Mini course: Julia

Lecture 1: Introduction

Friederike, Gaston & Juan

material taken/based on Ankur Dhar & James Schloss

friederike.metz@oist.jp

March 1, 2021

1. Introduction
    - Why does Julia exist?
    - What is wrong with the status quo?
2. Getting started
3. Language syntax and semantics
4. The ecosystem

Why does Julia exist?

Key Points

1. Julia is a **general-purpose** programming language, but it was specifically developed for **researchers** and data scientists.
2. Low-level languages are fast, but high-level languages are readable.
3. Julia tries to be both: **high-level** and **fast** by taking advantage of recent advances in compiler design.
4. Julia is **free** and **open source**.

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

The other contenders

The typical languages used in science are

1. Python
2. Matlab
3. R

Once a problem is becoming too 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.
- For fast code you need to write it in C.
- GIL (Global Interpreter Lock) makes multi-threading hard.
- Numpy is great, but syntax for math is not optimal.

Julia

- uses a "just-ahead-of-time" compiler by default
- provides parallelism out of the box
- excels at numerical computing with a syntax that is great for math

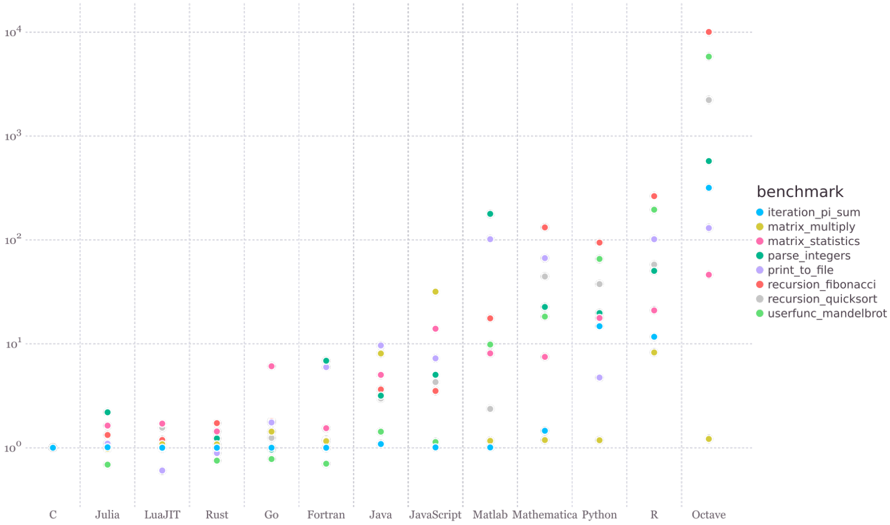
Matlab

- 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.

Julia

- is open-source
- is a general-purpose language and fast for “everything”

A (biased) performance comparison



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 **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> ] add IJulia
```

Starting a Jupyter session

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

Other notebook engines

Pluto.jl
Google Colab (example template)

Variables and datatypes

Julia is a dynamic language and you can simply create variables in any scope without specifying the type.

```julia
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
#=
This is a multiline comment
=#
```

Use `typeof(x)` to check the type of any variable `x`.

Variables and datatypes

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]`.

Exercise

Create a variable λ that stores the value of π. Check the type of λ. Look up the docs for the `convert` function. Convert λ to float.

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].

Exercise

Create a variable λ that stores the value of π. Check the type of λ. Look up the docs for the `convert` function. Convert λ to float.

Solution

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

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, ...)
password = Dict("Jeremie" => "12345", "Juan" => nothing)
password["Juan"] = "4444"

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

Julia uses 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("Jeremie is awesome!")
elseif rand() >= 0.5
  println("Gaston is awesome!")
else
  println("Juan is awesome!")
end

# Ternary operator: a ? b : c
s = rand() < 0.5 ? "Gaston  is awesome!" : "Juan  is awesome!"
```

`&&` is the logical and operator, `||` is the logical or operator.

Exercise

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

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 an iterator. One can use `break` to break out of a loop and `continue` to skip to the next iteration.

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

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

Exercise

Create a dictionary that holds integers (up to the value 100) as keys and their squares as values.

Exercise

Create a dictionary that holds integers (up to the value 100) as keys and their squares as values.

Solution

```
squares = Dict()
for i in 1:100
  squares[i] = i^2
end
println(squares)

# Array comprehension
squares_array = [i^2 for i in 1:100]
```

Functions and lambdas

Julia uses functions to organize operations. The first time a function is called it is compiled for the combination of its argument types.

```
function f(x, y)
  return x + y
end

g(x) = x^2
h = x -> 1/x

# map takes a function as input and applies that function to
    every element of the data structure you pass it.
map(lowercase, ['A', 'B', 'C'])
map(x->x+2, [1, 2, 3])
```

By convention, functions followed by ! modify their arguments and functions lacking ! do not, e.g. `sort()` vs. `sort!()`.

Types and multiple dispatch

The `::` operator annotates the type of variables. For example, `x::Int8` means that the value of `x` is an instance of `Int8`. There are abstract types (`Integer`, `Number`) and concrete types (`Int8`, `Float64`) and their relationships form a type graph.

In Julia a function is a set of multiple methods. Each method is defined for particular argument types and comes with its own implementation. When you call a function the most specific method is executed (dispatched).

```
h(x::Number) = println("x is most definitly a number.")

h(x::Integer) = println("x is an integer")

h(x::Int8) = println("Specific method for Int8")
```

Multiple dispatch makes your code generic and fast!

Types and structs

You can create your own types using `struct`.

```
abstract type Coordinate end

mutable struct Cartesian <: Coordinate
  position::Tuple{Float64, Float64}
end

struct Polar <: Coordinate
  radius::Float64
  phi::Real
end

point = Polar(1,pi)
point.radius
```

Keep in mind: Julia is not an Object-Oriented programming language so functions do not belong to a struct!

Exercise

Create a struct `Point` with two field names `x,y` both being of type Real. Implement a function `add` that takes two arguments of type `Point` and returns a new instance of Point holding the sum of the arguments fields `x`s and the difference of their `y`s.

Exercise

Create a struct `Point` with two field names `x,y` both being of type Real. Implement a function `add` that takes two arguments of type `Point` and returns a new instance of Point holding the sum of the arguments fields `x`s and the difference of their `y`s.

Solution

```
struct Point
  x::Real
  y::Real
end

function add(p1::Point, p2::Point)
  return Point(p1.x + p2.x, p1.y - p2.y)
end
```

Functions revisited

```julia
# Specifying a return type will convert the returned value
g(x, y)::Int8 = x * y

# Varargs (variable number of arguments)
h(a,b,x...) = (a,b,x)

# Optional arguments
f(x, y=1, z::Int64=2) = x + y + z

# Keyword arguments (seperated by a semicolon)
g(x, y; a=2, b::String="goya") = x+y+a, b
```

Everything above can be combined arbitrarily, e.g. keyword arguments can be collected using ...

Running julia scripts

There are mainly two different ways to run your julia scripts:

```
# in your command prompt or terminal
julia script.jl arg1 arg2...
# arguments will be placed in the global constant ARGS as a
    list of strings

# inside the REPL
include("script.jl")
```

Working from the `REPL` is preferred since all methods inside the script are compiled the first time and stay available as long as the session is not closed.

Try to break your programs down into lots of small specialized functions to really take advantage of the julia compiler and its speed-up.

Modules

If your julia files are becoming too large, organize them into modules. Module names are capitalised and you can also nest them.

```julia
module MyModule
  export f

  g() = "Internal function"
  f() = println(g())
end

# inside the REPL
include("MyModule.jl")

# Loading the module in a different file
using Main.MyModule
```

Additional advantage: Modules can be precompiled and saved to a file which means precompiled code won't disappear when closing the `REPL`.

Installing packages

Julia has an inbuilt package manager called `Pkg`. You can access it from the `REPL` by entering `]`. For example, to install and load the Julia package `Distributions.jl` in the `REPL` just run:

```
# Installing a package
] add Distributions
# Loading it afterwards
using Distributions
```

A few other commands:

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

Look here for more information about Julia packages.

Resources

Documentation	`https://docs.julialang.org/en/v1.5/`
Forum	`https://discourse.julialang.org`
GitHub	`https://github.com/JuliaLang/julia`
Slack	`https://julialang.org/slack/`
YouTube	`https://www.youtube.com/user/JuliaLanguage`
Learning	`https://julialang.org/learning/`

OIST also has a Slack workspace: julia@oist

What is next?

Questions?!

What do you want to hear/learn more about?

Wednesday	Arrays, data handling, plotting
Friday	Workflow, data structures, coding practices, algorithms

Exercise: Calculate π

Pseudocode

1. For each of N iterations
   1. Select a random point inside a square of area $4r^2$ as Cartesian, (x, y), coordinates. (You can choose $r = 1$ for simplicity.)
   2. Determine if the point also falls inside the circle embedded within this square of area πr^2.
   3. Keep track of whether or not this point fell inside the circle. At the end of N iterations, you want to know M – the number of the N random points that fell inside the circle!
2. Calculate π as $4\frac{M}{N}$

Exercise: Calculate π

Solution

```
function calculate_pi(N::Integer = 1000)
  M = 0
  for i in 1:N
    x = 1-2*rand()
    y = 1-2*rand()
    # Check if the distance to (x, y) from (0, 0) is less than
        the radius of 1
    if x^2 + y^2 < 1
      M += 1
    end
  end
  return 4 * M / N
end

map(calculate_pi, [10, 100, 1000, 10000, 100000])
```