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 some of `julia`'s key features, and we will also explore how documentation and packaging work in `julia`.

Outline

Introduction to Julia

Introduction to Julia

Key Concepts

Julia has a JIT compiler

`julia` has a just-in-time(JIT) compiler, which means that the code you write is compiled to machine code on the fly. This allows you to write code that is as fast as C or Fortran, while still having the interactivity of `python`, such as a for-loop. This implies the first time you run a function in `julia` it will compile the function before executing it. This compilation will incur some overhead, but then the subsequent times using the function will just call the cache such that you don't have to pay the overhead again and again. We will dive into this later in the lab.

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**. Instead, `julia` has a killer feature that is called multiple dispatch, meaning the exact function being execute depends on the type of arugments. This gives `julia` developers a lot of freedom in composing different software. We are going to see more detail about why this is an awesome feature later in this session.

Julia has a package manager

One thing that is quite nice about `julia` is it has a tightly integrated package manager. It helps you to manage your dependencies on a project to project level. In a `julia` REPL, you can enter the package manager mode by pressing `]`, and you can add, remove, and update packages in your project. This will also create a `Project.toml` and `Manifest.toml` file in your project directory, which will help you to manage your dependencies.

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 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("key" => "value")`. Accessing an element remains the same as `python`

Simliar 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.

Introduction to Julia

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:

```
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 sepcific type. This basically acts as a replace for method in class in `python`.

Control Flow

The syntax for defining control flow in `julia` is similar to `python`. The major difference between `python` and `julia` in syntax is since `julia` does not care about indentation, you need to use `end` to close the block of code. For example, you can define a for-loop like this:

```
for i in 1:10
    println(i)
end
```

`while` loop and `if-else` statements are also similar to `python`, and you can find more detail here

What I want to highlight is `julia`'s JIT capability. Let's write a function that takes a long time to run:

```
function add100000000(x)
    for i in 1:100000000
        x += 1
    end
    return x
end
```

You can try to run this counter part in `python` and you will see that `julia` is much faster. And we can even dig into the internal of the function by using `@code_llvm` to see the LLVM code generated by the function. In the `julia` REPL, you can run `@code_llvm add100000000(1)` to see the LLVM code generated by the function, and you will see that the cached LLVM intermediate representation (IR) code actually already have the number of iterations in the code. This gives us hint that `julia` is capable to optimize the code and run it more efficient than `python`.

Exercise: Writing an insertion sort algorithm again

Step 0: Download Julia

Follow the instruction on here to install `julia` on your machine.

Step 1: Clone the class 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: Moving main function to another file

Let's start our packaging process by moving the main functions in our file to another file. In the `src` folder, create a file called `utils.jl`. Move every functions in the `InsertionSort.jl` file to the `utils.jl` file. After moving the functions to the `utils.jl` file, make sure you add `export` statement to the functions you want to export. For example, you can add the following code to the `insertion_sort` function:

```
export insertion_sort!
```

This will allow the function to be discovered by the main module.

Step 2: Creating the main module

In your `src` folder, create a file called `InsertionSort.jl`. This file will be the main module of the package. Note that this file will not contain any actual function. Instead, it will mostly contain `include` statements to include the sub-modules.

In the `InsertionSort.jl` file, write the following code:

```
module InsertionSort
include("insertion_sort.jl")
end
```

Step 3: Adding the package to the project

When you are developing a package in `julia`, you want to observe the changes you make as you go. To do this, you need to add the package to the project in development mode. To do this, start the `julia` REPL, enter the package manager mode by pressing `]`, and run `dev .` to add the package to the project in development mode. This will allow your environment to see the changes you make to the package in real time. Once you are done with development, you can run `free .` to remove the package from development mode.

Once you have added the package to the project in development mode, you can run `using InsertionSort` to use the package in the project. Enter the `julia` REPL and run `using InsertionSort` to check whether the package is working correctly.

Building documentation

In almost all `julia` packages, you will find documentation page that looks similar to each other. This is because `julia` has a package called `Documenter.jl` that the community uses to generate their documentation. In this section, we are going to learn how to build a documentation page for your package using `Documenter.jl`.

Step 1: Setting up directory

The first step is to create a directory called `docs` in the root of your project. Inside the `docs` directory, create a file called `make.jl`. This file will contain the configuration for the documentation. Also in the `docs` directory, create a directory called `src`. This directory will contain the markdown files for the documentation. Put a markdown file called `index.md` in the `src` directory. This file will be the main page of the documentation.

In the `make.jl` file, write the following code:

```
using Documenter, MyPackage
makedocs(sitename="MyPackage.jl")
```

Step 2: Building the documentation

To build the documentation, start the `julia` REPL, enter the package manager mode by pressing `]`, and run `activate .` to activate the project. Then run `using Documenter` and `include("docs/make.jl")` to build the documentation.

After building the documentation, you will see a `build` directory in the `docs` directory. Inside the `build` directory, you will find an `html` file. Just to check the result immediately, open this file in a browser to see the documentation.

Step 3: Writing doc strings

The next thing we want to do is to write doc strings for the functions in the package. Doc strings are written in markdown format and are placed right above the function definition. For example, you can write a doc string like this:

```
"""
Insertion sort algorithm.

This function sorts an array using the insertion sort algorithm.

# Arguments
- `arr::Vector{Int}`: The array to be sorted.

# Examples
julia
arr = [3, 2, 1]
insertion_sort!(arr)
println(arr)  \# [1, 2, 3]
"""
function insertion_sort!(arr::Vector{Int})
    # implementation
end
```

This is different from `python` in the sense that in `python` you would write the doc string in the function body, while in `julia` you write it right above the function definition. Make sure this is structured correctly, otherwise you will not be able to generate the documentation correctly.

Step 4: Adding automatic API documentation

The next step is to add automatic API documentation to the documentation page. To do this, let's add another file called `api.md` in the `src` directory. Add the following code to the `api.md` file:

````
```markdown
# API

'''@autodocs
Modules = [InsertionSort]
'''
```
````

Now you can build the documentation again to see the new API documentation page. By default, every markdown file within the `src` directory will be included in the documentation with pretty formatting.

If you just open the `html` file, you may have some buggy issue related to navigation such as wrong navigation. Instead, you can either install `LiveServer` then start a server with `julia -e 'using LiveServer; serve(dir="docs/build")`. In this session, we will just start a http server with python by running `-m http.server --bind localhost --directory ./build`.

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

The way to test a function in `julia` isusing the test macro `@test`. For example, you can write a test like this:

```
using Test

@test insertion_sort!([3, 2, 1]) == [1, 2, 3]
```

Step 2: Group tests together with @testset

Similar to `python`, you can group tests together to let certain tests fail and still finish other tests. To do this in `julia`, we can use the `@testset` macro. For example, you can write a test set like this:

```
@testset "Insertion sort tests" begin
    @test insertion_sort!([3, 2, 1]) == [1, 2, 3]
    @test insertion_sort!([3, 2, 1, 4]) == [1, 2, 3, 4]
end
```

Now if you start the REPL and go into package manager mode, you can run `test` to run the tests. You should see a summary of the tests and whether they passed or failed.

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. In general, I find the development experience in `julia` to be pretty liberal.

Type stability

`julia` relies LLVM to compile the code, and knowing the type of the variable can help the compiler specialize the code to make it more performant. Let's look an example from here about type instability:

Suppose you define a function

```
x = 5.0

function f()
            s = 0
            for val in x
              s = s + val
            end
            return s
       end
```

since x could be any type, the compiler does not know how to specialize the code, meaning it needs to use a more general code that is slower. There is a built-in macro in `julia` called `@code_warntype` that can help you to check the type stability of your code. If you run `@code_warntype f()`, you will see that some part of the input is highlighted in red, meaning that the compiler is not sure about the type of the variable.

Now we define the exact same function but giving it an argument

```
function g(x)
           s = zero(eltype(x))
           for val in x
              s = s + val
           end
           return s
       end
```

Now when you run `@code_warntype g([1.0, 2.0, 3.0])`, since the compiler is given a concrete variable that the type is known, the code is specialized and the red highlight is gone.

Let's benchmark the two functions by running `@btime f()` and `@btime g([1.0, 2.0, 3.0])`. You will see that the second function is much faster than the first one.

Development tips

Use Revise

`julia` has a package called `Revise.jl` that allows you to reload your code without restarting the REPL. Generally if you want to redefine a function, you need to quit hte REPL and enter it again. This could cause you to lose some intermediate variables you may want to keep between session. Instead, you can make your package available to Revise, then any changes you made in the files will be reflected in the REPL. To do this, first add the package you are developing by running `dev .` in the package manager mode. Then run `using Revise` in the REPL to start the Revise package.

Don't be afraid of for loop and other low level operation

As you have seen in the control flow section, `julia` JIT compiler allows you to implement things like a for-loop without worrying about its performance implication. In `python`, the typical workflow is you try to break down a task into a bunch of functions that someone has implemented a high performance c function underthehood, therefore it is always good to use libraries like `numpy` and `scipy`. However, once in a while you will stumble upon a problem that there is no obvious package to call, and it could be a stuck in `python`. People then turn to tools like `cython` to improve the performance of the performence critical bit of their code. `julia` is try to solve this "two languages problems", which is if you want high performance `python`, sometime you also need to know low level programming languages like `c`. Instead, in `julia` you never need to do that, everything is in `julia`. You are allowed to write for loops and other patterns that may not be encouraged in `python`. This liberates the developers from concerns of design pattern, instead we can focus on really delivering the algoritm.

Don't be afraid to look into someone's source code.

Another merit of brough by `julia` one language system is you don't need to worry about hitting some other language when trying to dig through someone's code. Again, everything is in `julia`, and what even nicer about this is `julia` is supposed to be composable. Let say you have some function that works on a generic array type

```
function myfunc(arr::AbstractArray)
    return sum(arr)
end
```

If you want it to work on accelerator such as GPU, it might just work for you because `CuArray` is a subtype of `AbstractArray`, meaning you have to do nothing to make it work on GPU.

Noteworthy libraries

The SciML ecosystem

The SciML ecosystem is a collection of packages that are designed to solve scientific machine learning problems. They offers a wide range of tools for writing simulations, solving differential equations, optimization, uncertainty quantification, and more. It is great to play with if you are interested in scientific computing.

Flux and Lux

Flux and Lux are the big two machine learning libraries in `julia`. If you want to build more standard machine learning models, `Flux` should be a better choice. If you want something better integrated with solvers and scientific computing, `Lux` is a better choice.