Julia como lenguaje de programación para un curso de algoritmos

Capítulo 1 — Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

1. Julia como lenguaje de programación para un curso de algoritmos

Nuestro objetivo es trabajar sobre algoritmos, por lo que cualquier lenguaje que pueda expresar todo lo computable, puede ser adecuado. Pero dado que nuestro enfoque será experimental, y nuestra metodología incluye medir la factibilidad y desempeño de cada algoritmo en términos reales, entonces necesitamos un lenguaje donde las instrucciones, el acceso a memoria, y la manipulación de la misma sea controlable. En este caso, y mediando con la facilidad de aprendizaje y la productividad, este curso utiliza el lenguaje de programación Julia.[1] Pero no hay por qué preocuparse por aprender un nuevo lenguaje, el curso utiliza ejemplos en Julia y utiliza una variante de su sintaxis como pseudo-código, pero las actividades se esperan tanto en Julia como en Python.

Ambos lenguajes de programación son fáciles de aprender y altamente productivos. Python es un lenguaje excelente para realizar prototipos, o para cuando existen bibliotecas que resuelvan el problema que se esté enfrentando. Por otro lado, cuando se necesita control sobre las operaciones que se están ejecutando, o la memoria que se aloja, Python no es un lenguaje que nos permita trabajar en ese sentido. Julia está diseñado para ser veloz y a la vez mantener el dinamismo que se espera de un lenguaje moderno, adicionalmente, es posible conocer los tipos de instrucciones que realmente se ejecutan, así como también es posible controlar el alojamiento de memoria, ya sea mediante la utilización de patrones que así nos lo permitan, o mediante instrucciones que nos lo aseguren.

Este curso está escrito en Quarto, y se esperan reportes de tareas y actividades tanto en Quarto https://quarto.org como en Jupyter https://jupyter.org/. La mayoría de los ejemplos estarán empotrados en el sitio, y en principio, deberían poder replicarse copiando, pegando, y ejecutando en una terminal de Julia.

Es importante clarificar que este capítulo introducirá el lenguaje de programación Julia hasta el nivel que se requiere en este curso, ignorando una gran cantidad de capacidades que no son de interés para nuestro curso. Se recomienda al alumno interesado la revisión del manual y la documentación oficial para un estudio más profundo del lenguaje.

[1] Se recomienda utilizar la versión 1.10 o superior, y puede obtenerse en https://julialang.org/.

1.1. El lenguaje de programación Julia

El código se suele organizar en scripts, módulos y paquetes. Cada uno de estos define tipos y funciones que interactúan para componer las soluciones deseadas.

El resto de esta unidad está dedicada a precisar la sintaxis del lenguaje y anotaciones de importancia sobre su funcionamiento.

1.2. Instalación

El sitio oficial recomienda el uso de `juliaup`, una herramienta que permite manejar diferentes versiones de Julia y mantenerlas actualizadas.

https://julialang.org/install/

Las versiones de Julia siguen el paradigma de *semantic versioning* (semver), por lo que `juliaup` permite gestionarlas de manera simple y efectiva. La versión estable es la 1.10 y las más nuevas son la 1.11 y la 1.12.

También es posible usar Colab de Google con el kernel para `Julia`; este usa Julia 1.11 y hasta el momento, es el único disponible.

1.3. Manos a la obra

Una vez instalado, se puede ejecutar un REPL de Julia en la terminal ejecutando

```{bash}
$ julia
```

dado que instalamos con `juliaup` podemos mantener diferentes versiones, e.g.,

```{bash}
$ juliaup list
```

que nos mostrará una larga lista de posibles *channels* o versiones de instalación

```{bash}

$ juliaup add 1.10
$ juliaup default 1.10
```

estas instrucciones añadirán la versión 1.10 y la establecerán como versión o canal por omisión. Puedes llamar diferentes versiones ejecutando `julia +channel` como sigue:

```{bash}
$ julia +1.12
```

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.1 (2025-10-17)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

julia>
```

```

#### 1.3.1. Creando el famoso "Hola mundo"

Uno de los programas más comunes es el siguiente

```
println("¡Hola !")
```

```
¡Hola !
```

1.3.2. Colab

Es posible usar Colab para reducir la complejidad de la instalación, ya que cuenta con un kernel de Julia. Como se mencionaba anteriormente, solo se soporta la versión `1.11` que es subóptima con las versiones de paquetes que usaremos más adelante. Adicionalmente, se tiene la limitante de los recursos que se nos proporcionen, en particular al momento de escribir estas notas, aunque los recursos que se otorgan suelen ser suficientes para pruebas, no lo son en otros ámbitos: solo se tienen 2 vcpus y tiempos de ejecución limitados.

1.3.3. Jupyter

Una vez instalado Julia; debemos instalar el paquete `IJulia` que instalará todo lo necesario para correr Jupyter (ver la sección de `Pkg` al final de esta unidad para más información sobre paquetes). Una vez corriendo, se debe seleccionar crear un notebook especificando el kernel de Julia para utilizarlo.
```

1.3.4. Quarto

Quarto es un sistema de publicación científica y técnica de código abierto que permite combinar texto en Markdown, código ejecutable, figuras y ecuaciones en un mismo documento. Es el sucesor espiritual de R Markdown, pero diseñado desde el inicio para ser multi-lenguaje: soporta Julia, Python, R y Observable JS de manera nativa. Este libro está escrito en Quarto, y se espera que los reportes de tareas y actividades del curso también lo estén.

1.3.4.1. Instalación

Quarto se descarga como un ejecutable independiente desde https://quarto.org/docs/get-started/. No depende de ningún lenguaje en particular, por lo que puede instalarse antes o después de Julia. Para verificar la instalación:

```
quarto --version
```

Para usar Julia como motor de ejecución, Quarto se apoya en el paquete `IJulia` (el mismo que usa Jupyter). Basta con tenerlo instalado:

```
using Pkg
Pkg.add("IJulia")
```

1.3.4.2. Estructura de un documento `.qmd`

Un archivo Quarto (extensión `.qmd`) tiene tres partes principales:

1. **Encabezado YAML** — metadatos del documento (título, autor, formato de salida, engine, etc.)
2. **Texto en Markdown** — prosa, ecuaciones LaTeX (`$...$`), listas, tablas, etc.
3. **Bloques de código** — celdas ejecutables delimitadas por ```` ```{julia} ```` que se evalúan al renderizar.

Un ejemplo mínimo de reporte para este curso:

````
---
title: "Tarea 1 - Análisis asintótico"
author: "Tu Nombre"
engine: julia
format: html
---

## Introducción

En esta tarea analizamos la complejidad del algoritmo de búsqueda lineal.

```{julia}
function busqueda_lineal(arr, x)
````
````

```
    for (i, v) in enumerate(arr)
        v == x && return i
    end
    return -1
end
```

1.3.4.3. Renderizado

Para generar el HTML (o PDF) a partir de un archivo `.qmd`:

```
quarto render mi_reporte.qmd          # HTML por defecto
quarto render mi_reporte.qmd --to pdf # PDF con LaTeX
quarto preview mi_reporte.qmd         # servidor local con recarga automática
```

> **Flujo de trabajo recomendado**
>
> Usa `quarto preview` mientras escribes: abre un navegador con recarga automática cada vez que guardas el archivo, lo que agiliza mucho el ciclo de edición-visualización.

> **Freeze y caché**
>
> Quarto puede cachear los resultados de ejecución con la opción `freeze: auto` en el YAML del proyecto. Esto evita re-ejecutar celdas costosas si el código no ha cambiado — algo muy útil para notebooks con benchmarks largos.

1.3.4.4. Publicación en GitHub Pages

Una de las ventajas de Quarto es que permite publicar el reporte como un sitio web con un solo comando:

```
quarto publish gh-pages mi_reporte.qmd
```

Esto crea o actualiza una rama `gh-pages` en tu repositorio de GitHub y despliega el HTML automáticamente. Es la forma en que se publican las actividades de este curso (ver Galería de actividades).

1.3.5. Ejercicios

Según sus posibilidades de equipo:

- Instale Julia 1.10, luego instale el paquete `IJulia` y ejecute Jupyter.
- Cree un notebook con Colab con el kernel de Julia.
- Instale Quarto y cree un documento `.qmd` con `engine: julia` que incluya al menos un bloque de código ejecutable. Renderícelo a HTML con `quarto render`.
- (Opcional) Publique su documento en GitHub Pages con `quarto publish gh-pages`.

1.4. Sintaxis y estructuras

1.4.1. Funciones

Las funciones son centrales en Julia. Por ahora veremos la estructura y más adelante, definiremos algunas.

Para ejecutar una función se utiliza la sintaxis `fun(arg)`, esta regresará un valor, que depende de la función misma y muchas veces del tipo que tenga `arg`. Si fueran dos argumentos `fun(arg1, arg2)`, etc. También se soportan argumentos con nombre `fun(arg1, arg2, ...; kwarg=val)` (*kwargs* para nombrarlos de manera sintética). En este caso, los *kwargs* no influyen en los tipos de salida. Esto puede parecer extraño pero es debido a las decisiones de implementación relacionadas con el desempeño.

Las funciones se definen como sigue:

```
function fun(arg1, arg2...)                                          ①
    # ... expresiones ...
end

function fun(arg1, arg2...; kwarg1=valor1, kwargs2...)               ②
    # ... expresiones ...
end

fun(arg1, arg2...; kwarg1=valor1, kwargs2...) = expresion            ③

(arg1, arg2...; kwarg1=valor1, kwargs2...) -> expresion              ④

fun() do x                                                           ⑤
    x^2 # ... expresiones ...
end
```

① Definición de una función simple, los tipos de los argumentos se utilizan para generar múltiples versiones de una función.

② También se soportan argumentos nombrados, los cuales van después de `;`, se debe tener en cuenta que los tipos de los argumentos nombrados no son utilizados para determinar si una función debe compilarse. Los argumentos nombrados pueden o no tener valores por omisión.

③ Si la función tiene una estructura simple, de una expresión, es posible ignorar `function` y `end`, usando '=' para definirla.

④ Muchas veces es útil definir funciones anónimas, que suelen pasarse a otras funciones de orden superior.

⑤ Un embellecedor útil para generar una función anónima (definida entre `do...end`) que se pasa como primer argumento a `fun`, e.g., es equivalente a `fun(x->x^2)`.

El *ámbito* o *scope* de las variables en Julia es sintáctico, lo que significa que se hereda del código donde las funciones fueron definidas, y no dinámico (que se hereda desde dónde se ejecuta la función). Aunque es el comportamiento de la mayoría de los lenguajes modernos, es importante conocerlo sobre todo para la creación de *cerraduras sintácticas* en funciones.

1.4.2. Expresiones y operadores

Las *expresiones* son la forma más genérica de expresar el código en Julia, comprenden operaciones aritméticas, asignación y declaración de variables, definiciones de bloques de código, llamadas de funciones, entre otras.

Cada línea suele ser una expresión, a menos que se extienda por múltiples líneas por medio de un agrupador de código o datos, estos pueden ser

```
begin
    ...
end

let
    ...
end

(...)

[...]

[...]

for el in col
    ...
end

while cond
    ...
end

if cond
    ...
elseif cond
    ...
else
    ...
end

function fun(...)
    ...
end

try
    ...
catch
    ...
```

```
finally
    ...
end
```

entre las más utilizadas; claramente hay infinidad de formas de componerlas para formar los algoritmos que se estén escribiendo.

1.4.3. Comentarios en el código

Los comentarios en Julia se hacen por línea o por bloque.

Para comentar una línea se usa el carácter hash `#` y define una línea comentada desde ese punto hasta el salto de línea

```
# los siguientes son comentarios de línea completos, por lo que no se imprimirán
#println(:hola === :hola)
#println(typeof(:hola))
println(Symbol("hola mundo"))  # este sí se imprimirá, pero este comentario no
```

```
hola mundo
```

Para comentar por bloque, dicho bloque se encierra entre `#= ... =#`

```
println("hola mundo!" #=Symbol("hola mundo")=#)
```

```
hola mundo!
```

1.4.4. Documentación del código

La documentación oficial se encuentra en https://docs.julialang.org; la cual cubre el lenguaje y las bibliotecas estándar. Fuera de eso, habrá que ver los sitios y documentaciones de cada paquete, que casi siempre están en *github*.

Actualmente los chatbots como ChatGPT y Gemini también pueden ser una buena fuente de información sobre el API y las formas de trabajo. Nota: siempre se debe corroborar la información, ya que suelen alucinar.

La manera *empotrada* de consultar la documentación sobre una función es con el prefijo `?` en el REPL.

1.4.4.1. Ejemplo

```
```{julia}
?println("hola")
```
```

Al documentar nuestro código también lo tendremos accesible por este medio, y esto se puede hacer de la siguiente forma:

```
```{julia}

"""
    fun(...)

Esta función es un ejemplo de documentación
"""
function fun(...)
    ...
end
```
```

1.4.4.2. Definición de variables

Las definiciones de variables tienen la sintaxis `variable = valor`; las variables comúnmente comienzan con una letra o `_`, las letras pueden ser caracteres *unicode*, no deben contener espacios ni puntuaciones como parte del nombre; `valor` es el resultado de evaluar o ejecutar una expresión.

Los operadores más comunes son los aritméticos `+`, `-`, `*`, `/`, `÷`, `%`, `\`, `^`, con precedencia y significado típico. Existen maneras compuestas de modificar una variable anteponiendo el operador aritmético al símbolo de asignación, e.g., `variable += valor`, que se expande a `variable = variable + valor`. Esto implica que `variable` debe estar definida previamente a la ejecución.

Los operadores lógicos también tienen el significado esperado.

operación	descripción
`a && b`	AND lógico
`a || b`	OR lógico
`a  b`	XOR lógico
`!a`	negación lógica
`a < b`	comparación `a` es menor que `b`
`a > b`	comparación `a` es mayor que `b`
`a <= b`	comparación `a` es menor o igual que `b`
`a >= b`	comparación `a` es mayor o igual que `b`
`a == b`	comparación de igualdad
`a === b`	comparación de igualdad (a nivel de tipo/memoria)
`a != b`	comparación de desigualdad
`a !== b`	comparación de desigualdad (a nivel de tipo/memoria)

En particular `&&` y `||` implementan *corto circuito de código*, por lo que pueden usarse para el control de que operaciones se ejecutan. Cuando se compara a nivel de tipo 0 (entero) será diferente de 0.0 (real).

También hay operadores lógicos a nivel de bit, los argumentos son enteros.

operación	descripción
`a & b`	AND a nivel de bits
`a | b`	OR a nivel de bits
`a  b`	XOR a nivel del bits
`~a`	negación lógica a nivel de bits

1.4.4.3. Literales

Los valores literales son valores explícitos que Julia permite para algunos tipos de datos, y que permiten definirlos de manera simple; permitiéndonos escribir datos directamente en el código.

Los números enteros se definen sin punto decimal, es posible usar `_` como separador y dar más claridad al código. Los enteros pueden tener 8, 16, 32, o 64 bits; por omisión, se empaquetan en variables del tipo `Int` (`Int64`). Los valores hexadecimales se interpretan como enteros sin signo, y además se empaquetan al número de bits necesario mínimo para contener. El comportamiento para valores en base 10 es el de hexadecimal es congruente con un lenguaje para programación de sistemas.

```
a = 100
println((a, sizeof(a)))
b = Int8(100)
println((b, sizeof(b)))
c = 30_000_000
println((c, sizeof(c)))
d = 0xffff
println((d, sizeof(d)))
```

```
(100, 8)
(100, 1)
(30000000, 8)
(0xffff, 2)
```

[2]

[2]Existen números enteros de precisión 128 pero las operaciones al día de hoy no son implementadas de manera nativa por los procesadores; así mismo se reconocen números de punto flotante de precisión media `Float16` pero la mayoría de los procesadores no tienen soporte nativo para realizar operaciones con ellos, aunque los procesadores de última generación sí lo tienen.

Si la precisión está en duda o el contexto lo amerita, deberá especificarlo usando el constructor del tipo e.g., `Int8(100)`, `UInt8(100)`, `Int16(100)`, `UInt16(100)`, `Int32(100)`, `UInt32(100)`, `Int64(100)`, `UInt64(100)`.

Los números de punto flotante tienen diferentes formas de definirse, teniendo diferentes efectos. Para números de precisión simple, 32 bits, se definen con el sufijo `f0` como `3f0`. El sufijo `e0` también se puede usar para definir precisión doble (64 bit). El cero del sufijo en realidad tiene el objetivo de colocar el punto decimal, en notación de ingeniería, e.g., 0.003 se define como $3f-3$ o $3e-3$, dependiendo del tipo de dato que se necesite. Si se omite sufijo y se pone solo punto decimal entonces se interpretará como precisión doble. Los tipos son `Float32` y `Float64`.

Los datos booleanos se indican mediante `true` y `false` para verdadero y falso, respectivamente.

Los caracteres son símbolos para indicar cadenas, se suelen representar como enteros pequeños en memoria. Se especifican con comillas simples `'a'`, `'z'`, `'!'` y soporta símbolos *unicode* `' '`.

Las cadenas de caracteres son la manera de representar textos como datos, se guardan en zonas contiguas de memoria. Se especifican con comillas dobles y también soportan símbolos unicode, e.g., `"hola mundo"`, `"pato es un  "`.

3

En Julia existe la noción de símbolo, que es una cadena que además solo existe en una posición en memoria se usa el prefijo `:` para denotarlos.

```
println(:hola === :hola)
println(typeof(:hola))
println(Symbol("hola mundo"))
```

```
true
Symbol
hola mundo
```

1.4.5. Control de flujo

El control de flujo nos permite escoger que partes del código se ejecutaran como consecuencia de la evaluación de una expresión, esto incluye repeticiones.

Las condicionales son el control de flujo más simple.

[3] Julia guarda los símbolos de manera especial y pueden ser utilizados para realizar identificación de datos eficiente, sin embargo, no es buena idea saturar el sistema de manejo de símbolos por ejemplo para crear un vocabulario ya que no liberará la memoria después de definirlos ya que es un mecánismo diseñado para la representación de los programas, pero lo suficientemente robusto y bien definido para usarse en el diseño e implementación de programas de los usuarios.

```
a = 10
if a % 2 == 0                                                    ①
    "par"                                                        ②
else
    "impar"                                                      ③
end
```

① Expresión condicional.
② Expresión a ejecutarse si (1) es verdadero.
③ Expresión a evaluarse si (1) es falso.

```
"par"
```

Se puede ignorar la clausula `else` dando solo la opción de evaluar (2) si (1) es verdadero. Finalmente, note que la condicional es una expresión y devuelve un valor.

```
a = 10
if log10(a) == 1                                                 ①
    "es 10"                                                      ②
end
```

```
"es 10"
```

También pueden concatenarse múltiples expresiones condicionales con `elseif` como se muestra a continuación.

```
a = 9
if a % 2 == 0
    println("divisible entre 2")
elseif a % 3 == 0
    println("divisible entre 3")
else
    println("no divisible entre 2 y 3")
end
```

```
divisible entre 3
```

Es común utilizar la sintaxis en Julia (short circuit) para control de flujo:

```
a = 9

println(a % 2 == 0 && "es divisible entre dos")                  ①
println(a % 3 == 0 && "es divisible entre tres")                 ②
```

① El resultado de la condición es falso, por lo que no se ejecutará la siguiente expresión.

② El resultado es verdadero, por lo que se ejecutará la segunda expresión.

```
false
es divisible entre tres
```

Fnalmente, existe una condicional de tres vias `expresion ? expr-verdadero : expr-falso`

```
a = 9

println(a % 2 == 0 ? "es divisible entre dos" : "no es divisible entre dos")
println(a % 3 == 0 ? "es divisible entre tres" : "no es divisible entre tres")
```

```
no es divisible entre dos
es divisible entre tres
```

1.4.5.1. Ciclos

Los ciclos son expresiones de control de flujo que nos permiten iterar sobre una colección o repetir un código hasta que se cumpla alguna condición. En Julia existen dos expresiones de ciclos:

- `for x in colección ...expresiones... end` y
- `while condición ...expresiones... end`

En el caso de `for`, la idea es iterar sobre una colección, esta colección puede ser un rango, i.e., `inicio:fin`, `inicio:paso:fin`, o una colección como las tuplas, los arreglos, o cualquiera que cumpla con la interfaz de colección iterable del lenguaje.

```
for i in 1:5
    println("1er ciclo: ", i => i^2)
end

for i in [10, 20, 30, 40, 50]
    println("2do ciclo: ", i => i/10)
end
```

```
1er ciclo: 1 => 1
1er ciclo: 2 => 4
1er ciclo: 3 => 9
1er ciclo: 4 => 16
1er ciclo: 5 => 25
2do ciclo: 10 => 1.0
2do ciclo: 20 => 2.0
2do ciclo: 30 => 3.0
2do ciclo: 40 => 4.0
2do ciclo: 50 => 5.0
```

Al igual que en otros lenguajes modernos, se define la variante completa o *comprehensive for* que se utiliza para transformar la colección de entrada en otra colección cuya sintaxis se ejemplifica a continuación:

```
a = [i => i^2 for i in 1:5]
println(a)
```

```
[1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25]
```

También es posible definir un generador, esto es, un código que puede generar los datos, pero que no los generará hasta que se les solicite.

```
a = (i => i^2 for i in 1:5)
println(a)
println(collect(a))
```

```
Base.Generator{UnitRange{Int64}, var"#3#4"}(var"#3#4"(), 1:5)
[1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25]
```

Otra forma de hacer ciclos de intrucciones es repetir mientras se cumpla una condición:

```
i = 0
while i < 5
    i += 1
    println(i)
end
```

```
i
```

```
1
2
3
4
5
```

```
5
```

1.4.6. Tuplas y arreglos en Julia

Una tupla es un conjunto ordenado de datos que no se puede modificar y que se desea esten contiguos en memoria, la sintaxis en memoria es como sigue:

```
a = (2, 3, 5, 7)                                                    ①
b = (10, 20.0, 30f0)
c = 100 => 200
println(typeof(a))                                                  ②
println(typeof(b))
println(typeof(c))
a[1], a[end], b[3], c.first, c.second                               ③
```

① Define las tuplas.
② Imprime los tipos de las tuplas.
③ Muestra cómo se accede a los elementos de las tuplas. Julia indexa comenzando desde 1, y el término `end` también se utiliza para indicar el último elemento en una colección ordenada.

```
NTuple{4, Int64}
Tuple{Int64, Float64, Float32}
Pair{Int64, Int64}


(2, 7, 30.0f0, 100, 200)
```

La misma sintaxis puede generar diferentes tipos de tuplas. En el caso `NTuple{4, Int4}` nos indica que el tipo maneja cuatro elementos de enteros de 64 bits, los argumentos entre `{}` son parametros que especifican los tipos en cuestión. En el caso de `Tuple` se pueden tener diferentes tipos de elementos. La tupla `Pair` es especial ya que solo puede contener dos elementos y es básicamente para *embellecer* o *simplificar* las expresiones; incluso se crea con la sintaxis `key => value` y sus elementos pueden accederse mediante dos campos nombrados.

Los *arreglos* son datos del mismo tipo contiguos en memoria, a diferencia de las tuplas, los elementos se pueden modificar, incluso pueden crecer o reducirse. Esto puede implicar que se alojan en zonas de memoria diferente (las tuplas se colocan en el *stack* y los arreglos en el *heap*, ver la siguiente unidad para más información). Desde un alto nivel, los arreglos en Julia suelen estar asociados con vectores, matrices y tensores, y un arsenal de funciones relacionadas se encuentran definidas en el paquete `LinearAlgebra`, lo cual está más allá del alcance de este curso.

```
a = [2, 3, 5, 7]                                                    ①
b = [10, 20.0, 30f0]
println(typeof(a))                                                  ②
println(typeof(b))
a[1], a[end], b[3], b[2:3]                                          ③
```

① Define los arreglos `a` y `b`.
② Muestra los tipos de los arreglos, note cómo los tipos se promueven al tipo más genérico que contiene la definición de los datos.

③ El acceso es muy similar a las tuplas para arreglos unidimensionales, note que es posible acceder rangos de elementos con la sintaxis `ini:fin`.

```
Vector{Int64}
Vector{Float64}
```

```
(2, 7, 30.0, [20.0, 30.0])
```

```
a = [2 3;
     5 7]                                                    ①
display(a)                                                   ②
display(a[:, 1])                                             ③
display(a[1, :])                                             ④
```

① Definición de un arreglo bidimensional, note cómo se ignora la coma `,` en favor de la escritura por filas separadas por `;`.
② La variable `a` es una matriz de 2x2.
③ Es posible acceder una columna completa usando el símbolo `:` para indicar todos los elementos.
④ De igual forma, es posible acceder una fila completa.

```
2×2 Matrix{Int64}:
 2  3
 5  7

2-element Vector{Int64}:
 2
 5

2-element Vector{Int64}:
 2
 3
```

1.4.7. Diccionarios y conjuntos en Julia

Un diccionario es un arreglo asociativo, i.e., guarda pares llave-valor. Permite acceder de manera eficiciente al valor por medio de la llave, así como también verificar si hay una entrada dentro del diccionario con una llave dada. La sintaxis es como sigue:

```
a = Dict(:a => 1, :b => 2, :c => 3)                          ①
a[:b] = 20                                                   ②
println(a)
a[:d] = 4                                                    ③
println(a)
delete!(a, :a)                                               ④
a
```

① Definición del diccionario `a` que mapea símbolos a enteros.
② Cambia el valor de `:b` por 20.
③ Añade `:d => 4` al diccionario `a`.
④ Borra el par con llave `:a`.

```
Dict(:a => 1, :b => 20, :c => 3)
Dict(:a => 1, :b => 20, :d => 4, :c => 3)
```

```
Dict{Symbol, Int64} with 3 entries:
  :b => 20
  :d => 4
  :c => 3
```

Es posible utilizar diferentes tipos siempre y cuando el tipo en cuestión defina de manera correcta la función `hash` sobre la llave y la verificación de igualdad `==`.

Un conjunto se representa con el tipo `Set`, se implementa de manera muy similar al diccionario pero solo necesita el elemento (e.g., la llave). Como conjunto implementa las operaciones clasificación de operaciones de conjuntos

```
a = Set([10, 20, 30, 40])                    ①
println(20 in a)                             ②
push!(a, 50)                                 ③
println(a)
delete!(a, 10)                               ④
println(a)
println(intersect(a, [20, 35]))              ⑤
union!(a, [100, 200])                        ⑥
println(a)
```

① Definición del conjunto de números enteros.
② Verificación de membresia al conjunto `a`.
③ Añade 50 al conjunto.
④ Se borra el elemento 10 del conjunto.
⑤ Intersección de `a` con una colección, no se modifica el conjunto `a`.
⑥ Unión con otra colección, se modifica `a`.

```
true
Set([50, 20, 10, 30, 40])
Set([50, 20, 30, 40])
Set([20])
Set([50, 200, 20, 30, 40, 100])
```

1.5. El flujo de compilación de Julia

Basta con escribir una línea de código en el REPL de Julia y esta se compilará y ejecutará en el contexto actual, usando el ámbito de variables. Esto es conveniente para comenzar a trabajar, sin embargo, es importante conocer el flujo de compilación para tenerlo en cuenta mientras se codifica, y así generar código eficiente. En particular, la creación de funciones y evitar la *inestabilidad* de los tipos de las variables es un paso hacia la generación de código eficiente. También es importante evitar el alojamiento de memoria dinámica siempre que sea posible. A continuación se mostrará el análisis de un código simple a diferentes niveles, mostrando que el lenguaje nos permite observar la generación de código, que finalmente nos da cierto control y nos permite verificar que lo que se está implementando es lo que se especifica en el código. Esto no es posible en lenguajes como Python.

```
1 let
2     e = 1.1
3     println(e*e)
4     @code_typed e*e
5 end
```

```
1.2100000000000002

CodeInfo(
1   %1 = Base.mul_float(x, y)::Float64
        return %1
) => Float64
```

En este código, se utiliza la estructura de agrupación de expresiones `let...end`. Cada expresión puede estar compuesta de otras expresiones, y casi todo es una expresión en Julia. La mayoría de las expresiones serán finalizadas por un salto de línea, pero las compuestas como `let`, `begin`, `function`, `if`, `while`, `for`, `do`, `module` estarán finalizadas con `end`. La indentación no importa como en Python, pero es aconsejable para mantener la legibilidad del código. La línea 2 define e inicializa la variable `e`; la línea 3 llama a la función `println`, que imprimirá el resultado de `e*e` en la consola. La función `println` está dentro de la biblioteca estándar de Julia y siempre está visible. La línea 4 es un tanto diferente, es una macro que toma la expresión `e*e` y realiza algo sobre la expresión misma, en particular `@code_type` muestra cómo se reescribe la expresión para ser ejecutada. Note cómo se hará una llamada a la función `Base.mul_float` que recibe dos argumentos y que regresará un valor `Float64`. Esta información es necesaria para que Julia pueda generar un código veloz, el flujo de compilación llevaría esta información a generar un código intermedio de *Low Level Virtual Machine* (LLVM), que es el compilador empotrado en Julia, el cual estaría generando el siguiente código LLVM (usando la macro `@code_llvm`):

```
;  @ float.jl:411 within `*`
define double @"julia_*_2376"(double %0, double %1) #0 {
top:
  %2 = fmul double %0, %1
  ret double %2
}
```

Este código ya no es específico para Julia, sino para la maquinaría LLVM. Observe la especificidad de los tipos y lo corto del código. El flujo de compilación requeriría generar el código nativo, que puede ser observado a continuación mediante la macro `@code_native`:

```
    .text
    .file   "*"
    .globl  "julia_*_2415"                  # -- Begin function julia_*_2415
    .p2align    4, 0x90
    .type   "julia_*_2415",@function
"julia_*_2415":                             # @"julia_*_2415"
;   @ float.jl:411 within `*`
# %bb.0:                                    # %top
    push    rbp
    mov rbp, rsp
    vmulsd  xmm0, xmm0, xmm1
    pop rbp
    ret
.Lfunc_end0:
    .size   "julia_*_2415", .Lfunc_end0-"julia_*_2415"
;
                                            # -- End function
    .section    ".note.GNU-stack","",@progbits
```

En este caso podemos observar código específico para la computadora que está generando este documento, es posible ver el manejo de registros y el uso de instrucciones del CPU en cuestión.

Este código puede ser eficiente dado que los tipos y las operaciones son conocidos, en el caso que esto no puede ser, la eficiencia está perdida. Datos no nativos o la imposibilidad de determinar un tipo causarían que se generara más código nativo que terminaría necesitando más recursos del procesador. Una situación similar ocurre cuando se aloja memoria de manera dinámica. Siempre estaremos buscando que nuestro código pueda determinar el tipo de datos para que el código generado sea simple, si es posible usar datos nativos, además de no manejar o reducir el uso de memoria dinámica.

1.6. Ejemplos de funciones

Las funciones serán una parte central de nuestros ejemplos, por lo que vale la pena retomarlas y dar ejemplos.

```
function f(x)
    x^2
end
```

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

Siempre regresan el valor de la última expresión; note cómo el tipo (y no solo el valor) de retorno depende del tipo de la entrada, e.g., si `x` es un entero entonces `x^2` será un entero, pero si `x` es una matriz, `x^2` será una matriz.

Hay valores opcionales y kwargs, ambas tienen características diferentes:

```
function f(x, t=1)
    (x+t)^2
end

function g(x; t=1)
    (x+t)^2
end
```

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

1.7. Definición de estructuras

```
struct Point
  x::Float32
  y::Float32
end
```

La idea suele ser que todo se use de manera armoniosa

```
"""
  Calcula la norma de un vector representado
  como una tupla
"""
function norm(u::Tuple)
  s = 0f0

  for i in eachindex(u)
    s += u[i]^2
  end

  sqrt(s)
end

"""
  Calcula la norma de un vector de 2 dimensiones
  representado como una estructura
"""
function norm(u::Point)
  sqrt(u.x^2 + u.y^2)
end
```

```
(norm((1, 1, 1, 1)), norm(Point(1, 1)))
```

```
(2.0f0, 1.4142135f0)
```

1.8. Arreglos

Una matriz aleatoria de 4×6 se define como sigue

```
A = rand(Float32, 4, 6)
```

```
4×6 Matrix{Float32}:
 0.537205  0.767044  0.630408   0.860311  0.514467  0.0667717
 0.290038  0.306325  0.0707925  0.556101  0.606056  0.00703126
 0.47159   0.20139   0.0234204  0.577445  0.600936  0.306406
 0.162078  0.867685  0.610756   0.731511  0.988312  0.131053
```

Un vector aleatorio de 6 dimensiones sería como sigue:

```
x = rand(Float32, 4)
```

```
4-element Vector{Float32}:
 0.54791504
 0.30822784
 0.24923694
 0.3885581
```

entonces podríamos multiplicar `x` con `A` como sigue:

```
y = x' * A
```

```
1×6 adjoint(::Vector{Float32}) with eltype Float32:
 0.564255  0.902033  0.610382  1.07094  1.00248  0.166042
```

```
y'
```

```
6-element Vector{Float32}:
 0.56425464
 0.9020327
 0.6103818
 1.0709383
 1.0024796
 0.16604197
```

También existen otras formas para realizarla, aunque no suelen ser la mejor idea si se tienen alternativas canónicas:

```
using LinearAlgebra

dot.(Ref(x), eachcol(A))
```

```
WARNING: using LinearAlgebra.norm in module Notebook conflicts with an existing identifier.
```

```
6-element Vector{Float32}:
 0.56425464
 0.9020327
 0.6103818
 1.0709383
 1.0024796
 0.16604199
```

Este ejemplo muestra la técnica *broadcasting* que aplica una función a una colección; se indica añadiendo un punto al final del nombre de la función. Adicionalmente, hay una serie de reglas que se deben seguir para el manejo de las colecciones. La función `eachcol` crea un iterador sobre cada columna de la matriz `A` y `Ref(x)`, nos permite que el `broadcasting` reconozca al vector `x` como un único elemento en lugar de una colección de valores.

1.8.1. Ejercicios

Dados dos vectores, cree las siguientes funciones:

1. Calcule el coseno entre dos vectores u, v, de dimensión d:
   - $cos(u, v) = \frac{\langle u,v \rangle}{\|u\|\|v\|}$; donde $\langle u, v \rangle = \sum_i^d u_i \cdot v_i$ y $\|\cdot\|$ es la norma de un vector.
2. Calcule la distancia Euclidea entre dos vectores u, v:
   - $euclidean(u, v) = \sqrt{\sum_i^d (u_i - v_i)^2}$.

1.9. Paquetes y módulos

El ecosistema de paquetes de Julia es una de sus mayores fortalezas, impulsado por su gestor de paquetes *Pkg*, el cual viene integrado en el REPL y en su instalación mínima; es muy robusto. Se encarga de la instalación y actualización de librerías, así como también garantiza la reproducibilidad de los proyectos. Cada ambiente de trabajo en Julia utiliza archivos como `Project.toml` y `Manifest.toml` para registrar la paquetería usada, así como las versiones necesarias de todas las dependencias.

1.9.1. Pkg en REPL

Para entrar al modo Pkg desde cualquier sesión de Julia en el REPL se debe teclear el corchete que cierra `]`.

El prompt del REPL cambiará:

Modo	Prompt
Normal (Julia)	`julia>`
Modo Pkg	`(@v1.10) pkg>`

Ahora se puede ver qué paquetes están instalados en tu entorno actual.

Comando	Acción
`st` o `status`	Muestra la lista de todos los paquetes instalados y sus versiones específicas.

Para añadir una *paquete* a tu entorno, usa el comando `add`.

Comando	Acción
`add Paquete`	Descarga e instala el paquete.

En el caso de que un paquete ya no sea necesario, se puede desinstalar con el comando `rm` (de *remove*).

Comando	Acción
`rm Paquete`	Elimina el paquete del entorno actual.

Finalmente, es posible actualizar paquetes de manera individual o colectiva usando el comando `up`.

Comando	Acción
`up` o `update`	*Actualiza* todos los paquetes instalados a su última versión compatible.
`up Paquete` o `update Paquete`	Actualiza `Paquete` a su última versión compatible.

1.9.1.1. Manejo de ambientes

El entorno o ambiente (*environment*) se puede especificar de manera global o por directorio, y nos sirve para aislar las aplicaciones y no entrar en dificultades por versiones.

Comando	Acción
`activate dir`	Activa el directorio `dir` como ambiente.

Supongamos que nos comparten un proyecto escrito en julia, lo primero que debemos hacer es activar e instanciar el ambiente; la instanciación es como sigue

Comando	Acción
`instantiate`	Se instalan todos los paquetes indicados por el ambiente.

Muchas veces también cambiamos paquetes locales que requieren reactualizar el ambiente, eso se consigue con `] resolve` que actualizará las nuevas dependencias que cambiaron.

1.9.1.2. Saliendo del modo Pkg

Para volver al modo de ejecución de código normal de Julia, se debe presionar *backspace*.

El prompt cambiará de nuevo a `julia>` y podrás usar los paquetes que instalaste con el comando `using`.

```
using DataFrames, CSV
```

esto traerá el paquete al entorno en memoria haciendo accesibles sus métodos y estructuras públicas.

1.9.2. Usando `Pkg` desde el modo normal de Julia (fuera del modo Pkg del REPL)

Existe un paquete interno de las instalaciones de Julia llamado `Pkg` que es el que maneja todo lo anterior, este puede ser utilizado como cualquier paquete. Básicamente tiene funciones similares a las del modo `Pkg` (con nombres completos).

Ejemplos:

```
julia> import Pkg
julia> Pkg.add("PlotlyLight")
julia> Pkg.add(["CSV", "DataFrames"])
julia> Pkg.rm("PlotlyLight")
julia> Pkg.update()
julia> Pkg.status()
```

Ahora para el manejo de los ambientes:

```
julia> import Pkg
julia> Pkg.activate(".")
julia> Pkg.instantiate()
julia> Pkg.add("Statistics")
```

1.10. Otras estrategias para la organización de código

La función `include("nombre_archivo.jl")` es el método más simple en Julia para organizar código en múltiples archivos. Su función es equivalente a *copiar y pegar* el contenido del archivo especificado directamente en la línea donde se llama a `include`.

Sirve para estructurar grandes *scripts* en archivos más pequeños y manejables; el código incluido se ejecuta en el mismo *alcance* (scope) donde se llamó a `include`. Si llamas a `include` en el alcance global, las funciones y variables definidas en el archivo incluido se vuelven globales. Si lo llamas dentro de un módulo, se vuelven parte de ese módulo.

Es simple, pero no proporciona aislamiento, y puede generar conflictos de nombres si no se usa de manera adecuada.

Por otro lado, los *módulos* permiten crear *espacios de nombres (namespaces)* aislados y bien definidos, útiles para organizar proyectos grandes y complejos.

1.10.1. Aislamiento y alcance (Scoping)

Un módulo actúa como una *caja* que encierra sus funciones y variables. Todo lo que se define dentro de un módulo es *privado* por defecto para evitar conflictos de nombres con código externo.

```
module MiCalculadora
    # Esta función es PRIVADA
    function interna(x)
        return x * 2
    end

    # Esta función se hace PÚBLICA con 'export'
    export sumar

    function sumar(a, b)
        return a + b
    end
end
```

Para que las funciones, tipos o constantes dentro de un módulo sean accesibles desde afuera, deben ser explícitamente *exportadas* utilizando la palabra clave `export`.

Los módulos pueden anidarse.

Para utilizar las funciones de un módulo en otro script o en el REPL, se usan dos comandos principales:

Comando	Acción
`using NombreModulo`	Importa solo los símbolos que han sido **exportados** por el módulo.
`import NombreModulo`	Importa el módulo completo. Para usar sus funciones, debes prefijarlas (ej: `NombreModulo.sumar(1, 2)`).

En la práctica, un paquete o un proyecto grande de Julia casi siempre usa tanto `include` como módulos. De esta manera, `include` ayuda a la organización de archivos, mientras que el bloque `module` garantiza que todo el código esté contenido en un espacio de nombres único y limpio, evitando colisiones.

En particular, los *paquetes* pueden verse como la preparación de un módulo para su distribución, indicando los paquetes que usan (dependencias) y sus versiones especificas para los cuales fueron diseñados. También suelen incluir documentación y pruebas unitarias.

1.11. Recursos para aprender más sobre el lenguaje

- Información sobre como instalar Julia y flujos de trabajo simples (e.g., REPL, editores, etc.) para trabajar con este lenguaje de programación: *Modern Julia Workflows* https://modernjuliaworkflows.github.io/.
- Libro sobre julia *Think Julia: How to Think Like a Computer Scientist* https://benlauwens.github.io/ThinkJulia.jl/latest/book.html.
- Curso *Introduction to computational thinking* https://computationalthinking.mit.edu/Fall20/