Basics on Julia

Marco Mendoza

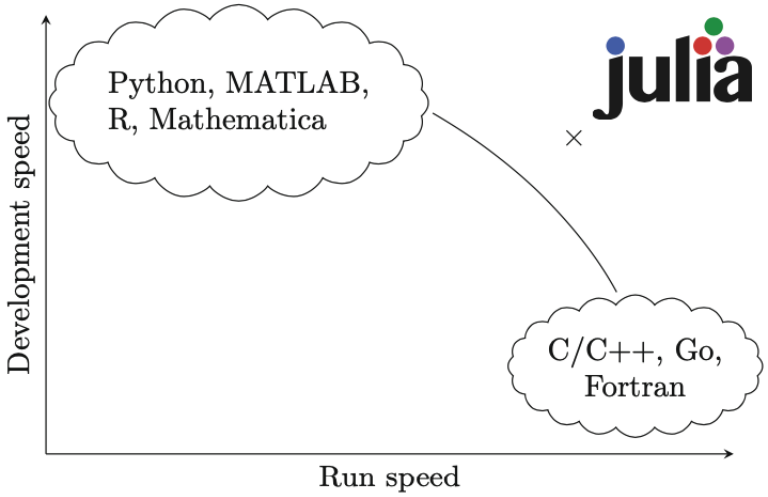
July 7, 2022

Contents

1 Introduction

Programming goes hand in hand with mathematics, statistics, data science, and many other fields. Scientists, engineers, data scientists, and statisticians often need to automate computation that would otherwise take too long or be infeasible to carry out. This is for the purpose of prediction, planning, analysis, design, control, visualization, or as an aid for theoretical research. Often, general programming languages such as *Fortran, Python, Go, C++*, are used. Fore more mathematical statistical purposes other languages such as *Mathematica, MATLAB/Octave, R and Maple* also are used. The process of coding involves analyzing a problem "by hand", witring the first version of

the code, analyzing behavior and output, re-factoring, iterating and improving the model/code. At the end of the day a critical component is **speed**, specifically the amount of time taken by solving the problem fomr the very beginning to its final solution.



2 About Julia

Julia is first and foremost a scientific programming language. It is perfectly suited for statistics, machine learning, data science, as well as for light and heavy numerical computational tasks. It can also be integrated in user-level applications; however, one would not typically use it for front-end interfaces or game creation. It is an open-source language and platform, and the Julia community brings together contributors from the scientific computing, statistics, and data-science worlds. This puts the Julia language and package system in a good place for combining mainstream statistical methods with methods and trends of the scientific computing world.

3 Setup and Interface

3.1 REPL Command Line Interface

The **Read Evaluate Print Loop** (REPL) commmand line interfae is a simple and straightforward way to use Julia. To install it, just run in your

Mac, in case you have some problems intalling it by brew package-manager, go here and follow the instructions.

```
brew intall julia
```

once you succesfully download and install Julia in your machine, you can now launch the REPL in your shell/command lkne environment just typing `Julia` on it. What you will see then is the main interface for Julia programming. When using the REPL, tipycaly one will also work with Julia files, which will have the `.jl` extentention.

3.2 The Package Manager

3.2.1 The Standar way

The simples and most used way to install packages in Julia (we will see in the next section why) is using the package `Pkg`. Inside the REPL or in a notebook, you must "import" the package in the first line and following the installation of the desired package. For example:

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

which will add the package `Foo.jl` to your current Julia build.

3.2.2 The Package manager

When using REPL you can enter the *package manager mode* by typing "]". This mode can be exited by hitting the backspace key. In this mode you can see

- `add Foo` adds the package `Foo.jl` to the current Julia build
- `status` lists what packages and versions are currently installed
- `update` updates existing packages
- `remove Foo` removes package `Foo,jl` from current Julia build.

3.2.3 Most used packages

- `Base.jl` is the basics Julia package
- `Calculus.jl` calculus operatoins including differential and integration both numerically and symbolically
- `CSV.jl` for working with CSV and other delimited files
- `DataFrames.jl` for brush your elephant
- `Dates.jl` provides support for working with dates/time
- `Plots.jl` the main plotting package
- `PyPlot.jl` in case you cannot forget your ex
- `Statistics.jl`
- `Random.jl`
- `LinearAlgebra.jl`
- `IJulia.jl` The most important package of all

3.3 Jupyter notebooks

An alternative for those who loves the noteboks is the usage of the package `IJulia`, in order to use it correctly you need to have installed in your system the *Jupyter Notebook*. When you install the package mentioned above, just run your notebook as always and when you clikc on *new* you will now have the option of create a Julia-notebok.

4 Basic Sintax

Quite similar than Python, there are a lot of different ways to says hello.

```
println("Hello world")
```

```
Hello world
```

in this particular case, the sintax is quite commmon to Python, the only difference is the `println` statement which includes the `ln` sufix.

For cicles, there are also the same general structures than python, for example, vector/lists. Here are two different ways to set a for statement in Julia, the most common and the python-object way.

```
hello_array = ["hello", "world"]

for i in 1:length(hello_array)
    println(hello_array[i])
end

println("----------------------")

for i in 1:2
    println(hello_array[i])
end

println("----------------------")

for word in hello_array
    println(word)
end
```

```
2-element Vector{String}:
 "hello"
 "world"
hello
world
----------------------
hello
world
----------------------
hello
world
```

You can also write comprehension lists in Julia

```
squares = [i^2 for i in 0:10]

println(squeares)
```

```
11-element Vector{Int64}:
   0
   1
   4
   9
```

```
 16
 25
 36
 49
 64
 81
100
```

The `function` definition and the `if` statement is also pretty similar to python. The next code is an example shows us a bubbleSort function.

```
function bubbleSort!(a)
    n = length(a)
    for i in 1:n-1
        for j in 1:n-i
            if a[j] > a[j+1]
                a[j], a[j+1] = a[j+1], a[j]
            end
        end
    end
    return a
end

data = [65, 51, 32, 12, 23, 84, 68, 1]
bubbleSort!(data)

bubbleSort! (generic function with 1 method)
8-element Vector{Int64}:
 65
 51
 32
 12
 23
 84
 68
  1
8-element Vector{Int64}:
  1
 12
 23
```

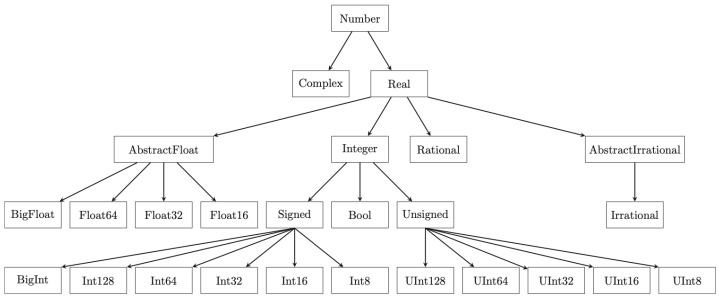
```
32
51
65
68
84
```

Note that the input argument `a` is implicitly expected to be an array. The function sorts `a` in place and returns a reference to the array. Also note that in this case, the function-name ends with a `!` by convention, this mark decorates the name of the function, letting us know that the function argument `a`, will be modified.

5 Types nd Multiple dispatch

Functions in Julia are invoked via multiple dispatch. This means the way a function is executed, i.e. its method is based on the type of its inputs, i.e. its argument types. Indeed functions can have multiple methods of execution, which can be checked using the `methods()` command.

Julia has a powerful type system which allows for user-defined types. One can check the type of a variable using the `typeof()` function, while the functions `subtypes()` and `supertype()` return the subtypes and supertype of a particular type, respectively. As an example, Bool is a subtype of Integer, while Real is the supertype of Integer. This is illustrated in the next Figure, which shows the type hierarchy of numbers in Julia.



for more information, you can follow this link.