Getting Started with Julia: Notes

Moritz M. Konarski

August 24, 2020

Contents

The Rationale for Julia

The scope of Julia

- born out of frustration with existing tools for technical computing
- prototyping needs a easy-to-use language, flexible, high-level language so the focus may be on the problem
- actual computation needs maximum performance hence for production things tend to be re-written in C or Fortran
- this lead to prototyping in slow but easy languages and then re-writes in difficult but fast languages
- Julia was designed to bridge this gap – using LLVM JIT (Just in Time) makes near-C speeds possible while keeping high-level usability

This resulted in:

- open source and liberal license (MIT)
- easy to use and learn, elegant, clear, dynamic, familiar, almost like pseudocode:

```
x -> 7x^3 + 30x^2 + 5x + 42
```

- Julia provides the needed speed without the need to switch languages
- metaprogramming to increase capability
- useable for normal computing tasks, not just pure computing
- easy-to-use multicore and parallel capabilities

Julia's place among the other programming languages

- Julia brings together the two worlds of typed and untyped languages
- Julia does not have a static compilation step but uses a type-inference engine to nonetheless deliver similar speeds
- types can still be used to make compilation easier and to document the code
- *dynamic multiple dispatch* is the approach to pick the best fitting function out of a pool of functions depending on the data type, it's basically polymorphism with type inference
- Julia does not have static type checking however, runtime errors can occur if data types do not match
- Julia also makes it easy to design pure functions and apply functional programming
- Julia is also suited for general purpose programming similar to Python

A comparison with other languages for the data scientist

- Julia's speed approaches C and leaves all other normal alternatives behind
- one of Julia's goals is that one never has to step down to C
- Julia is especially good at running MATLAB and R style programs

MATLAB

- the syntax should be very familiar for MATLAB users, but Julia is more general purpose
- most function names are similar to MATLAB and not R
- Julia is much faster than MATLAB, but it can also interface with it

R

- until now R has dominated statistics
- Julia has the same level of usability, but is 10 to 1000 times faster
- Julia also has an interface to R

Python

- Julia is again much faster than Python, reads similar to it, and can interface with it

Useful Links

- main website: http://julialang.org
- documentation: http://docs.julialang.org
- packages: http://pkg.julialang.org

Installing the Julia Platform

- if parallelization (n concurrent processes) is to be used, compile the julia code with

```
make -j n
```

Working with Julia's shell

- use `quit()` or `CTRL + D` to quit the REPL
- after an expression is evaluated, the result will be stored in the variable `ans`, but only in REPL
- assign values like so:

```
a = 3
```

- type annotations are not needed, they are inferred
- strings are defined by `"` (double quotes)
- to clear the screen but keep the data or variables, type `CTRL + L`
- to clear the workspace and variables, use `workspace()`
- all previous commands are stored in a `.julia_history` file at `/home/$USER/`
- typing `?` will give access to the docs, specific help is available through `help(<item>)`
- to find all the places a function is defined or used, type `apropos("<name>")`
- mulitple commands on one line are separated by `;`
- multi-line expressions also work and the shell will wait until the expression is complete

```
julia>  if 10 > 0
            println("10 is bigger than 0")
        end
10 is bigger than 0

julia>
```

- use tab for automatic completion, double tab to show the available functions
- starting a line with `;` makes the rest of the line a shell command
- to exit shell mode, type `backspace`
- the REPL can also execute written programs with

```
julia>  include("<name>.jl")
```

- the content of the file will then be executed
- for keybindings see here

Startup option and Julia scripts

- commands can be evaluated from the command line without starting the repl

```
julia -e 'a = 6 * 7; println(a)'
```

- a script taking arguments can be run like this

```
julia script.jl arg1 arg2 arg3
```

- the arguments are then available in the global constant `ARGS`
- files can also execute other files by calling `include("file.jl")` in them

Packages

- official Julia packages can be found at METADATA.jl at https://github.com/JuliaLang/METADATA.jl
- a searchable list can be found at http://pkg.julialang.org/
- Julia has a built-in package manager called `Pkg` for installing packages
- to find out which packages are installed, use `Pkg.status()`
- one of the better packages is IJulia, a jupyter mod

```
using PyPlot
x = linspace(0, 5)
y = cos(2x + 5)
plot(x, y, linewidth=2.0, linestyle="--")
title("a nice cosine")
xlabel("x axis")
ylabel("y axis")
```

How Julia works

- Julia uses LLVM JIT to generate machine code just-in-time
- the process works like this:
    1. when a function is run the types are inferred
    2. the JIT compiler turns the function into native machine code
    3. the next time a function is called the already compiled code is run (this is the reason that functions are faster the second time around – important for benchmarking)
- the code is dynamic because it is not dependent on the type of the variable
- these functions are by default *generic*, but JIT bytecode for specific types can be inspected like this

```
julia>  f(x) = 2x + 5
    f(generic function with 1 method)
julia>  code_llvm(f, (Int64,))
```

- the same can be done to inspect the assembly code using the function `code_native(f, (Int64,))`
- Julia automatically allocates and frees memory, it has a GC that runs at the same time as the program and is somewhat unpredictable

- calling `gc()` will call the GC, `gc_disable()` to disable it

Variables, Types, and Operations

- Julia is an optionally typed language – users can choose to specify the types
- typing in Julia is important for speed, documentation, tooling

Variables, naming conventions, and comments

- Julia differentiates between strings and characters, strings are denoted by double quotes, while characters are denoted by single quotes
- to see what type a variable or reference is one can use the `typeof(<var>)` function
- variables don't have to be typed, but they have to be initialized
- variables can change type, they can be over-written
- **everything** is a expression in Julia
- Julia is strongly typed
- variable names have to begin with a letter, then it can be letter, number, underscore, exclamation point, including unicode characters
- comments begin with `#` and are thus ignored
- multi line comments can be created with `#=` and terminated with `=#`
- colored output can be created with `print_with_color(:red, "I love Julia!")`
- objects are often interacted with in Julia – actions on objects are written functionally, like `action(object)` and not `object.action()`
- to display objects from code while outside of REPL, use `display(object)`

Types

- the type system is pretty unique, variables can be bound again to the same name