# Mini course: Julia

## Lecture 1: Introduction

Friederike, Gaston & Juan

*friederike.metz@oist.jp*

February 22, 2021

## Why does Julia exist?

**Key Points**

1. Low-level languages are fast, but high level languages are readable.
2. Recent advances in compiler design could bridge these two aspects.
3. Julia is being openly developed by researchers, for researchers.

The old mission statement is available here and some good discussion is available at here.

**My personal reason**

A fast, open-source, high level language that is powerful enough to do serious numerical work, while being designed to encourage good code.

## The other contenders

The typical languages used in science are

1. Python
2. Matlab
3. R

Once a problem is becoming to big we usually move to

1. C/C++
2. Fortran

This is called the 2+ language problem and Julia is trying to solve that.

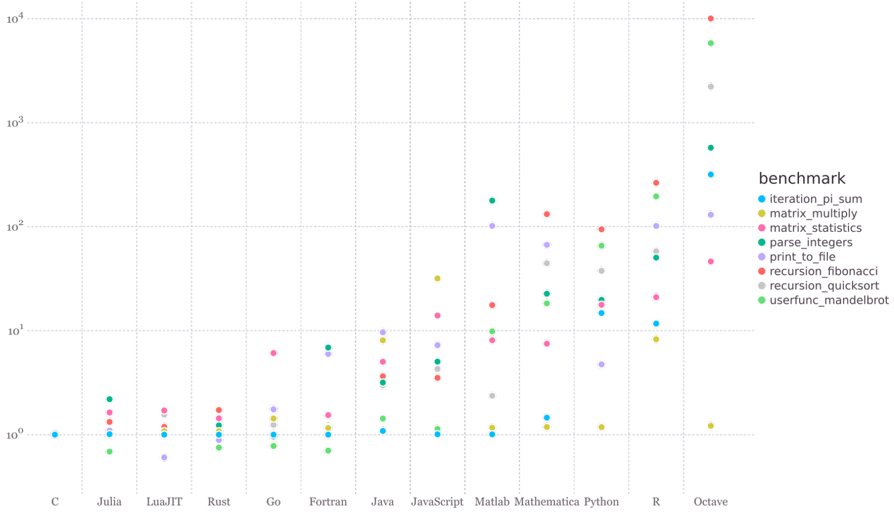
## Python and Numpy

- The compilers that exist (Numba) only work on primitive types and not user defined ones.
- GIL (Global Interpreter Lock) mask multi-threading hard.
- For fast code you need to write it in C.
- Numpy is great, but awful syntax for math.

## Matlab

- It costs a lot of money and is not open-source.
- Matlab will only be fast for a subset of operations.
- Matlab tends to hide the computer from the programmer.

# A (biased) performance comparision



## The REPL

### The Read-Eval-Print-Loop

The REPL is a command-line interface to Julia and is ideal for short experiments.

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _‘ |  |
  | | |_| | | | (_| |  |  Version 1.5.3 (2020-11-09)
 _/ |\__’_|_|_|\__’_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

In the `REPL` you can use `?` to switch your `REPL` mode into help mode and get information about functions.

## IDEs

There are two main IDEs that are *feature* complete and can be used for Julia. The main one is based on Atom and is called Juno `http://junolab.org/`.

The second one is based on on Visual Studio Code and available at `https://marketplace.visualstudio.com/items?itemName=julialang.language-julia`.

We will not be using these during the course, but you are welcome to try them out

## Jupyter

Jupyter is an interactive web-based client for Python, Julia, R and many other languages. It offers a programming environment that is well suited for explorative data analysis or prototyping.

### Installation

```
julia> ENV["JUPYTER"] = ""
julia> ] add IJulia
```

### Starting a Jupyter session

```
julia> using IJulia
julia> notebook()
```

### JuliaBox

There is an online service provided by JuliaComputing at `https://juliabox.com` that gives you a cloud version of Jupyter.

## Variables and datatypes

Julia is a dynamic language and so you can simply create variables in any scope.

```
x = 1  # x will be of type Int64
y = 1.0 # y will be of type Float64
z = 1.0 - 2.0im # z will be an Complex{Float64}
1//2 # Rational numbers
"This is a String"
"""
This is a multiline
String
"""
'C' # Character literal
1.0f0 # Float32 literal
```

Use `typeof` to check the type of any variable. Variable names can be unicode and so greek symbols can be used. In the `REPL` and most editors you can insert them by entering their LaTeXname and press `[Tab]`.

## Variables and datatypes

**Exercise**

Create a variable $\lambda$ that stores the value of $\pi$. Check the type of $\lambda$. Look up the docs for the `convert` function. Convert $\lambda$ to float.

**Solution**

```
# \lambda + <tab> = \pi + <tab>
λ = π
typeof(λ)
?convert
convert(Float32, λ)
```

## Data structures

```julia
# Tuple: (item1, item2, ...)
okinawa = ("Onna", "Nago", "Naha")
okinawa[1]

# NamedTuple: (name1 = item1, name2 = item2, ...)
okinawa = (central = "Onna", north = "Nago", south = "Naha")
okinawa.north

# Dictionaries: Dict(key1 => value1, key2 => value2, ...)
number = Dict("Jeremie" => "12345", "Juan" => nothing)
number["Juan"] = "4444"

# Arrays: [item1, item2, ...]
fibonacci = [1, 1, "two", 3, "five", 8, 13]
fibonacci[3] = 2
```

Julia is 1-based indexing, not 0-based like Python!!!
You can add or remove elements using `push!` and `pop!`.

## Conditionals

Julia has all the typical conditionals `if`, `else`, `elseif` which have to end in an `end`. Blocks in Julia are not whitespace sensitive and conditionals do not need to be wrapped in round brackets.

```
if rand() < 0.5
  println("Gaston awesome!")
else
  println("Juan is awesome!")
end

# Ternary operator: a ? b : c
rand() < 0.5 ? println("Gaston great!") : println("Juan
    great!")
```

&& is the and operator, || is the or operator. Booleans are `true` and `false`.

## Conditionals

**Exercise**

Print an error message if a variable `x` is greater than zero

**Solution**

```
if x > 0 error("x cannot be greater than 0") end

# Ternary operator
x > 0 ? error("x cannot be greater than 0") : nothing

# Short-circuit evaluation
(x > 0) && error("x cannot be greater than 0")
```

## Loops

Julia has for and while loops. A while loop takes a condition and a for loop takes a iteratior. One can use break to break out of a loop and continue to skip to the next iteration. It is noteworthy that a for loop can take an arbitrary iterator and even desugar tuples.

```julia
for (i, x) in enumerate(['A', 'B', 'C'])
  if x == 'B'
    continue
  end
  println(i)
end

while true
  # ternary operator ?!
  rand() < 0.1 ? break : println("You are trapped!")
end
```

## Functions and lambdas

Julia uses functions to organize operations. Every function is compiled for the combination of input parameters.

```
"""
    f(x, y)

‘f‘ will add two numbers together.
"""
function f(x, y)
  return x + y
end

g(x) = x^2

h = (x)-> 1/x
map(lowercase, ['A', 'B', 'C'])
map((x)->x+2, [1, 2, 3])
```

## Types

Julia's type system allows you to restrict functions to certain types and specialized functions for others. You can also create your own types. The names of types are typically captialized while functions are lowercase.

```
abstract type Entity end
mutable struct Player <: Entity
  mass::Float64
  name::String
  position::Tuple{Float64, Float64}
end
struct Object <: Entity
  position::Tuple{Float64, Float64}
end
newObject=Object(0,0)
```

## Multiple dispatch in a nutshell

Julia is not an Object-Oriented programming language so functions do not belong to an object. A function is a set of multiple methods each with their own signatures. When you call a function the most specific methods is executed.

```
function h(x::Number)
  println("x is most definitly a number.")
end
function h(x::Integer)
  println("x is a integer")
end
function h(x::Int8)
  println("Specific method for Int8")
end
```

This becomes really powerful when having multiple argurments and being able to select the most specific method.

## Modules

Julia code is organised as modules (namespaces). Module names are capitalised and you can nest modules as well.

```
module MyModule
  export f

  g() = "Internal function"
  f() = println(g())
end
using Main.MyModule
```

## Installing packages

Julia has an inbuilt package manager called `Pkg`. You can access it from the REPL by entering `]`. Julia packages end in `.jl` and it is customary to refer to them by their full name online, but within Julia you drop the `.jl`. So to install the Julia package `Distributions.jl` in the `REPL` just run:

```
] add Distributions
```

A few other commands:

```
# Updating the installed packages
] update
# What packages are installed?
] status
```

In order to create your own packages you have to install `PkgDev.jl`.

## Resources

| | |
|---|---|
| Documentation | `https://docs.julialang.org/en/v1.1/` |
| Forum | `https://discourse.julialang.org` |
| Issue Tracker | `https://github.com/JuliaLang/julia` |
| Downloads | `https://julialang.org/downloads/` |
| Packages | `https://pkg.julialang.org/` |
| | `https://juliaobserver.com/` |

## What is next?

**Question?!**

What do you want to hear learn about?

| | |
|---|---|
| Next Session | How to use Julia for computation and plotting |
| Next Tuesday | Problem Sets with James |