Getting Started with Julia: Notes

Moritz M. Konarski

August 25, 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
- if a variable is given a type, it will only accept variables of the same type
- type annotations are generally done as `var::TypeName` – type annotations make error checking and optimization easier

```
calculate_position(time::Float64)
```

- type assertions work in exactly the same way – an error is raised if the tpes differ

```
(expression)::TypeName
```

- to achieve top performance, types of variables have to remain stable
- Julia has a type hierarchy, with `Any` being the top – any undeclared variable is of this type
- the type `None` is one no object can have, but it is a subtype of all other types
- custom types can be defined – their named in CamelCase

- `typeof(x)` gives the type of variables, `isa(x, T)` checks if `x` is of type `T`
- data types are all of type `DataType`
- if data types can be converted by using the lower case type name

```
int64(3.14)
```

Integers and Booleans

- available integers are `Int8`, `Int16`, `Int32`, `Int64`, `Int128`
- unsigned integers are defined using `U` in front of the type name, like `UInt32`
- depending on machine architecture, the standard is either 32 or 64 bit
- this is given by the variable `WORD_SIZE`
- `typemin(var)` and `typemax(var)` give the min and max sizes of types
- if `typemax` is exceeded, overflow occurs
- explicit checking for overflow needs to occur
- other ways of writing integers are: `0b` (binary), `0o` (octal), `0x` (hexadecimal)
- in case arbitrary sizes are needed, `BigInt` can be used

```
BigInt("1234")
```

- they support the same operations as normal integers, but conversions are not automatic
- divisions of `BigInt` always give floating point results
- `div` gives integer division, `rem` gives the remainder, `^` gives powers
- `true` and `false` are actually 8bit integers, where `0` is `false` and anything else is `true`, `bool(32)` returns true
- other logical operations are `!`, `==`, `!=`, `<`, `>`
- logical expressions can be chained `0 < x < 3`

Floating point numbers

- follow IEEE 754
- come in `Float16`, `Float32`, `Float64`
- single precision floating point numbers: `2.5f2 == 2.5 * 10^2`
- double precision floating point numbers: `2.5e2 == 2.5 * 10^2`
- `BigFloat` is used for arbitrary precision floats
- `Inf` is infinity, `NaN` is not a number
- floating point numbers do cause errors because of the bad precision

Elementary mathematical functions and operations

- the bit representation of any number can be seen using the `bits(num)` operation
- `round()` rounds to float
- `iround()` rounds to int
- other functions: `sqrt()`, `cbrt()`, `exp()`, `log()`, `sin()`, `cos()`, `tan()`, `erf()` (error function), `rand()`
- `()` enforce precedence
- chained assignments are allowed `a = b = c = d = 1`
- assignments can be combined `a, b = c, d` means `a = c` and `b = d`

- bool operations have `||` and `&&`, `|` and `&` are used for non-short-circuit operations, those that check all the options even if the result is already found
- bitwise operations are supported http://docs.julialang.org/en/latest/manual/mathematical-operations/

Rational and complex numbers

- i is represented as `im`, `3.2 + 7.1im` has the type `Complex{Float64}`
- all normal maths functions are also implemented for this type
- `abs()` returns the absolute value
- `complex(a, b)` turns a and b into a complex number
- rational numbers are good for exact ratios
- `3//4` is a rational for $\frac{3}{4}$ and has the type `Rational{Int64}`
- `float()` converts to a floating point number, `num()` and `denum()` give the numerator and the denominator

Characters

- `'a'` is a char with type `Char` – a 32-bit integer with a unicode code point inside of it
- `int('A')` returns 65, `char(65)` returns A
- unicode can be entered using single quotes around `'\uxxxx'` or `'\Uxxxxxxxx'`
- `is_valid_char()` checks if the entered number is a valid character
- normal escape sequences also exist, `\n`, `\t`,...

Strings

- strings are always ASCII or UTF8, depending on the contained letters
- strings are immutable and contained in `"string"` or `'''string'''`
- arrays of chars starting at 1, so `s[1]` returns the first letter
- `s[end]` returns the end, `endof(s)` returns the last byte's index, `length()` the length
- iteration through a string is best done in an iteration and not an index

```
for c in string
    println(c)
end
```

- substrings can be obtained by string slices, `s[3:5]` or `s[2:end]`
- string interpolation

```
a = 3; b = 2;
s = "$a * $b = $(a * b)"
# s is "3 * 2 = 6"
```

- concatenate with

```
"abd" * "def" == string("abc", "def")
```

- **Symbols** are strings prefixed with a colon `:green`
- these more efficient strings are used as IDs or keys, they can't be concatenated
- `methodswith(String)` returns all the functions that use it

- `search(string, char)` – returns the index of the first char in the string
- `replace(string, str1, str2)` – replaces `str1` in `string` with `str2`
- `split(string, char or [chars])` – splits string at char, returns an array of strings – if there is no char given, split uses whitespace

Formatting numbers and strings

- formatted printing is done with the `@printf` macro, it takes one format string and one or more variables

```
name = "Paskal"
@printf("Hello, %s\n", name)
# prints "Hello, Paskal"
```

- to return the formatted string, use `@sprintf`
- `show()` prints a text representation of an object that is often more specific than simple `@printf`
  - integer `%d`

  ```
  @printf("%d\n", 100)                    #> 100
  ```

  - float `%.3f`

  ```
  @printf("%.2f\n", 2.346)                #> 2.34
  # or
  str = @sprintf("%.2f\n", 2.346)
  show(str)                               #> "2.34"
  ```

  - scientific `%0.3e`

  ```
  @printf("%.2e", 123)                    #> 1.23e+2
  ```

  - character `%c`

  ```
  @printf("%c", 'd')                      #> d
  ```

  - string `%s`

  ```
  @printf("%s", "hi")                     #> hi
  ```

  - right justify `%50s`

  ```
  @printf("%8s", "hi")                    #>       hi
  ```

- a specia type of string is the `VersionNumber`, preceded by a `v`, is used to run different code for different versions of Julia

```
v"0.3"
```

Regular Expressions

- Julia uses Perl syntax for regex, see http://www.regular-expressions.info/reference.html
- regular expressions are denoted by a string preceede by `r`

```
r"..."
```

- matching an email address

```julia
# '+' matches any non-empty group of characters
email_pattern = r".+@.+"
input = "john.doe@mit.edu"
println(ismatch(email_pattern, input))  #> true
```

- `ismatch(pattern, string)` returns a boolean whether or not the pattern is matched
- `match(patters, string)` returns `nothing` or `RegexMatch` if the pattern is found
- `RegexMatch` has the properties `match` (contains substring that matches), `offset` (position of match in string), `offsets` (start positions for each of the substrings), `captures` (captured substrings as tuple)
- capturing in regex can be done with parethesis `()`, line

```julia
email_pattern = r"(.+)@(.+)"
```

- this will capture the hostname and username of the email address
- `replace` also takes regex as inputs to replace by
- `matchall` returns all the matches in an array
- `eachmatch` returns an iterator of matches

Ranges and arrays

- ranges work as

``````julia
for i in <start>:(<interval>):<end>
    ...
end
````

- the start and end are inclusive, if the interval is one it can be omitted
- arrays are of type array and can only hold the same type of value
- `Array{T, n}` is type and dimension
- 1D arrays are also called vectors
`````julia
arr = [100, 25, 37]
``````

- arrays are really vectors and they grow dynamically
- arrays can also be constructed abstractly

```julia
arr = Array(Int64, 5)
show(arr)                #> [0,0,0,0,0]
```

- to add to an array, use push!

```julia
push!(arr, 1)
```

- arrays can also be created from slices

```julia
arr = [1:9]
```

- to allocate a certain amount of memory (so it does not need to be resized) to one array at once do

```julia
sizehint(arr, 10^5)
```

- array indices start at `1` and can be used in brackets `arr[1]` and `end` is the final element
- `eltype` gives element type, `length` the number of elements, `ndims` the number of dimensions, `size(arr, n)` gives the elements per dimension
- to turn an array into a comma separated string, use `join(arr, "sep")`
- slices can retrieve sub-arrays and they can also be assigned to

```
arr[1:3]    #> [1, 2, 3]
arr[1:3] = 1
arr[1:3] = [2, 3, 4]
```

Other ways to create arrays

- `zeros(n)` returns an array with n 0s, `ones` does the same for 1s
- `linspace(start, stop, n)` returns an array of n with equally spaced numbers from start to stop
- `fill!(arr, n)` returns an array filled with only ns
- to get random values, use

```
arr = rand(T, n)
# returns n random numbers of type T
```

- arrays can also be converted using the normal conversion functions

Some common functions for arrays

- concatenate arrays with `append!(arr1, arr2)`
- **any function ending in `!` modifies the first element it is given**
- `push!` and `pop!` work like on the stack
- `shift!` pops the first element, `unshift!` pushes an element to arr[1]
- `splice!(index)` removes the element at index
- `in(val, arr)` checks if val is in arr
- `sort` and `sort!` sort the arrays, the latter changes it
- `for` loops can iterate over arrays
- element wise operations are done with a point

```
a1 = [1, 2, 3]
a1 = [2, 4, 6]
a1 .* a2
```

- the dot product is done with `dot(a1, a2)`
- the sum over an array is `sum(a1, a2)`
- arrays are passed as references – they point to the same memory
- many functions are simply applied to each element in an array
- `repeat` repeats each element in an array a certain number of times
- `copy` makes a copy of the first level, `deepcopy` goes through everything recursively

Array of chars to string

- `join(arr)` turns a [char] into a string, `utf32(arr)` does the same

- to pass the contents of array as separate parameters, use the splice operator, like `string(arr...)`

Dates and times

- `time()` returns seconds since epoch
- `strftime(time())` returns more useful formatted time
- `Date` and `DateTime` functions return more useful and subtractable values

Scope and constants