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 interactiveness 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

Introduction to Julia

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

Functions

It is fair to say `julia` centers around writing functions

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 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.

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

Introduction to Julia

Working with IDEs

Noteworthy libraries