`:

```
true
```
```
1 'A' < 'a'
```

¿Qué caracter sucede a `'z'`?

```
'{': ASCII/Unicode U+007B (category Ps: Punctuation, open)
```
```
1 'z' + 1
```

¿Cuál es la distancia entre `'A'` y `'a'`?

```
32
```
```
1 'a' - 'A'
```

Strings

Las cadenas se escriben con comillas dobles. Para concatenar tipos `Char` o `Strings` utilizamos el operador `*`:

```
cadena = "François"
```
```
1 cadena = "Fran" * "çois"
```

Podemos indexar una cadena, pero no es tan simple. Los caractéres que componen la cadena *podrían* ocupar más de un bit en memoria, e indexar una cadena cuenta bit a bit. Por ejemplo, en `cadena`, el símbolo 'ç´ ocupa dos bits, por lo que el símbolo 'o' ocupa el índice 7:

```
'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
```

```
cadena[7]
```

Podemos preguntar la *pertenencia* de un carácter en una cadena con el operador ∈:

```
true
```

```
'ç' ∈ cadena
```

Para preguntar la pertenencia de una cadena en otra, utilizamos la función `occursin`:

```
true
```

```
occursin("çois", cadena)
```

Para imprimir en consola varias líneas de texto utilizamos tres comillas dobles:

```
print("""
Texto con
varias lineas...
""")
```

```
Texto con
varias lineas...
```

Ciclo `for`

El ciclo `for` ocupa la siguiente sintáxis:

```
for i in collection
    code
end
```

> 💡 Alternativamente se puede utilizar el símbolo `=` en lugar del keyword `in`, o, incluso, el símbolo `∈`.

```
for i in 1:5
    print(i)
end
```

```
12345
```

Es posible evitar el uso de ciclos anidados añadiendo más índices con su colección. Por ejemplo dos ciclos anidados ocuparía la sintáxis

```
for i in collection1, j in collection2
    code
end
```

```
for i ∈ 2:-1:1, j ∈ 1:2
    println("$i + $j = $(i+j)")
end
```

```
2 + 1 = 3
2 + 2 = 4
1 + 1 = 2
1 + 2 = 3
```

Como se menciona, el ciclo `for` puede iterar cualquier colección. Por ejemplo, la matriz A:

```
for i ∈ A
    print("$i ")
end
```

```
1 3 2 4
```

El ciclo también itera de manera lícita un `string`:

```
for i in cadena
    print("$(i)_")
end
```

```
F_r_a_n_ç_o_i_s_
```

Funciones

Las funciones simples en Julia pueden ocupar una sintáxis simple:

```
cuadrado (generic function with 1 method)
```

```
cuadrado(x,) = x ^ 2
```

Funciones más complejas ocupan la sintáxis

```
function name(args)
    code
end
```

Por ejemplo la distancia entre un punto $P(x, y)$ desde el origen:

```
distancia (generic function with 1 method)
```

```
function distancia(x,y)
    sqrt(x^2 + y^2)
end
```

Las funciones devuelven la última expresión evaluada, o la última sentencia con el operador `return`. Después de `return` la función se detiene.

```
distancia_positiva (generic function with 1 method)
```

```
function distancia_positiva(x,y)
    if x < 0 || y < 0
        return "Distancias positivas"
    else
        sqrt(x^2 + y^2)
    end
end
```

Las funciones también pueden recibir funciones como argumentos. Por ejemplo, una función que anuncie el resultado de otra función:

```
anunciar (generic function with 1 method)
```

```
function anunciar(f, x)
    print("El resultado es: ")
    f(x)
end
```

> 💡 Las *funciones puras* no generan efectos secundarios (crear archivos, imprimir en terminal, descargar de internet).

Composición de funciones

En Julia podemos realizar composición de funciones como en matemáticas:

```
81
```

```
cuadrado(cuadrado(3))
```

También se puede utilizar el símbolo ∘:

```
81
```

```
(cuadrado ∘ cuadrado)(3)
```

Una tercera opción indica utiliza |> para indicar cómo funciona el procesamiento (izquierda a derecha):

```
81
```

```
3 |> cuadrado |> cuadrado
```

Funciones *anónimas*

Las funciones anónimas se utilizan cuando una función recibe como argumento otra función. La función interna resulta desechable, pues solo se necesita mientras la función externa esté en cómputo. La sintáxis es:

```
(x,y,...)->f(x,y,...) # A (x,y,...) se le asigna el valor definido por f
```

Broadcasting (difusión)

En Julia se incorporó el operador `humble dot`, y se utiliza para conseguir que una función evalúe un arreglo de elementos valor por valor. Sea `f` una función:

```
f.[array] # Devuelve [f(array[1]), f(array[2]), ...]
```

Por ejemplo

```
▸ [1, 4, 9]
```

```
cuadrado.([1,2,3])
```

Alcance de variables

Se distinguen las variables globales de las variables locales. Las globales tienen posibilidad de uso en todo el programa.

> 💡 En un programa como archivo, es recomendable utilizar el keyword `cons` para evitar modificaciones accidentales.

Todas las variables definidas fuera de un bloque se consideran globales. Los bloques como `begin` no generan variables locales.

Alcance en funciones

Hay que tener un cuenta lo siguiente dentro de una función:

- Siempre se utilizará una variable local, si existe. Si no, se utilizará una variable global, si existe.
- *Toda* asignación crea una variable local.
- Si se requiere utilizar una variable global, se utiliza el decorador `global`.

Alcance en ciclos

Las reglas para los ciclos son las mismas, sin embargo, cuando se incluye un programa desde un archivo que incluye un ciclo con una variable cuyo nombre es igual a una variable global que ya existe, **se lanza un mensaje de advertencia sobre lo que sucede con las variables**.

> 💡 Soluciones para evitar la ambigüedad son: añadir decoradores como `global` o `local`, o colocar las variables y ciclos dentro de funciones, para evitar la existencia de variables globales.

Cuando se interactúa en el REPL o Pluto, no se lanza ningún mensaje de advertencia. En los entornos interactivos las variables globales son de uso común.

Mutabilidad

La mutabilidad implica la modificación, cambio o evlución de un objeto. Los arreglos se consideran objetos mutables, ya que son objetos con su propia dirección de memoria, por lo tanto:

```
true
```

```
begin
    w1 = [1]
    w2 = w1
    w2[1] = 5
    w1 === w2
end
```

Pero los arreglos sin la misma dirección de memoria, siendo estrictos, son diferentes:

```
[1] === [1]: false
[1] == [1]: true
```

> 💡 Los números *no* son mutables, ya que no se ubican los números en memoria. Estas variables son nombres, pero no los valores en sí mismos.

Funciones `push!` y `pop`

Para mutar un vector añadiendo un nuevo elemento se puede utilizar la siguiente sintáxis:

```
vector = [vector, a] # Añade a al final del vector
```

Sin embargo, esto es ineficiente para grandes cantidades de datos. En su lugar, utilzamos la función `push!`, que devuelve el arreglo mutado:

```
vector = push!(vector, a) # Añade a al final del vector
```

La función opuesta a `push!` es `pop!`, que elimina el último elemento de un vector y devuelve tal elemento.

```
pop!(vector) # Devuelve el último elemento
```

Strings

Los strings son objetos inmutables, es decir, no se pueden modificar directamente. Para manejar un string es buena idea realizar concatenaciones

```
string = string * new
```

Es mejor idea realizar concatenaciones con la función `push!`. Para unir elementos de un arreglo de strings se utiliza la función `join`:

```
join(arreglo) # Unir en un string un arreglo de strings
```

La función contraria a `join` es `split`, que separa los caracteres de un string en un arreglo:

```
split(string, "chars") # Separar string segun aparezca "chars"
```