

Introduction to Julia

Foreword

`julia` is a language which I have a love-hate relationship with. If the first programming language you learned is `python`, I think `julia` offers a fresh take on what you can do with computers while having the interactivensess of `python`. It has a lot of modern features built into the language, such as its just-in-time (JIT) compilation, multiple dispatch, and metaprogramming capabilities. It also comes with its own package manager, which is quite nice to use. This makes `julia` a great “advance” language for data scientists to learn after `python`. However, the `julia` ecosystem is not nearly as mature as `python`, as a lot of its packages are maintained by small communities, and some time they lead to down some dead ends.

Nonetheless, `julia` is a fun language to play with. `julia` often offers more flexibility and performance than `python`, and its ecosystem has a lot of interesting research codes which are often not found in other ecosystem. So in this lab, we are going to go through

Outline

Foreword	1
Key Concepts	3
Julia has a JIT compiler	3
Julia has multiple dispatch	3
Julia has a package manager	3
Basic Syntax	3
Variables	3
Writing Structs	3
Functions	3
Control Flow	4
Exercise: Writing an insertion sort algorithm again	4
Step 1: Clone the repository	4
Step 2: Implement the sorting algorithm	4
Step 3: Test the algorithm	4
Packaging code	4
Step 0: Looking at some examples	5
Step 1: Create the main module	5
Step 2: Create a sub-module	5
Building documentation	5
Documenter.jl	5
Writing tests	5
Add Test to the dependency	5
Writing test suites	5
Step 1: Write tests for the insertion sort algorithm	5
Step 2: Group tests together with <code>@testset</code>	5
Running tests	5
Best practices	5
All roads lead to Rome	5
Type stability	6
Write functions	6

Introduction to Julia

Development tips	6
Use the REPL	6
Use the package manager	6
Working with IDEs	6
Noteworthy libraries	6

Introduction to Julia

Key Concepts

Julia has a JIT compiler

Julia has multiple dispatch

For the people who learn `python` as their first programming language, and perhaps engaged in some projects related to `python`, you may find `julia` quite odd in the sense that **it does not have classes**.

Julia has a package manager

Basic Syntax

Variables

`julia` is similar to `python` in many ways. For example, `a = 1` will define a variable `a` with the value of `1`. As we mentioned, `julia`'s multiple dispatch system relies on the type of the variable to determine which function to call, so it is important to learn how to inspect the type of a particular variable. You can check the type of a variable by using the `typeof` function. If you run `typeof(a)`, it will return `Int64` because `1` is an integer.

To define a list, the basic syntax is similar, that is `a = [1, "hello", 3.14]`. However, in order to get a list from say a range, the syntax is a bit different. Instead of `a = list(range(1, 10))`, you would write `a = collect(1:10)`. There is a bunch of functions related to collection you can checkout [here](#).

From a `python` background, the biggest difference in defining variables in `julia` is in dictionary. In `python`, you would define a dictionary like this `a = {"key": "value"}`, but in `julia`, you would define a dictionary like this `a = Dict{String, String}()`. Accessing an element remains the same as `python`

Similar to `python`, if you have a variable defined by referencing another variable, i.e. `b = a` while `a = [1, 2]`, changing element in `a` will also change `b` automatically.

Writing Structs

While there is no class in `julia`, it is still useful to be able to bundle a bunch of variables together so you can access them easily. This called composite type in `julia`, and we use the `struct` keyword to define them. For example, you can define a struct like this:

```
struct myStruct
    name::String
    somedata::Vector{Int}
end
```

Note `::` syntax is used to specify the type of the variable. In this case, `name` is a `String` and `somedata` is a vector of `Int`. You can create an instance of this struct like this:

```
a = myStruct("hello", [1, 2, 3])
```

Functions

It is fair to say `julia` centers around writing functions, and one of its most interesting features is multiple dispatch. In `julia`, you can define a function like this:

Introduction to Julia

```
function f(x)
    return x + 1
end
```

Note that `julia` does not care about indentation, however it is still a good practice to indent your code properly.

So far we haven't seen much difference between `python` and `julia` in function definition. However, here comes the interesting bit, a function with the same name but with different argument types will be treated as a different function. For example, you can define a function like this:

```
function f(x::Int)
    return x + 1
end
```

And you can define another function with the same name but with a different argument type like this:

```
function f(x::String)
    return x * "!"
end
```

Now if we call `f(1)`, it will return `2`, and if we call `f("hello")`, it will return `hello!`. This is really powerful (and somewhat cursed), since it reduces the coding task to defining your struct and manipulating it with the function with argument that takes that specific type. This basically acts as a replace for method in class in `python`.

Another difference between `julia` and `python` is the way unnamed function (lambda) are defined. We didn't cover this

Control Flow

Exercise: Writing an insertion sort algorithm again

Step 1: Clone the repository

Fork [this repository](#) and clone it to your local machine.

Step 2: Implement the sorting algorithm

Open the `src/insertion_sort.jl` file and implement the insertion sort algorithm within `insertion_sort` function.

Step 3: Test the algorithm

Once you have implemented the body of the algorithm, start the `julia` REPL and run the following command to test the algorithm:

1. Press `]` to enter the package manager mode.
2. Run `activate .` to activate the project.
3. Run `test` to test the algorithm.

Packaging code

Packaging code in `julia` could take a while to get used to. Instead of creating submodules by directories and `__init__.py`, fundamentally in `julia` you just `include("file.jl")` in your main

Introduction to Julia

module. Any subdirectories is just to group those files together. And instead of using syntax like `from scipy.optimize import minimize` to import a function from a submodules, one needs to export functions that are written such that they can be imported by the main module.

Step 0: Looking at some examples

Have a look these three examples: [DifferentialEquations.jl](#), [Flux.jl](#), and [CUDA.jl](#).

Step 1: Create the main module

Step 2: Create a sub-module

Building documentation

`Documenter.jl`

Writing tests

Add Test to the dependency

Start the julia REPL, run `]` to enter the package manager mode, and run `add Test` to add the `Test` package to the project.

Writing test suites

There are two levels of writing tests in `julia`: `@test` and `@testset`. The purpose of `@test` is to test a single function or statement, which is similar to test functions you have learned in the `python` session. On the other hand, you don't want your entire testsuite to exist whenever one function fails. In this case, you can use `@testset` to group tests together and isolate them from the rest of the tests.

Step 1: Write tests for the insertion sort algorithm

Step 2: Group tests together with `@testset`

Running tests

You have already tried running tests in the previous exercise. Once again, the way to run tests is to start the `julia` REPL, enter the package manager mode by pressing `]`, and run `test` to run the tests.

Best practices

All roads lead to Rome

In `python`, the intention of the language is to have only one obvious way to a solution. Although that is often violated and people dunk on their motto, it is still largely true. Creating modules, writing classes, and writing tests, they can all be done in a similar fashion. On the other hand, there are many ways to do the same thing in `julia`. We have seen the three different ways to build your package hierarchy, and the support of metaprogramming in `julia` together with multiple dispatch allows you to come with wild solutions to your problems.

Introduction to Julia

Type stability

Write functions

This is something that took me a while to really understand what do they mean by

Development tips

Use the REPL

Use the package manager

Working with IDEs

Noteworthy libraries