**Modulo 2.** Conceptos básicos

El objetivo es enseñar a programar en Julia, por ende, podemos abarcar conocimientos muy básicos, sin ahondar en otros, y a la par mostrar característica de Julia. Así que lo mejor, para terminar rápido y ser conciso en temas, hablar sobre:

- Tipos de datos

- Operadores

- Variables

- Estructuras de control

- Funciones

- Estructuras de datos (vectores y constructores)

- Essentials (otros)

- Algún otro tema que se necesario y cubra lo básico, ya que este módulo es para cubrir lo básico, sin necesidad de ahondar.

Cabe revisar este apartado -> [Mathematics · The Julia Language](https://docs.julialang.org/en/v1/base/math/) (base) y de allí sacar información importante.

**Tabla de contenido**

[1 Tipos de datos 2](#_Toc125134544)

[1.1 Datos primitivos 3](#_Toc125134545)

[1.2 Tipos de datos en Julia 3](#_Toc125134546)

[1.2.1 Enteros y puntos flotantes 4](#_Toc125134547)

[1.2.1.1 Enteros 4](#_Toc125134548)

[1.2.1.2 Punto flotante 6](#_Toc125134549)

[1.2.1.3 Otros conceptos (Aritmética de precisión arbitraria, Coeficientes de literales numéricos) 8](#_Toc125134550)

[1.2.2 Números complejos y racionales 9](#_Toc125134551)

[1.2.3 Caracteres y Strings 10](#_Toc125134552)

[1.2.3.1 Características de las strings 10](#_Toc125134553)

[1.2.3.2 Operaciones 11](#_Toc125134554)

[2 Tipos de operadores y operaciones matemáticas 13](#_Toc125134555)

[2.1.1 Aritméticos 13](#_Toc125134556)

[2.1.2 Lógicos o Booleanos 15](#_Toc125134557)

[2.1.3 Operadores de comparación 16](#_Toc125134558)

[2.1.4 Operadores a nivel de bit 16](#_Toc125134559)

[2.1.5 Operadores de actualización 17](#_Toc125134560)

[3 Variables 19](#_Toc125134561)

[3.1 Estructura de definición 19](#_Toc125134562)

[3.2 Clasificación de las variables según su alcance 21](#_Toc125134563)

[3.3 Constantes 22](#_Toc125134564)

[3.3.1 Particularidades a tener en cuenta con las constantes 23](#_Toc125134565)

[4 Estructuras de control 24](#_Toc125134566)

[4.1.1 Expresiones compuestas 24](#_Toc125134567)

[4.1.2 Condicionales y evaluaciones lógicas cortas 24](#_Toc125134568)

[4.1.3 Iteraciones-ciclos 25](#_Toc125134569)

[4.1.4 Manejo de excepciones 25](#_Toc125134570)

[4.1.5 Tareas-corutinas 25](#_Toc125134571)

[5 Funciones 25](#_Toc125134572)

[5.1 Estructura y caracteristicas 25](#_Toc125134573)

[6 Estructuras de datos 25](#_Toc125134574)

[6.1 Arreglos 26](#_Toc125134575)

[6.2 Constructores 26](#_Toc125134576)

[7 Interfaces 26](#_Toc125134577)

[8 Essentials 27](#_Toc125134578)

# Tipos de datos

Fuente: [Integers and Floating-Point Numbers · The Julia Language](https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/)

En las computadoras actuales se pueden identificar distintos tipos de información, desde datos primitivos, es decir datos atómicos (que están en su forma mínima; en un contexto de funcionamiento de un programa, no son reducibles, por ejemplo, los números enteros), hasta estructuras compuestas estos últimos (lo que se conoce como estructuras de datos o datos abstractos).

## Datos primitivos

En el campo de la programación a lo largo del gran número de lenguajes existentes se encontrarán diferentes formas de datos las cuales se basan cada uno en una idea general. A continuación, se clasificarán los tipos de datos primitivos o naturales dentro del campo con el fin de tener una idea general sobre lo que se puede esperar dentro de cualquier lenguaje.

* **Enteros:** Conocidos por su traducción al inglés como *integer* o *int*, representan una cantidad numérica entera positiva o negativa. Existe versiones para representar enteros con signo y otras sin signo, la diferencia radica; además de la presencia del signo; en la cantidad de memoria usada para representar cada uno y el nombre (*uinteger* para entero sin signo).
* **Flotantes:** Representan a los números con decimales. Se destacan por precisar la magnitud de un número, si es grande o pequeño en comparación con otros. Se les conoce con el nombre de *float.*
* **Booleanos:** Representan los valores lógicos verdadero y falso. En programación suelen representarse como 0’s o 1’s, al igual que utilizando las convenciones de ***true*** o ***false*** respectivamente. Los operadores lógicos y de comparación producen resultados de este tipo. Son usados mayormente en estructuras de control como el *for, while* o *if*. Se conocen por el nombre de *bool*.
* **Caracteres y cadenas de texto:** Un carácter es la forma más básica de representación de las letras y todo lo relacionado a texto y símbolos. El conjunto de uno o más caracteres conforman una cadena de texto. Existen lenguajes que permiten el uso de ambos por separado. Ambos se conocen como *char* y *string*.

Se podrán encontrar otros tipos de datos como doubles, longs, shorts, structs, o datos propios del lenguaje. Al final, todos cumplen con que internamente se basan en las características de los datos primitivos mencionados.

## Tipos de datos en Julia

Julia a nivel interno maneja dos tipos de representación numérica:

* **Numérica primitiva**: representación binaria dentro de la memoria del computador.
* **Numérica literal**: valores inmediatos como 1 (Entero literal) o 2.0 (flotante literal)

Esta diferenciación puede ayudar más adelante entiendo otros conceptos y herramientas aplicables a una u otra representación.

### Enteros y puntos flotantes

#### Enteros

El lenguaje maneja los siguientes tipos de enteros:

|  |  |  |  |
| --- | --- | --- | --- |
| Tipo | Con signo | Valor más pequeño | Valor más grande |
| Int8 | ✓ | -2^7 | 2^7 - 1 |
| UInt8 |  | 0 | 2^8 - 1 |
| Int16 | ✓ | -2^15 | 2^15 - 1 |
| UInt16 |  | 0 | 2^16 - 1 |
| Int32 | ✓ | -2^31 | 2^31 - 1 |
| UInt32 |  | 0 | 2^32 - 1 |
| Int64 | ✓ | -2^63 | 2^63 - 1 |
| UInt64 |  | 0 | 2^64 - 1 |
| Int128 | ✓ | -2^127 | 2^127 - 1 |
| UInt128 |  | 0 | 2^128 - 1 |
| Bool | N/A | false (0) | true (1) |

Todos los tipos parten de la representación de un entero, siendo esto con signo (Int) o sin signo (UInt), con la diferencia de que hay tantos como cantidad de bytes manejables por computadores convencionales existen, esto es, desde los 8 bits hasta los 128 bits (un byte es igual a 8 bits).

Por defecto los datos se adaptan al sistema en el cual se ejecutan, por ejemplo:

|  |
| --- |
| julia> Sys.WORD\_SIZE |
| 64 |
|  |
| julia> Int |
| Int64 |
|  |
| julia> typeof(3) |
| Int64 |

Sys.WORD\_SIZE es una función que permite conocer cuántos bits maneja una computadora. Es decir que la computadora que ejecutó dicha instrucción maneja 64 bits de base.

Conociendo ello, se observa en la siguiente línea que con solo indicar el tipo de dato Int, el lenguaje nos da como respuesta el tipo de dato por defecto en uso y el cual concuerda con el tamaño de bits manejado por la computadora. Por otra parte, utilizando una función *typeof* que indica el tipo de dato de algún elemento, se determina que el numero 3 es un tipo de dato entero con signo de 64 bits.

Se puede presentar la situación en que se quiera usar una categoría específica de los enteros, para ello, se puede indicar explícitamente de la siguiente manera:

|  |
| --- |
| julia> x = Int32(65) |
| 65 |
|  |
| julia> typeof(x) |
| Int32 |
|  |
| julia> x = Int128(100) |
| 100 |
|  |
| julia> typeof(x) |
| Int128 |

Julia determinará el tipo de dato que mejor se ajuste al valor a representar si aquella información no se puede almacenar según el rango de información permitido por el valor base dado por el sistema (32 o 64 bits):

|  |
| --- |
| julia> typeof(11112222333344445555666677778888) |
| Int128 |

Otra gran característica es que se permita representaciones en otros sistemas diferentes al decimal como el binario, Octal y hexadecimal:

|  |
| --- |
| julia> bin = 0b11 #en decimal es 3 |
| 0x03 |
|  |
| julia> bin = 0b110 #en decimal esto es 6 |
| 0x06 |
|  |
| julia> typeof(bin) |
| UInt8 |
|  |
| julia> hex = 0x1234 #en decimal esto es 4660 |
| 0x1234 |
|  |
| julia> typeof(hex) |
| UInt16 |
|  |
| julia> oct = 0o34 |
| 0x1c |
|  |
| julia> typeof(oct) |
| UInt8 |

#### Punto flotante

Este tipo de dato contiene las siguientes categorías:

|  |  |  |
| --- | --- | --- |
| Tipo | Precisión | Número de bits |
| Float16 | half | 16 |
| Float32 | single | 32 |
| Float64 | double | 64 |

Los términos de precisión son dados gracias a un estándar de nomenclatura (IEEE 754 o Standard for Floating-Point Arithmetic) propuesto por la IEEE (Institute of Electrical and Electronics Engineers en inglés) en el año de 1985.

Algunos ejemplos de estos son:

|  |
| --- |
| julia> 1.0 |
| 1.0 |
|  |
| julia> 2. |
| 2.0 |
|  |
| julia> 0.5 |
| 0.5 |
|  |
| julia> .890 |
| 0.89 |
|  |
| julia> -0.456 |
| -0.456 |
|  |
| julia> 1e10 |
| 1.0e10 |
|  |
| julia> 2e-5 |
| 2.0e-5 |

Algunas características presentes en los números de tipo flotante:

* Conversión a flotantes de 32 bits (Float32):

|  |
| --- |
| julia> x = Float32(45) |
| 45.0f0 |
|  |
| julia> typeof(x) |
| Float32 |
|  |
| julia> typeof(90.56f0) |
| Float32 |
|  |
| julia> typeof(90.56f-4) |
| Float32 |

* Representación de hexadecimales

|  |
| --- |
| julia> 0x1p0 |
| 1.0 |
|  |
| julia> 0x1.8p3 |
| 12.0 |
|  |
| julia> x = 0x.4p-1 |
| 0.125 |

* Existe una representación[[1]](#footnote-1) para valores infinitos, esto permite mantener coherencia con ciertas las operaciones aritméticas que involucran conceptos de no-finito:

|  |
| --- |
| julia> 1/Inf |
| 0.0 |
|  |
| julia> 1/0 |
| Inf |
|  |
| julia> -5/0 |
| -Inf |
|  |
| julia> 500 + Inf |
| Inf |
|  |
| julia> 0/0 |
| NaN |

#### Otros conceptos (Aritmética de precisión arbitraria, Coeficientes de literales numéricos)

**Aritmética de precisión arbitraria**

Es una rama del campo de las ciencias de la computación encargado de los números extremadamente grandes, o también conocidos como números de precisión infinita. Son muy utilizados en procesos de ingeniería donde se necesiten de operaciones aritméticas entre valores demasiado grandes con resultados de alta precisión. Julia en su objetivo de ser un lenguaje para representación matemática y trabajo con datos de la manera más rápida posible incorpora el concepto de precisión infinita o arbitraria mediante el uso de tipos de datos especiales como BigInt y BigFloat para los datos de tipos entero y punto flotante respectivamente.

Algunos ejemplos son:

|  |
| --- |
| julia> BigInt(typemax(Int64)) + 1 |
| 9223372036854775808 |
|  |
| julia> big"1.23456789012345678901" |
| 1.234567890123456789010000000000000000000000000000000000000000000000000000000004 |
|  |
| julia> factorial(BigInt(40)) |
| 815915283247897734345611269596115894272000000000 |

No se profundizará sobre este tema, más sin embargo es importante resaltar esta característica en caso tal de que se requieran, además no muchos lenguajes de uso convencional suelen tener estos mecanismos incorporados.

**Coeficientes de literales numéricos**

En el intento de ser más fieles a como se escriben las expresiones matemáticas, Julia provee de la capacidad de presentar los números antepuestos a una variable (represntando la multiplicación), igual a como sucede en los polinomios:

|  |
| --- |
| julia> x,y = 3,4 |
| (3, 4) |
|  |
| julia> 3x + 2y - (5x/8^y) |
| 16.996337890625 |

### Números complejos y racionales

Fuente: [Complex and Rational Numbers · The Julia Language](https://docs.julialang.org/en/v1/manual/complex-and-rational-numbers/)

De manera general, los **números complejos** son aquellos que intentan representar una unidad imaginaria conocida como (representa ). Son muy utilizados en diferentes campos de la matemática e ingeniería (cálculos de aerodinámica, fluidos, ecuaciones diferenciales), así como en la física con la descripción de la mecánica cuántica.

Julia permite su representación mediante el uso del indicador im después de cualquier número. Por ejemplo:

|  |
| --- |
| julia> (4 + 6im)\*(1 - 9im) |
| 58 - 30im |
|  |
| julia> (1 + 2im)/(1 - 2im) |
| -0.6 + 0.8im |

**Nota:** La representación de un imaginario se basa en una parte real (a) y una parte imaginaria (bim) para luego unirse mediante una suma una resta (a +- bim).

El conjunto de los complejos vienen acompañados de funciones dedicadas como complex que construye numeros imaginarios a partir de Su parte real y su parte imaginaria, real e imag para obtener la información anterior por separado, entre muchas otras.

Por otra parte, los **números racionales** también pueden ser expresados en Julia, permitiendo así una sintaxis cercana a las expresiones matemáticas hechas a mano:

|  |
| --- |
| julia> 3//4 |
| 3//4 |
| julia> typeof(3//4) |
| Rational{Int64} |

Junto a esto se pueden hacer diferentes combinaciones entre diferentes tipos de datos:

|  |
| --- |
| julia> 5//0 |
| 1//0 |
|  |
| julia> 2//7 \* (1 + 2im) |
| 2//7 + 4//7\*im |
|  |
| julia> 1//3 - 0.33 |
| 0.0033333333333332993 |

### Caracteres y Strings

Fuente: [Strings · The Julia Language](https://docs.julialang.org/en/v1/manual/strings/)

Como ya se indicó anteriormente, los caracteres son un símbolo, la representación de un solo elemento mientras que las Strings son conjuntos de caracteres. En Julia, ambos tipos de datos están diseñados para poder manejar el estándar UTF-8 que permite definir cada representación a un único valor utilizable por las computadoras (hay diferentes estándares de codificación).

Para su manejo dentro del lenguaje se tiene que los caracteres utilizan comilla simple y las strings comillas dobles:

|  |
| --- |
| julia> myChar = 'h' |
| 'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase) |
|  |
| julia> myStr = "Hello" |
| "Hello" |
|  |
| julia> typeof(myChar) |
| Char |
|  |
| julia> typeof(myStr) |
| String |

#### Características de las strings

Información útil al momento de manejar este tipo de datos es:

* Cuando se tienen strings de gran longitud, para mejor redacción se puede utilizar lo siguiente:

|  |
| --- |
| julia> "This is a long \ |
| line" |
| "This is a long line" |

* Para acceder a la información dentro de una cadena, es decir, un carácter en una posición especifica, basta con utilizar sus índices:

|  |
| --- |
| julia> myStr = "Esto es una cadena" |
| "Esto es una cadena" |
|  |
| julia> myStr[1] |
| 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase) |
|  |
| julia> myStr[2] |
| 's': ASCII/Unicode U+0073 (category Ll: Letter, lowercase) |
|  |
| julia> myStr[1:length(myStr)] |
| "Esto es una cadena" |
|  |
| julia> myStr[1:5] |
| "Esto " |

Nótese que hay expresiones como myStr[1:length(myStr)] o myStr[1:5]

Dan como resultado un sub-cadena de la cadena myStr. Esta sintaxis, conocida en la programación como slices permite definir un rango dentro de una cadena, y en general de un elemento indexable. La notación de modo general podría verse como cadena[posición del elemento inicial: posición del elemento final]. La primera expresión indica que se quiere obtener la información de myStr desde el elemento en la posición 1 hasta el elemento en la posición lenght(myStr). Lenght() es una función, concepto que se desarrollará más adelante, que permite conocer la longitud de un elemento que en este caso sería una cadena, por ende, la posición final dentro de la definición del slice sería aquella que concuerde con el valor arrojado por dicha función. De manera similar, la segunda expresión indicaría una sub-cadena compuesta por el elemento en la posición 1 dentro de myStr y continuando hasta aquel en la posición 5.

**Nota:** Según la cadena presentada puede darse el caso de que no todas las posibilidades de índices sean correspondientes a un elemento de dicha cadena. En este caso, se arrojará un error por parte del lenguaje. Para consultar sobre ello dirigirse a [Strings en Juli con unicode y UTF-8](https://docs.julialang.org/en/v1/manual/strings/#Unicode-and-UTF-8).

* Se permite el uso de cadenas de caracteres con comillas triples “””, estas presentan un funcionamiento especial que puede servir para encerrar textos demasiados largos, o inclusive código con su respectiva identación:

|  |
| --- |
| julia> str = """ |
| Esto puede ser |
| codigo. |
| """ |

El Código anterior retorna como resultado la cadena " Esto puede ser\n código\n" , nótese que la palabra código va con un espacio demás, esto indicaría un primer nivel de identación.

**Nota:** Los caracteres ‘\n’ son llamados caracteres de escape. Permiten representar acciones como salto de línea, espacios, tabulaciones, entre otras. En esta caso, ‘\n’ hace referencia a un salto de línea al momento de imprimir la cadena de texto, o inclusive el carácter.

#### Operaciones

Algunas de las operaciones más comunes y útiles son:

|  |  |
| --- | --- |
| Operación | Concepto-ejemplo |
| Operadores de comparación | “Hola” == “Adios”  “Hola” > “Adios”  A nivel interno, las cadenas también son numeros gracias a estándares de codificación como el UTF-8. Con ello, sería como hacer comparaciones entre numeros.  El resultado de las comparaciones se hace con base a la codificación de cada carácter, es decir, si la codificación de una letra, es decir, su número, es mayor, menor, igual o diferente que otro de otra letra. |
| findfirst(carácter, cadena) findlast(carácter, cadena) | findfirst('a', "Cadena") retorna 2, lo que indican que la posición de la primera aparición del carácter ‘a’ es la 2.  findlast('a', "Cadena") retornar 6, lo que indica que la posición de la última aparición del carácter ‘a’ es la 6. |
| occursin(subcadena, cadena) | occursin(“como estas”,”Hola tú, como va todo, como estas?”) retorna true, un valor booleano que me indica que la sub-cadena si se encuentra dentro de la cadena. |
| lastindex(cadena) | lastindex(“Ejemplo”) retorna 7 ya que el ultimo índice posible dentro de la cadena indicada, es decir, la posición del último elemento, es la 7. |
| length(cadena) | Número de caracteres (longitud de una string) |
| length(cadena, posición inicial, posición final) | Retorna el número de índices posibles dentro del rango indicado (incluyéndolos). Por ejemplo:  length("Hello", 1 ,length("Hello")) retorna 5 ya que las posibilidades son:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | H | e | l | l | o | | 1 | 2 | 3 | 4 | 5 | |
| Concatenación mediante string() o ‘\*’ | Para juntar dos o más cadenas se puede hacer lo siguiente:  string("Hola"," " ,"como"," ", "estas") retorna "Hola como estas"  "Hola"\*" como "\*"estas" retorna "Hola como estas"  Particularmente la segunda forma utiliza un asterisco ya que trae beneficios al momento de trabajar con algebra abstracta |
| Interpolación | Situaciones como:  "1 + 2 + 3 + 4 = $(1+2+3+4)" retorna  "1 + 2 + 3 + 4 = 10"  julia> greet = "Miguel"  "Miguel"  julia> "Hola señor $greet"  "Hola señor Miguel"  Muestran la creación y concatenación de cadenas de una manera más sencilla mediante el uso del carácter $. |
| Repeat(cadena, número de repeticiones) | repeat("--(:3)--",2) retorna "--(:3)----(:3)--".  Como su nombre indica, repite la cadena indicada el número de veces proporcionado. |

# Tipos de operadores y operaciones matemáticas

Fuente: [Mathematical Operations and Elementary Functions · The Julia Language](https://docs.julialang.org/en/v1/manual/mathematical-operations/)

Al igual que las matemáticas en donde se utilizan ciertas operaciones representadas por cierta simbología como lo son la suma (+), resta (-), multiplicación (\*), división (/), o incluso operaciones más complejas como derivadas e integrales, la programación también maneja su propio conjunto de símbolos comúnmente llamados operadores. El nombre permite conocer cuál será la función que representa dicha simbología, y en este caso se trata de poder ejecutar operaciones entre uno, dos o más elementos.

### Aritméticos

Representan las operaciones elementales (sumas, resta, multiplicación y división) y otras operaciones básicas entre números.

|  |  |  |
| --- | --- | --- |
| Expresión | Nombre | Función |
| +x | Suma unaria | Indicar que un número es positivo |
| -x | Resta unaria | Indicar que un número es negativo |
| x + y | Suma | Ejecutar la suma entre dos números |
| x - y | Resta | Ejecutar la resta entre dos números |
| x \* y | Multiplicación | Ejecutar la multiplicación entre dos números |
| x / y | División flotante | Ejecutar la división entre dos números y el resultado es un dato de tipo flotante |
| x ÷ y | División entera | Ejecutar la división entre dos números y el resultado es un dato de tipo entero |
| x \ y | División inversa | Equivalente a y / x. Su resultado devuelve un dato de tipo flotante. |

Algunos ejemplos son:

|  |
| --- |
| julia> +3 |
| 3 |
|  |
| julia> -3 |
| -3 |
|  |
| julia> 3 + 4 |
| 7 |
|  |
| julia> 3-4 |
| -1 |
|  |
| julia> 3 \* 4 |
| 12 |
|  |
| julia> 3 / 4 |
| 0.75 |
|  |
| julia> 3 ÷ 4 |
| 0 |
|  |
| julia> 3 \ 4 |
| 1.3333333333333333 |

### Lógicos o Booleanos

Determinan las comparaciones usadas en la lógica de proposiciones como el *Y* (AND) o el *O* (OR), *Negación* (NOT) o el *O exclusivo* (XOR).

|  |  |
| --- | --- |
| Expresión | Nombre |
| !x | Negación |
| x && y | Conjunción o AND (Y en español) |
| x || y | Disyunción o OR (O en español) |

Algunos ejemplos son:

|  |
| --- |
| julia> !true |
| False |
|  |
| julia> true || false |
| True |
|  |
| julia> false && false |
| false |

Cada operando maneja con un conjunto de valores por defecto:

|  |  |  |
| --- | --- | --- |
| X | Y | X && Y |
| Verdad | Verdad | Verdad |
| Verdad | Falso | Falso |
| Falso | Verdad | Falso |
| Falso | Falso | Falso |

|  |  |  |
| --- | --- | --- |
| X | Y | X || Y |
| Verdad | Verdad | Verdad |
| Verdad | Falso | Verdad |
| Falso | Verdad | Verdad |
| Falso | Falso | Falso |

|  |  |
| --- | --- |
| X | ¡Y |
| Verdad | Falso |
| Falso | Verdad |

### Operadores de comparación

Cuando se habla de comparación, se hace referencia a si un elemento es más grande, pequeño, igual o diferente que otro. Su representación es:

|  |  |  |
| --- | --- | --- |
| Nombre | Símbolo | Función |
| Igual que | == | Determina si dos elementos son iguales |
| No igual o distinto de | =! | Determina si dos elementos son diferentes |
| Menor que | < | Si un elemento a es menor que un elemento b |
| Menor o igual que | <= | Determina si es un elemento a es menor o igual que un elemento b |
| Mayor que | > | Si un elemento a es mayor que un elemento b |
| Mayor o igual que | >= | Determina si es un elemento a es mayor o igual que un elemento b |

Algunos ejemplos son:

|  |
| --- |
| julia> 3 == 4 |
| False |
|  |
| julia> 3 != 3 |
| False |
|  |
| julia> 4 < 0 |
| False |
|  |
| julia> 2 <= 5 |
| True |
|  |
| julia> 3 > 2 |
| True |
|  |
| julia> 5 >= 5 |
| true |

### Operadores a nivel de bit

Como su nombre lo indica, estos operandos trabajan con las representaciones binarias de uno o más elementos.

|  |  |
| --- | --- |
| Expresión | Nombre |
| ~x | Not bit a bit |
| x & y | And bit a bit |
| x | y | Or bit a bit |
| x ⊻ y | Xor bit a bit |
| x ⊼ y | nand bit a bit |
| x ⊽ y | nor bit a bit |
| x >>> y | corrimiento lógico a la derecha |
| x >> y | Corrimiento aritmético a la derecha |
| x << y | Corrimiento lógico/aritmético a la izquierda. |

Algunos ejemplos son:

|  |
| --- |
| julia> ~123 |
| -124 |
|  |
| julia> 123 & 234 |
| 106 |
|  |
| julia> 123 | 234 |
| 251 |
|  |
| julia> 123 ⊻ 234 |
| 145 |
|  |
| julia> xor(123, 234) |
| 145 |
|  |
| julia> nand(123, 123) |
| -124 |
|  |
| julia> 123 ⊼ 123 |
| -124 |
|  |
| julia> nor(123, 124) |
| -128 |
|  |
| julia> 123 ⊽ 124 |
| -128 |

Como se observa en los ejemplos, se puede hacer uso de funciones con el nombre de la operación a realizar, como lo es el caso del *xor()* o el *nand()*.

### Operadores de actualización

Este tipo de operadores ayudan a cambiar el valor de algún elemento efectuando la operación que precede al signo igual y siguiendo el orden de asignación de valores x = y en donde x (lo que se sitúa a la izquierda) se le asigna y (lo que hay a la derecha), o lo que es en otras palabras, x es igual a y, por ejemplo, x += y es lo mismos que decir x = x + y, que se leería como “x será igual a la suma de su valor actual más el valor de y”.

|  |  |
| --- | --- |
| Expresión | Descripción |
| += | Suma a la izquierda lo que hay en la derecha |
| -= | Resta a la izquierda lo que hay en la derecha |
| \*= | Multiplica a la izquierda lo que hay en la derecha |
| /= | Divide (flotantes) a la izquierda lo que hay en la derecha |
| \= | Divide (flotantes) a la inversa la izquierda lo que hay en la derecha |
| ÷= | Divide (enteros) a la izquierda lo que hay en la derecha |
| %= | Efectúa el módulo entre lo que en la izquierda y la derecha para luego asignar ese resultado a lo que hay en la izquierda |
| ^= | Eleva a la izquierda lo que hay en la derecha |
| &= | Efectúa el operador binario AND entre la izquierda y la derecha para luego asignarlo a la izquierda |
| |= | Efectúa el operador binario OR entre la izquierda y la derecha para luego asignarlo a la izquierda |
| ⊻= | Efectúa el operador binario XOR entre la izquierda y la derecha para luego asignarlo a la izquierda |
| >>>= | Efectúa la operación de corrimiento lógico a la derecha entre la izquierda y la derecha para luego asignarlo a la izquierda |
| >>= | Efectúa la operación de corrimiento aritmético a la derecha entre la izquierda y la derecha para luego asignarlo a la izquierda |
| <<= | Efectúa la operación de corrimiento aritmético a la izquierda entre la izquierda y la derecha para luego asignarlo a la izquierda |

Algunos ejemplos son:

|  |
| --- |
| julia> x = 3 |
| 3 |
|  |
| julia> x += 1 |
| 4 |
|  |
| julia> x -= 2 |
| 2 |
|  |
| julia> x \*= 5 |
| 10 |
|  |
| julia> x /= 2 |
| 5.0 |
|  |
| julia> x |= 3 |
| 3 |
|  |
| julia> x ^= 3 |
| 27 |
|  |
| julia> x &= 3 |
| 3 |
|  |
| julia> x ⊻= 3 |
| 0 |
|  |
| julia> x >>>= 4 |
| 0 |
|  |
| julia> x <<= 2 |
| 0 |

Se recomienda analizar con detenimiento el resultado de ciertas operaciones que involucran convertir el número (el valor de x) a otras representaciones como lo hacen las operaciones a nivel de bit.

# Variables

Fuente: [Variables · The Julia Language](https://docs.julialang.org/en/v1/manual/variables/)

Son elementos que permiten almacenar información, algunos que pueden cambiar constantemente como otros que siempre tendrán un mismo valor asignado. La información almacenada puede ser de alguno de los tipos de datos vistos anteriormente.

## Estructura de definición

Cuando se usa un lenguaje de programación, será muy común escuchar oraciones como “*declarar la variable…*”, esto hace referencia al momento en el cual se indica dentro del lenguaje que se está haciendo uso de estos elementos. Algunos ejemplos de cómo se haría la declaración de variables en Julia son:

|  |
| --- |
| julia> myVar = 10 |
| 10 |
|  |
| julia> myVarInt = 100 |
| 100 |
|  |
| julia> myVarStr = "Hello world!" |
| "Hello world!" |
|  |
| julia> myVarFloat = 10.76890 |
| 10.7689 |
|  |
| julia> myVarBit = 0b110 |
| 0x06 |

El proceso de creación es fácil gracias a la característica de proporcionar sintaxis de alto nivel. Existen otros lenguajes en donde se manejas reglas de sintaxis más estrictas, esto quiere decir que el proceso de definición tanto de las variables como de otros elementos suele venir con muchas más pasos, un ejemplo muy común es comparar el famoso programa de *Hello world* en diferentes lenguajes, en este caso observe como sería mostrar el mensaje en pantalla mediante Julia y Java:

|  |  |
| --- | --- |
| Código en Julia | Código en Java |
| myVar = "Hello, world!"  println(myVar) | package Excercises;  public class PrintDemo {  public static void main(String[] args) {  String myVar = "Hello, world!";  System.out.println(myVar);  }  } |
| Resultado de ejecución:  Hello, world! | Resultado de ejecución:  Hello, world! |

Ambas formas obtienen el mismo resultado, con la diferencia de que aquella escrita en Java lleva más líneas de código y es más descriptiva en cuanto a las operaciones, mientras que aquella escrita en Julia es más rápida y concreta.

De manera general y usando la comparación anterior, algunas de las características que posee la sintaxis en Julia son:

* Tipado dinámico, es decir, no hay necesidad de indicar el tipo de dato que se almacena como sucede con String myVar = “Hello, world!”.
* Puede optarse por usar delimitadores de línea, esto es el uso del ‘;’.
* Los nombres usados para las variables son de caso sensitivo, esto quiere decir que importa el uso de mayúsculas y minúsculas.
* Se permite el uso de caracteres codificados según el estándar UTF-8, por ejemplo:

|  |
| --- |
| 🥹 = "Extremely Happy that I'm about to cry" |
| println(🥹) |
| > Extremely Happy that I'm about to cry |
|  |
| 中文變量 = "This is a chinese var name" |
| println(中文變量) |
| > This is a chinese var name |
|  |
| 須磨 = 45 + 2 |
| println(須磨) |
| > 47 |

## Clasificación de las variables según su alcance

Fuente: [Scope of Variables · The Julia Language](https://docs.julialang.org/en/v1/manual/variables-and-scoping/), [Scope of Variables · The Julia Language (mit.edu)](https://web.mit.edu/julia_v0.6.0/julia/share/doc/julia/html/en/manual/variables-and-scoping.html#Soft-Local-Scope-1)

El alcance de una variable se refiere a la región dentro de un código a la cual la variable es accesible, en otras palabras, hasta qué punto puedo acceder a una variable. Su propósito es evitar confusiones en situaciones donde se utilizan los mismos nombres, aunque se refieran a cosas diferentes.

Julia define las siguientes regiones de código (también llamados *scope blocks*) para las cuales una variable define su alcance:

|  |  |  |  |
| --- | --- | --- | --- |
| Elementos | Concepto | Tipo de alcance | Permitido dentro de |
| Modulos, baremodules | **Modulo:** Documento que contiene instrucciones definidas según Julia y que puede ser utilizado por otros documentos mediante llamados (acción también conocida como importes).  **Baremodule:** ver capitulo 9. | Global (soft) | Global |
| Struct | **Struct:** Permite crear tipos de datos, por ejemplo, un dato que represente una figura geométrica, una persona, un dirección. | Local (soft) | Global |
| For, while, try | Estructuras de control y flujo de datos.  **For:** permite ejecutar un conjunto de instrucciones un número de veces indicado.  **While:** Similar al Foren cuanto a ser estructura de iteración con la diferencia que este se usa para iterar según una condición**.** Ambas estructuras se verán con detalle en capítulos posteriores.    **Try:** Permite la intercepción de errores durante la ejecución de un programa. | Local (soft) | Local |
| Macro | **Macro:** Permite insertar código generado dentro de un programa. Ver más en el capítulo 9. | Local (hard) | Local |
| Do, let, comprehensions, generators | Herramientas que permiten hacer la escritura de código más rápida y eficiente. Estas herramientas se verán con detalle en capítulos posteriores. | Local (hard) | Local |

**Nota:** Esta clasificación servirá para cuando se conozco el funcionamiento de las estructuras de control, flujo y funciones.

Tanto Julia como cualquier otro lenguaje de programación podrá manejar dos tipos de alcance diferentes:

* **Global:** Variables que pueden ser usadas en todo momento y lugar.
* **Local:** Variables que pueden ser únicamente utilizadas en su ambiente de declaración, esto es, cada vez que se define un nuevo bloque de alcance como los mencionados en la tabla anterior.

Por otra parte, se introducen los conceptos de *soft* y *hard*, en donde ambos comparten características como el hecho de que hacen referencia a las variables que se pueden usar dentro de un nuevo bloque de alcance a menos que se especifique con la palabra *local*. Por otra parte, se diferencian en el tipo de estructuras que describen.

Estos dos términos serán trabajados de manera más detallada en la siguiente sección de este capítulo y en el capítulo 5 que habla de funciones.

## Constantes

Similar a lo que ocurre con las variables, elementos que almacenan información que no puede ser cambiada en ningún momento. Algunos ejemplos serían constantes matemáticas como pi (π) o el número de Euler (*e*).

En Julia las constantes se definen de la siguiente manera:

|  |
| --- |
| julia> const e = 2.71828182845; |
|  |
| julia> const pi = 3.141592653; |

### Particularidades a tener en cuenta con las constantes

Algunas datos a tener presentes cuando se usan estos elementos son:

* Declarar constantes de manera global como buena práctica de codificación y ayuda al compilador. Como lo mencionan en la documentación oficial “Es difícil para el compilador optimizar el código que involucra variables globales, ya que sus valores (o incluso sus tipos) pueden cambiar en casi cualquier momento.”
* Una constante puede almacenar información sobre un elemento que puede cambiar, como por ejemplo estructuras de datos como arreglos. Esto no impide que la información en el cambie.

|  |
| --- |
| julia> const myVec = [1,2,3] |
| 3-element Vector{Int64}: |
| 1 |
| 2 |
| 3 |
|  |
| julia> myVec = [1,2,3] |
| WARNING: redefinition of constant myVec. This may fail, cause incorrect answers, or produce other errors. |
| 3-element Vector{Int64}: |
| 1 |
| 2 |
| 3 |
|  |
| julia> myVec = [3,4,5] |
| WARNING: redefinition of constant myVec. This may fail, cause incorrect answers, or produce other errors. |
| 3-element Vector{Int64}: |
| 3 |
| 4 |
| 5 |

* Al intentar reasignar un valor a una constante ya definida, si esto viene con un nuevo tipo de dato se lanzará un error.

|  |
| --- |
| julia> const x = 1.0 #A simple vista el tipo de dato es un flotante |
| 1.0 |
| julia> x = 1 #Aquí se intenta redefinir por un valor de tipo Entero |
| ERROR: invalid redefinition of constant x |

* Caso contrario a la situación anterior, si se reasigna u valor a una constante definida pero esta vez coinciden los tipos de datos, se generará solo una advertencia.

|  |
| --- |
| julia> const y = 1.0 |
| 1.0 |
|  |
| julia> y = 2.0 |
| WARNING: redefinition of constant y. This may fail, cause incorrect answers, or produce other errors. |
| 2.0 |

**Nota:** Estructuras de datos como arreglos y relacionados serán trabajados en el capítulo 6.

## Nombramiento de variables

# Estructuras de control

Fuente: [Control Flow · The Julia Language](https://docs.julialang.org/en/v1/manual/control-flow/)

Herramientas que permiten modificar el flujo de la información ya sea por medio de pausas para realizar operaciones, como determinar condiciones, entre otras. Julia proporciona las siguientes estructuras de control:

* **Expresiones compuestas** con el uso de begin y ‘;’.
* **Condiciones**: if-elseif-else y el operador ternario ‘?:’.
* **Iteraciones-Ciclos**: while y for.
* **Manejo de excepciones**: try-catch, error y throw.
* **Tareas-corutinas**: yieldto.

Estas categorías son el resultado de aplicar los fundamentos de la programación, por lo que se incluyen en la gran mayoría de lenguajes (por no decir no todos) con pequeñas diferencias atribuidas mayormente a sintaxis.

### Expresiones compuestas

Son muy útiles en situaciones donde se quieren realizar varias cosas al tiempo, por ejemplo la siguiente expresión:

|  |
| --- |
| someVarX = 2 |
| someVarY = 10 |
| someExpression = someVarX + someVarY |

En donde se define una operación aritmetica entre dos variables y cuyo resultado se almacena en someExpression, puede ser re-escrita como:

|  |
| --- |
| someExpression = begin |
| someVarX = 2 |
| someVarY = 10 |
| someVarX + someVarY |
| end |

O también como:

|  |
| --- |
| someExpression = (someVarX = 2; someVarY = 10; someVarX + someVarY) |

Ambas maneras generan el mismo resultado:

|  |
| --- |
| > 12 |

La diferencia en cuanto a escritura permite en algunas ocasiones tener código más compacto (generalmente cuando se tienen muchas líneas de código), incluso para ayudar a determinar de manera más rápida los componentes de alguna expresión matemática. Por otra parte, son muy útiles con el aspecto de las funciones las cuales se desarrollarán en capítulos siguientes.

### Condicionales

Permiten la ejecución de un conjunto de instrucciones dado el resultado de evaluar una condición, es decir, si algo es verdad se ejecuta una acción determinada, si algo es falso se ejecuta otra acción determinada.

Julia representa las condiciones se la siguiente manera:

|  |
| --- |
| using Printf |
|  |
| myNumber = rand(-10:10) |
|  |
| if myNumber > 0 |
| @printf("%d es MAYOR a 0", myNumber) |
| Else |
| @printf("%d es MENOR a 0", myNumber) |
| End |

La variable myNumber almacena un numero aleatorio entre -10 y 10 generado por rand(). Dependiendo del valor asignado, si este es mayor a cero (determinado por myNumber > 0) se ejecuta el código siguiente el cual muestra en pantalla un mensaje indicando que el número es mayor a cero. En caso contrario se ejecuta el código siguiente al else el cual también muestra en pantalla un mensaje, pero esta vez indicando lo contrario al mensaje anterior, que el número es menor a cero.

En esta ocasión solo se analizó una sola condición, es decir, que el numero generado fuese mayor a cero. Puede haber situaciones en donde se requiera más de una condición, y por ello se podría pensar que entonces se requieren múltiples condiciones aisladas. Es aquí donde el uso de los elseif:

|  |
| --- |
| … |
|  |
| if myNumber == 5 |
| @printf("%d es IGUAL a 5", myNumber) |
| elseif myNumber == 6 |
| @printf("%d es IGUAL a 6", myNumber) |
| else |
| @printf("%d es MENOR a 5 y a 6 ", myNumber) |
| End |

Aquí se abarcan dos condiciones en una sola ejecución. Primero, si myNumber es igual a 5, se muestra el mensaje en pantalla indicando lo anterior, lo mismo sucede para la siguiente condición en caso tal de que la actual sea determine falsa. Cuando ninguna de las condiciones evaluadas se determine verdadera, se ejecutan las instrucciones siguientes al else.

Partiendo de todo lo mencionado hasta el momento, podemos resumir la estructura de los condicionales de la siguiente manera:

|  |
| --- |
| If <condición> |
| <código que se ejecuta si la condición es verdadera> |
| elseif <condición> # condición que se evalúa si la anterior es falsa |
| <código que se ejecuta si la condición elseif es verdadera> |
| {…} # Se continúa igual al bloque elseif anterior con tantas condiciones como se requieran |
| Else |
| <código que se ejecuta cuando ninguna de las condiciones |
| evaluadas con anterioridad es verdadera, |
| es decir, esto es cuando la condición es falsa> |

Ahora bien, algunas características a tener presente en el uso de condicionales son:

* Recordando el concepto de *alcance de las variables*, las estructuras de condición if permiten el uso externo de variables definidas dentro de esta, es decir:

|  |
| --- |
| if 5 > 0 |
| result = true |
| else |
| result = false |
| end |
|  |
| println(result) |

El código anterior muestra la evaluación de una condición y en donde por medio de una variable result, y estando ya fuera del código definido en la estructura if se logra mostrar un mensaje en pantalla.

Lo particular aquí es la manera en cómo se utiliza la variable result ya que en el ejemplo anterior se hizo algo similar con la diferencia que allí se define la variable antes de entrar a definir la condición.

Este comportamiento permite más flexibilidad al momento de escribir de determinar la lógica necesaria para resolver algún problema, sin embargo, se debe de tener en cuenta que para el correcto uso de esta característica se debe asegurar que la variable definida dentro de la estructura condicional obtenga todos sus posibles valores en sus posibles caminos, de otra manera resultará en un error de variable con valor indefinido o UndefVarError.

* Citando a la documentación oficial “A diferencia de C, MATLAB, Perl, Python y Ruby, pero al igual que Java y algunos otros lenguajes tipados más estrictos, es un error si el valor de una expresión condicional es cualquier cosa menos verdadero o falso.”:

|  |
| --- |
| julia> if 1 |
| println("true") |
| end |
| ERROR: TypeError: non-boolean (Int64) used in boolean context |

Ejecutando un código similar en Python, se obtiene lo siguiente:

|  |
| --- |
| >>> if 1: |
| ... print("true") |
| ... |
| True |

Lo que se indica es que en Julia solo se permiten expresiones que puedan retornar valores booleanos, o simplemente valores booleanos. Como se obtiene en código de Python, este puede interpretar un uno como un valor booleano, de la misma manera puede hacerse con la existencia de datos como los objetos. Esto a diferencia de Julia, no es posible.

* Si se desea representar un condición sencilla sin la necesidad de usar varias líneas, se puede hacer uso del operador ternario ‘?:’ de la siguiente manera:

|  |
| --- |
| julia> "A" > "B" ? true : false |
| False |
|  |
| julia> 4%2==0 ? true : false |
| True |

De forma general:

|  |
| --- |
| Condición ? <si es verdad> : <si es falsa> |

Deben respetarse los espacios entre los dos puntos y el signo de interrogación, de lo contrario resultará en error.

### Iteraciones-ciclos

Componen un conjunto de operaciones que se ejecutan un determinado número de veces en función de una condición. Existen dos estructuras que son el for y el while, una se diferencia de otra en el sentido de que la primera está diseñada para procesos en donde se conoce el número total de iteraciones, como ocurre cuando se recorre un rango de numeros o un conjunto de elementos. Por su parte, la segunda estructura se enfoca generalmente en procesos donde no se tiene claro cuantas veces se itera. Sin embargo, ambas pueden ser usadas para describir una misma situación.

Julia define ambas estructuras de la siguiente manera:

* Para la estructura for:

|  |
| --- |
| for i = 1:10 |
| print(i, " ") |
| End |
|  |
| > 1 2 3 4 5 6 7 8 9 10 |

Luego de la palabra clave for, se indica cual será el número de iteraciones a realizar y la variable que almacena el conteo de estas (i = 1:10). Durante cada nueva iteración se evalúa si el número de iteraciones se encuentra dentro del rango definido que para el caso de ejemplo sería 1:10. Una vez alcanzado, se detiene la ejecución de las instrucciones siguientes.

Existen otras maneras de definir el rango de iteraciones:

|  |
| --- |
| julia> for i in [1,4,0] |
| println(i) |
| end |
| 1 |
| 4 |
| 0 |
|  |
| julia> for s ∈ ["foo","bar","baz"] |
| println(s) |
| end |
| foo |
| bar |
| baz |

Como se mencionó anteriormente, las estructuras de iteración permiten iterar sobre un rango bien sea de numeros o de un conjunto de elementos. Los ejemplos anteriores están usando estructuras for para iterar sobre lo que se conocerá en el capítulo 6 como arreglos.

En estos casos, el rango de iteración será el total de elementos presentes en el arreglo. La acción de la variable i y s será la de almacenar el elemento presente en una iteración determinada, es decir, si existen 3 elementos en el arreglo se determinan entonces 3 iteraciones totales en donde en cada una se pasará sobre cada elemento utilizando las variables mencionadas. Es por ello por lo que al imprimir en pantalla el contenido de estas encuentra el contenido total de cada arreglo.

Nótese que, en lugar de usar una definición de variable y dos puntos para designar un rango de iteraciones, se usó la palabra clave in y el carácter especial . El primero y traducido del inglés *dentro* permite asignar a una variable un elemento que se encuentre dentro de un conjunto. Por su parte, el carácter especial siendo mayormente usado en matemáticas describe todo aquello que pertenece a un conjunto.

Hay una característica importante de resaltar en las estructuras for y es que se permiten múltiples iteraciones, es decir:

|  |
| --- |
| julia> for i = 1:2, j = 3:4 |
| println((i, j)) |
| end |
| (1, 3) |
| (1, 4) |
| (2, 3) |
| (2, 4) |

Lo que es lo mismo que hacer:

|  |
| --- |
| julia> for i=1:2 |
| for j=3:4 |
| println((i,j)) |
| End |
| End |
| (1, 3) |
| (1, 4) |
| (2, 3) |
| (2, 4) |

* Para la estructura while:

|  |
| --- |
| i = 0 |
|  |
| while i < 5 |
| print(i, " ") |
| global i += 1 |
| End |

Toda la estructura se podría leer como “mientras la variable i sea menor a 5, se ejecutan las instrucciones de print(…) y global i += 1”. Cada vez que la condición que va después de la palabra clave while se determina como verdadera, se ejecutan las instrucciones que se ubican después de ella. Este funcionamiento es similar al for analizado con anterioridad, con la diferencia de que se debe indicar por separado el incremento de cada iteración con su respectiva variable i.

El uso de global es gracias a los conceptos vistos en la sección 3.2 de este módulo, en donde, por razones de especificación se debe hacer indicación sobre que variable se está utilizando, que para el ejemplo sería una referencia a la definición de la variable i antes de la estructura while.

#### **Uso de** break **y** continue

Puede darse el caso en donde se necesite intervenir el proceso de ejecución ya sea para parar por completo o adelantar. Es por ello que break es la palabra reservada que ayuda en la primera situación y continue es la palabra reservada para la segunda. Por ejemplo:

|  |
| --- |
| for i = 1:typemax(Int64) |
| if i == 100 |
| break |
| elseif i % 2 == 0 |
| continue |
| else |
| print(i, " ") |
| end |
| End |

### El resultado del código anterior es:

|  |
| --- |
| > 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 |

Gracias a la condición i == 100 se detecta el momento en que alcanzado el valor 100, y es allí en donde break detiene la ejecución del código. Por otra parte, el uso del continuo junto a la condición i % 2 == 0 permiten imprimir únicamente aquellos numeros de todo el rango posible que sean impares. Cada vez que se encuentran con un número que cumple la condición, la palabra continue hace que se salte a la próxima iteración, saltándose por consiguiente la impresión del número par.

### Manejo de errores y excepciones

Para esta sección, Julia provee de un gran número de herramientas, sin embargo solo se mencionarán algunas de ellas. Estas permitirán tanto especificar errores (conocidos también como excepciones) que se puedan presentar durante la ejecución del código como manipularlos mediante su almacenamiento en variables.

**Uso de** throw

Permite lanzar errores específicos además de dar un mensaje descriptivo. Julia por default maneja un gran conjunto de estos, algunas de ellos van desde errores de división, índices, poca memoria, variables no definidas, hasta errores en llamados o composición de otros elementos.

**Nota:** Para consultar más sobre las excepciones estándar en Julia, visitar [Excepciones en Julia](https://docs.julialang.org/en/v1/manual/control-flow/#Exception-Handling)

|  |
| --- |
| julia> throw(DomainError(1, "Aqui iria un mensaje descriptivo sobre la excepcion lanzada")) |
| ERROR: DomainError with 1: |
| Aqui iria un mensaje descriptivo sobre la excepcion lanzada |
| Stacktrace: |
| [1] top-level scope |
| @ REPL[5]:1 |

La función throw(…) con el uso de las excepciones en forma de funciones como se muestra en el ejemplo permite retornar un error descrito según el mensaje indicado.

**Uso de** error

Esta palabra clave presenta una excepción como un error general, es decir, sin hacer especificaciones sobre si el error es relacionado a un número, a sintaxis, estructura de sintaxis, entre otros.

|  |
| --- |
| julia> raiz(x) = x >= 0 ? sqrt(x) : error("Valores negativos no admitidos") |
| raiz (generic function with 1 method) |
|  |
| julia> raiz(-1) |
| ERROR: Valores negativos no admitidos |
| Stacktrace: |
| [1] error(s::String) |
| @ Base ./error.jl:35 |
| [2] raiz(x::Int64) |
| @ Main ./REPL[10]:1 |
| [3] top-level scope |
| @ REPL[11]:1 |

El código presenta la creación de una función raíz la cual retornará la raíz de un elemento x. Por medio de una expresión ternaria determina si el valor de x (el valor pasado a la función) es positivo o no. En caso de que no lo sea, se utiliza la función error() con un mensaje descriptivo sobre el valor correcto de x. Igual a como ocurre con throw(), error() también detiene la ejecución y muestra un error de manera general, es decir, sin especificar la tipología de la excepción.

**Uso de** try/catch

Como su nombre lo indica, *intenta* y *atrapa* excepciones:

|  |
| --- |
| julia> try |
| sqrt("ten") |
| catch e |
| println("You should have entered a numeric value") |
| end |
| You should have entered a numeric value |

En el código se intenta sacar la raíz cuadrada de un dato de tipo string. Al no estar permitido se debería de lanzar un error. En este caso el error después de haberse ejecutado la operación con try es atrapado por catch y posteriormente almacenado en una variable e, posteriormente se detiene la ejecución y en lugar de mostrar un error tan específico como en casos anteriores, se muestra el mensaje indicado después del catch.

Esta estructura trae consigo un conjunto de beneficios como el hecho de que permite manipular el error por medio de una variable, suavizar el mensaje de error, y proporcionar de lógica en cualquiera de las dos fases (antes de un error y después de un error).

# Funciones

Fuente: [Functions · The Julia Language](https://docs.julialang.org/en/v1/manual/functions/) y [Methods · The Julia Language](https://docs.julialang.org/en/v1/manual/methods/)

Son segmentos de código encargados a realizar una tarea o **función**. Son una parte fundamental de la programación ya que permiten una codificación modularizada (es decir, por módulos o partes destinadas a una sola tarea), mantienen una eficiencia tiempo-espacio, proveen de mayor calidad estética al código, entre muchos otros beneficios.

Durante el desarrollo de las secciones anteriores se ha podido observar el uso de alguna función, siendo este la delegación de una actividad específica para llevar a cabo una actividad mayor.

## Estructura y caracteristicas

La forma general que sigue la definición de una función es:

|  |
| --- |
| function <nombre de la función>(argumentos) |
| <conjunto de instrucciones> |
| end |

Se pudo observar en ejemplos anteriores que el uso de un función se hace por medio de un llamado, es decir, escribir el nombre de la función en el lugar donde se requiere, esto también viene acompañado de la información que esta pueda necesitar, es decir, sus argumentos. Por ejemplo:

|  |
| --- |
| function myFirstFunction() |
| print("Hello world!") |
| End |
|  |
| myFirstFunction() |

En este caso, la función descrita imprime en pantalla el mensaje “Hello world!”. Esta es solicitada mediante el llamado myFirstFunction().

|  |
| --- |
| function myFirstFunction(name) |
| print("Hello $name") |
| end |
|  |
| myFirstFunction("Juan") |

La función anterior utiliza información adicional externa almacenándola en una variable name, que luego servirá para imprimir en pantalla un mensaje personalizado. Para el ejemplo en cuestión se da como resultado:

|  |
| --- |
| > Hello Juan |

Existe otra manera muchos rápida y corta de definir las funciones:

|  |
| --- |
| myFirstFunction(name) = print("Hello $name") |
| myFirstFunction("Juan") |

Y retorna como resultado:

|  |
| --- |
| > Hello Juan |

Similar a como definir una variable, este método suele ser muy útil para cuando se necesitan realizar tareas muy cortas y de manera repetida reduciendo el “ruido” visual del resto de código. Aquí se vuelven muy útiles las expresiones compuestas, por ejemplo:

|  |  |
| --- | --- |
| Usando begin-end | Usando ‘;’ y parentesis |
| mySecondFunction(x, y) = begin  print(x + y)  end  mySecondFunction(1, 2) | mySecondFunction(x, y) = (print(x + y));  mySecondFunction(1, 2) |

Ambas formas dan como resultado >3.

## Vinculación entre la información y las funciones

En el ámbito de la funciones existen dos conceptos mediante los cuales la información se puede vincular, estos son:

* **Argumentos:** Son los valores con los cuales se desarrollan las instrucciones almacenadas en la función. Estos valores son asignados en el momento de llamar a una función, por ejemplo, la variable name del ejemplo anterior.
* **Parámetro(función):** Indican el tipo de datos que se pueden usar en el algoritmo que desarrolla la función en su cuerpo de instrucciones, es decir, las variables y demás elementos que se crean dentro de la función.

## Retorno de información

Con todo lo visto, se puede entender que las funciones permiten dividir el trabajo en pequeñas partes (modularizar). Esta característica es muy útil en el desarrollo de algoritmos, un conjunto de pasos ordenados, concretos y finitos que buscan solucionar un problema.

Muchas veces en los algoritmos se requiere no solamente que una función ejecute ciertas instrucciones, también se necesite que retornen información al punto en donde han sido llamadas. Por ejemplo, una situación particular sería que en lugar de mostrar en pantalla el resultado de una suma entre dos numeros dados, s retorne ese número para poder manipularlo. Es entonces que todo esto es posible gracias a la palabra clave return:

|  |
| --- |
| function mySumOperation(a, b) |
| return a + b |
| end |
|  |
| result = mySumOperation(5, 10) |
|  |
| print("El resultado de la suma entre 5 y 10 es $result") |

En general, después de indicar la palabra return se escribe la información a retornar, bien sea una operación corta como se muestra en el ejemplo, lo que evita líneas de código, o bien sea una variable o tipo de dato.

El funcionamiento del proceso de retorno a nivel interno se compone de evaluar la última expresión definida dentro de la función hasta que se encuentre el final de esta o la palabra reservada return, por ende, casos como el siguiente:

|  |
| --- |
| function mySumOperation(a, b) |
| return a + b |
| print("Despues del return") |
| End |

solo resultarán efectivos hasta la línea de retorno.

## Retorno de múltiples valores y asignación por destructuración

Continuando con la idea de retornar información, puede darse la situación en que una función retorne más de un dato:

|  |
| --- |
| function someOperations(a, b) |
| return a + b, a - b, a \* b, a // b |
| End |

Para recibir la información se hace lo que se conoce como *asignación por* *deconstrucción*:

|  |
| --- |
| suma, resta, multiplicacion, division = someOperations(4, 2) |

Imprimiendo el valor de cada variable se tiene que:

|  |
| --- |
| print("suma:$suma\nresta:$resta\nmultiplicacion:$multiplicacion\ndivision:$division\n") |
| > suma:6 |
| resta:2 |
| multiplicacion:8 |
| division:2//1 |

Cabe resaltar que de esta manera se debe de conocer el orden en el que llegan los datos, de lo contrario se harán asignaciones a variables erróneas.

## Funciones de argumentos múltiples, opcionales y keywords

**Argumentos múltiples**

Caso contrario a como sucede con las funciones de múltiples retornos, puede darse la situación de tener una función con múltiples argumentos. En estos casos se hace lo siguiente:

|  |
| --- |
| function multipleArgs(a, b, x...) |
| return (a, b, x) |
| end |
|  |
| println(multipleArgs(1, 2, 3)) |
| println(multipleArgs(1, 2, 3, 4)) |
| println(multipleArgs(1, 2, 3, 4, 5)) |
| println(multipleArgs(1, 2, 3, 4, 5, 6)) |

Como resultado de los llamados se obtiene:

|  |
| --- |
| > (1, 2, (3,)) |
| (1, 2, (3, 4)) |
| (1, 2, (3, 4, 5)) |
| (1, 2, (3, 4, 5, 6)) |

De lo anterior se debe resaltar que:

1. Se introducen los tres puntos suspensivos que permiten indicar múltiples valores ‘…’. Se observa que estos están antecedidos por una variable x. Esta es la encargada de almacenar todos los valores siguientes a los que ya se definieron en los argumentos anteriores.
2. La función retorna los elementos proporcionados como argumentos en forma de tupla. De manera general son estructura de datos (ver capitulo 6) que permiten almacenar información de uno o más tipos, y contienen características particulares como que son objetos que no admiten cambios en la información que almacenan y por ende solo permiten indexación como sucede con las strings, adición y eliminación. Estos elementos proporcionan una buena manera de empaquetar los datos por lo que son muy útiles en los aspectos de retorno, como en la definición de variables en donde también es posible hacer *asignación por destructuración*.

## Funciones anónimas y bloques Do

## Operadores como funciones

## Composición y canalización entre funciones

# Estructuras de datos

Elementos que permiten unir y manipular datos de uno o distintos tipos con el objetivo de proporcionar un uso más eficiente y comprensible de la información. Suelen estar presentes en todo el campo de la programación en situaciones que involucran manejo de grandes volúmenes de información ayudando en los procesos del desarrollo del software como las bases de datos o cálculos de gran magnitud en proyectos de ingeniería; También son una parte intrínseca en la implementación de algoritmos permitiendo estandarizar la solución de un problema en un conjunto de pasos con un conjunto de herramientas consistente.

5.1.1 Clasificación

La variedad de estructuras se clasifica con base en el tamaño y procesamiento. De lo anterior, se tienen lo siguiente:

5.1.1.1 Tamaño (por organizar-reescribir)

● Estático: Estructura de datos que mantiene un tamaño de almacenamiento fijo en ejecución, es decir, solo permite una cantidad de datos sin posibilidad de nuevos ingresos. Al operar con estructura de este tipo no se podrá cambiar el valor de su tamaño a menos que se utilice una nueva estructura y se re-posicionen los datos almacenados en la estructura inicial.

Ejemplos: arreglos (vectores, matrices).

● Dinámico: Estructura de datos que permite cambiar su tamaño de almacenamiento en ejecución. Al operar con estas estructuras, en medio de la ejecución pueden almacenar o eliminar información configurando a la vez el tamaño que las describe.

Ejemplos: Listas, pilas, colas, árboles, grafos, deque.

5.1.1.2 Procesamiento

● Lineal: Los elementos dentro de estas estructuras se organizan uno tras otro.

Ejemplos: arreglos (vectores, matrices), listas, pilas, colas, deque.

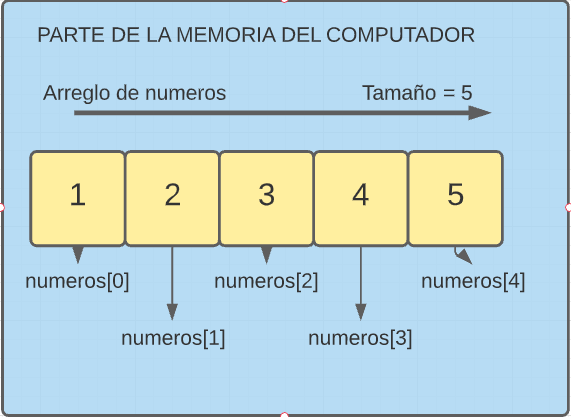
● No lineal: En comparación con el caso anterior, los datos en estructuras no lineales se organizan de forma aleatoria.

Ejemplos: Árboles, grafos.

## Arreglos

Fuente que puede dar información útil sobre arreglos: [Multi-dimensional Arrays · The Julia Language](https://docs.julialang.org/en/v1/manual/arrays/)

Son estructuras de datos que permiten almacenar información de un mismo tipo dentro de un espacio de memoria fijo durante la ejecución, es decir, su tamaño no podrá cambiar a menos que se cree otra nueva estructura. De manera gráfica se puede entender de la siguiente manera:



## Constructores

## Otras estructuras presentes en Julia

Fuente: [Collections and Data Structures · The Julia Language](https://docs.julialang.org/en/v1/base/collections/)

# Interfaces

Fuente: [Interfaces · The Julia Language](https://docs.julialang.org/en/v1/manual/interfaces/)

# Essentials

Fuente: [Essentials · The Julia Language](https://docs.julialang.org/en/v1/base/base/#baremodule)

1. Visitar [Integers and Floating-Point Numbers · The Julia Language](https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/) en el apartado de *special floating points.* [↑](#footnote-ref-1)