Introducción a Julia

Introducción sobre Julia.

Julia es un lenguaje de programación que es a la vez de alto nivel y tiene una gran velocida de ejecución. Además, a diferencia de otros lenguajes como R o Phthon, las librerías de Julia están escritas completamente en Julia. En Python y R hay librerías que, para maximizar su velocidad, están implementadas en otros lenguajes en los que se puede lograr una mayor velocidad de ejecución, como es el caso de C o C++.

Julia y su ecosistema de paquetes tienen cinco características que son relevantes para aquellos que deseen realizar análisis de datos:

- Velocidad de ejecución del código (puesto que es compilado).
- Ha sido diseñando con la capacidad de uso interactivo en mente.
- Presenta facilidade para escribir código reutilizable.
- Sistema de gestión de paquetes integrado.
- Fácil integración con otros lenguajes.

Julia es un lenguaje compilado

TODO-incluir información sobre el sistema de compilación. LLVM.

Julia también permite realizar ejecución de código en paralelo (utilizando varios núcleos de la CPU) y en de manera distribuida (utilizando varios ordenadores). Además, por medio de paquetes como `CUDA.jl` puede ejecutar código en GPUs.

Julia soporta flujos de trabajo interactivos

Aunque Julia es un lenguaje compilado, también permite ejecutar código de manera interactiva. Esto se logra mediante alguno de los siguientes métodos:

- REPL (read-eval-print loop) de Julia, es decir, la shell de Julia.
- Jupiter notebooks.
- Los cuadernos del paquete `Pluto.jl`. Se diferencian de los cuadernos de Jupyter en que, por ejemplo, cuando cambiamos algo en el código `Pluto.jl` actualizará todos los valores que dependan de él en todo el cuaderno.
  En todos estos casos el código se compila siempre que el usuario lo ejecute aunque esto sea de manera transparente para él.

Entorno de trabajo

Instalación de Julia y entorno de trabajo.

Posibilidades: Pluto.js / IDE y Julia REPL.

Pluto.jl

Plutojl es un paquete de Julia que nos permite trabajar con *notebooks* al estilo de Jupyter. La principal diferencia con Jupyter es que en Plutojl las celdas se ejecutan de forma secuencial y no de forma independiente.

Para utilizar Plutojl primer hemos de instalar el paquete:

```
import Pkg

Pkg.add("Pluto")
```

Una vez instalado el paquete podemos abrir un *notebook* con el siguiente comando:

```
import Pluto
Pluto.run()
```

Esto abrirá una página web en la que podremos crear un nuevo *notebook* o abrir uno ya existente.

Markdown en Pluto.jl

Para crear una celda de texto en formato Markdown en Pluto.jl hemos de hacerlo de escribir `md` seguido de una cadena de texto entre comillas. Por ejemplo:

```
md"# Título 1

La razón de la sinrazón que a mi razón se hace, de tal manera mi razón
enflaquece, que con razón me quejo de la vuestra fermosura.

## Título 2

La palabra *razón* es muy **importante** en el texto anterior porque es la única
palabra que se repite."
```

VS Code + Julia REPL

Visual Studio Code incluye un plugin para Julia que permite ejecutar código en Julia REPL. REPL son las siglas de *read-eval-print loop* y hacen referencia a una *shell* interactiva que permite escribir y ejecutar código Julia. A diferencia de los REPL de Python y Nodejs el código que vamos a escribir en el REPL de Julia **no es interpretados** si no que se compila y a continuación se ejecuta y muestra el resultado.

Visual Studio Code dispone de un plugin para Julia que nos permite trabajar con el lenguaje de programación de una forma más cómoda. Para instalar el plugin hemos de ir a la pestaña de extensiones y buscar `Julia`.

Comentarios en Julia

Para comentar una línea en Julia ha de usarse el símbolo `#` al comienzo de la línea.

Si queremos comentar varias líneas podemos usar la notación `#=` y `=#`.

Tipos de datos

Tipos simples

No tipado pero con opción de declarar el tipo de una variable usando el operador `::`.

Los tipos de variables más comunes en análisis de datos son:

- Enteros: `Int64`.
- Número reales: `Float64`.
- Booleanos: `Bool`.
- Cadenas de texto: `String`.

Respecto a los enteros, si necesitamos menos o más precisión también disponemos de `int8` e `int128`.

Para crear una nueva variable podemos hacerlo de la misma forma que en Pyhton, es decir, con el operador `=`:

```
nombre = `Manuel`

edad = 25
```

Si queremos especificar el tipo de una variable podemos hacerlo con el operador `::`:

```
edad::Int8 = 25
```

Podremos comprobar el tipo de una variable con la función `typeof`:

```
typeof(edad)
Int8
```

Tipos de definidos por el usuario

En Julia el usuario puede definir tipos compuestos de datos. Estes tipos se definen usando la palabra reservada `struct`:

```
struct Lenguaje
    nombre::String
    creador::String
    año::Int64
end

julia::Lenguaje = Lenguaje("Julia", "Jeff Bezanson", 2012) # Dice el Copiloto.
Lenguaje("Julia", "Jeff Bezanson", 2012)
```

Podemos consultar qué campos tiene un tipo struct con la función `fieldnames`:

```
fieldnames(Lenguaje)
(:nombre, :creador, :año)
```

Para hacer referencia a un campo de un tipo `struct` usamos el operador `.`:

```
julia.creador
"Jeff Bezanson"
```

Los datos `struct` son **inmutables**, es decir, no se pueden modificar una vez creados. Si queremos crear un tipo de datos que sea mutable podemos hacerlo con la palabra reservada `mutable struct` al definir el `sturct`.

Es recomendable usar datos inmutables siempre que sea posible, ya que son más eficientes y menos propensos a errores.

Operadores lógicos y comparaciones

Los valores booleanos en Julia son `true` y `false`. Los operadores lógicos son:

- `&&`: and.
- `||`: or.
- `!`: not.

```
!true
false

(10 isa Int64) && (10 > 5)
true
```

Las comparaciones de valores en Julia son:

- `==`: igual.
- `!=` o ≠: distinto.
- `>`: mayor que.
- `<`: menor que.
- `>=` o ≥: mayor o igual que.
- `<=` o ≤: menor o igual que.

Funciones

Para definir una función en Julia usamos la palabra reservada `function`:

```
function suma(a::Int64, b::Int64)::Int64
    return a + b
end
```

Existe una forma compacta de definir funciones en Julia, que es la siguiente:

```
suma(arg1, arg2) = arg1 + arg2
```

Se pueden declarar distintas funciones con el mismo nombre pero con distintos tipos de argumentos:

```
function suma(a::Int64, b::Int64)::Int64
    return a + b
end

function suma(a::String, b::String)::String
    return a * b
end
```

Argumentos por referencia

Los argumentos de una función en Julia **siempre se pasan por referencia**, es decir, si modificamos el valor de un argumento dentro de una función, este cambio se verá reflejado fuera de la función.

```
function swap!(x, y)
    x, y = y, x
end

x = 1
y = 2

swap!(x, y)

print("x = $x, y = $y.") # x = 2, y = 1.
```

Cuando una función modifica alguno de sus argumentos se suele añadir un signo de exclamación `!` a continuación del nombre de la función. Esto es una convención en Julia, el `!` no modifica el comportamiento de la función.

Múltiples argumentos de retorno

Las funciones de Julia pueden devolver más de un valor. Para hacerlo usamos la palabra reservada `return` seguida de una tupla con los valores a devolver:

```
function swap (x, y)
   return (y, x)
end
```

Los paréntesis para indicar la tupla son opcionales:

```
function swap (x, y)
   return y, x
end
```

A la ora de recibir esos resultados múltiples se indicará también una tupa:

```
(x, y) = swap(1, 2) # x = 2, y = 1

# También se pueden omitir los paréntesis:

x, y = swap(x, y)  # x = 1, y = 2
```

Si queremos ignorar algunos de los valores devueltos en lugar de indicar una variable para que reciba el valor usaremos el guión bajo `_`.

Funciones anónimas

Cuando sólo necesitamos usar una función de una manera puntual (por ejemplo indicar una función de comparación para un *sort* o en funciones de filtrado) por lo que no hace falta asignarle un nombre. Para definir una función anónima se utiliza el operador `->`:

```
x -> x^2
```

Argumentos con nombre

En las funciones también podemos especificar argumentos con nombre. Si lo hacemos estos tienen que ir después de los argumentos sin nombre y separados por un punto y coma `;`:

```
function union(str1::String, str2::String; prefijo::String = "",
separador::String = " ",
    sufijo::String = "")
    return prefijo * str1 * separador * str2 * sufijo
end

# La llamada a la funcón sería:

resultado = union("Hola", "Mundo", separador=" ", prefijo="¡", sufijo="!")

# Nótese que no es necesario mantener el orden ni icluir todos los argumentos con
nombre.
```

Funciones con despacho múltiple

En Julia podemos definir distintas implementaciones, o métodos, para cada función. En el momento de ejecutar el código se seleccionará una función de los argumentos que se definan para la función.

```
function suma(a::Int, b::Int)::Int = a + b

function suma(a::String, b::String)::String = a * b
```

Broadcasting

En Julia podemos aplicar una función u operador a un array de una forma muy sencilla. Para ello usamos el operado `.`:

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

b = a .+ 1
```

Estructuras de control

Condicionales

La sintaxis de los condicionales en Julia es similar a la de otros lenguajes de programación:

```
x = 10
y = 5

if x > y
    println("$x es mayor que $y")
elseif x < y
    println("$x es menor que $y")
else
    println("$x es igual a $y")
end
```

Bucle for

Los bucles for en Julia son similares a los de Python ya que no se usa una condición si no que se itera sobre una colección de elementos (como por ejemplo un rango):

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

La variable `i` tomará los valores del rango `1:5`, del 1 al 5 en cada iteración.

Bucle while

Este bucle sí que permite usar una condición para determinar si se sigue iterando o no:

```
x = 5

while x > 0
    println(x)
    x -= 1
end
```

Macros

Las macros es la forma que tiene Julia de implementar metaprogramación. El concepto de metaprogramación es el de un lenguaje disponga de mecanismos para escribir código que, a su vez, sea capaz de transformar otro código. De esta forma, las macros son un tipo de funciones que tienen como entrada una representación del ***código fuente*** del lenguaje y que proporcionen como resultado código fuente nuevo antes de que se iniciar el proceso de compilación.

Las macros de Julia se nombran con `@` seguido del nombre de la macro. Por ejemplo, la macro `@time` mide el tiempo de ejecución de una expresión.

```
@time suma(1, 2)
0.000004 seconds
3
```

Otras dos macros que utilizaremos en estes apuntes son `@show`, `@assert`, `@benchmark` y `@view` (esta última relacionada con las vistas de arrays).

Colecciones en Julia

Arrays

En Julia los arrays forman parte de la especificación del lenguaje. Se incluye una sintaxis especial para trabajar con arrays y son muy rápidos.

Creación de arrays

Una de las formas más sencillas de crear un array es "escribir" la matriz entre corchetes:

```
julia> arr = [1 2 3
              4 5 6
              7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

Si queremos comprobar el tamaño y dimensiones de una matriz usaremos la función `size`. `size(arr)` nos mostrará el tamaño de la matriz `(3, 3)` y si indicamos una dimensión `size(arr, 1)` nos mostrará el tamaño del eje *x* del array (las filas), `size(arr, 2)` el número de columnas, etc.

En caso de que queramos especificar el tipo de los elementos de la matriz usaremos la siguiente sintaxis:

```
julia> arr = Int8[1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int8}:
 1  2  3
 4  5  6
 7  8  9
```

Finalmente, también podemos usar el constructor `Array` para crear arrays:

```
julia> arr = Array{Int8, 2}(undef, 3, 3)
3×3 Matrix{Int8}:
 -80  -119   0
 -58   -34   0
   3     1  32
```

Tuplas vs Arrays

Otra estructura de Julia muy similar a los arrays son las tuplas. La diferencia entre las tuplas y arrays es que la primera tiene una **dimensión fija** y es **inmutable**. Esto quiere decir que no se le pueden añadir datos ni cambiar sus valores. Por este motivo las tuplas son más rápidas que las arrays.

Un ejemplo de uso de las tupas es cuando necesitamos que una función devuelva más de un resultado. En este caso lo que hacemos es devolver una tupa `return (x, y)` con los valores.

Las tuplas se crean de la misma forma que los array pero usado `()` en lugar de `[]`.

Indexado

En primer lugar hemos de indicar que las posiciones de los arrays y tuplas de Julia empiezan en 1.

En general, para hacer referencia a una posición de una matriz *n-dimensial* hemos de indicar la posición de cada eje separado por comas: $X = A[I_1, I_2, \ldots, I_n]$ donde cada I_k puede ser un entero, un array de enteros (para seleccionar determinadas posiciones) o un `:` si queremos seleccionar todos los elementos de dicha dimensión.

Por ejemplo, si queremos acceder al elemento de la fila 2 y columna 3 de una matriz `arr` haremos `arr[2, 3]`. Si queremos **copiar** la segunda columna de una matriz lo que haremos es: `col2 = arr[:, 2]`. Nótese que hemos dicho **copiar** pues si modificamos el valor de `col2` no se modificará el valor de `arr`.

También hemos de tener en cuenta que al hacer copias habrá que reservar espacio nuevo e inicializarlo con los valores de la matriz con el coste computacional que ello conlleva.

Si no deseamos realizar una copia de la matriz sino trabajar con una vista de la matriz original podemos usar la macro `@view`.

Vistas

Para crear una vista hemos de usar la macro `@view` y a continuación indicar la matriz y las posiciones que queremos ver. Por ejemplo, si queremos ver la primera fila de una matriz `arr` haremos `view(arr, :, 1)` o bien `@view arr[1, :]`.h

Comprehensions

Las *comprehensions* en Julia son, al igual que las *list comprehensions* de Python, una forma de crear arrays de forma compacta a partir de una sentencia.

La sintaxis es la siguiente:

```
arr = [f(x) for x in iterable]
```

Donde `f(x)` es la función que queremos aplicar a cada elemento de `iterable`.

Si queremos crear una matriz con los valores de la media de cada columna de otra matriz podríamos hacerlo de la siguiente forma: