Northwestern
INFORMATION TECHNOLOGY

Julia

Sophie van Genderen, Assoc Computational Specialist
Julia Giannini, Computational Specialist

Northwestern IT Research Computing and Data Services

November 21th 2024

Julia

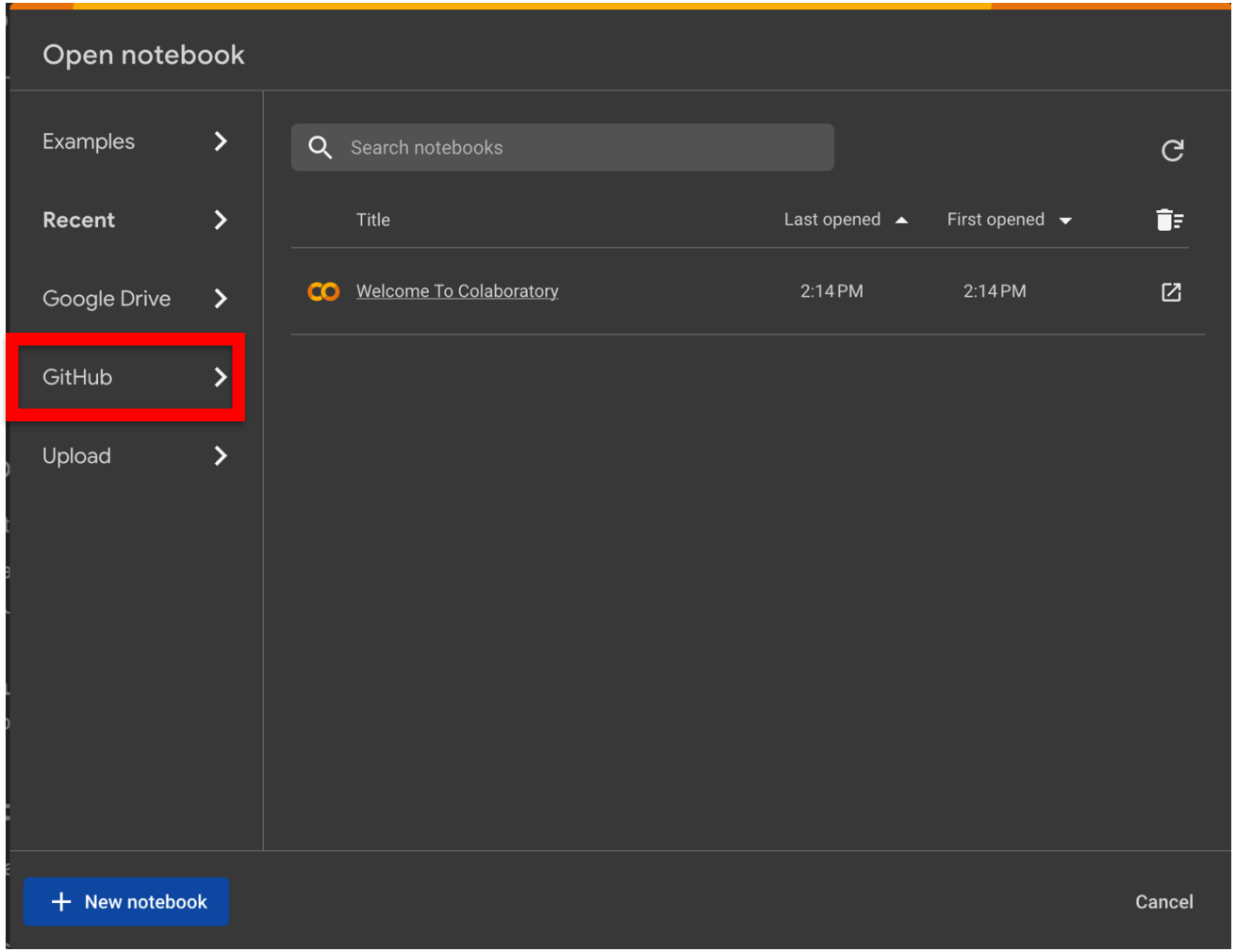
- **Installing a Julia Kernel on Google Colab**
- Why Julia?
- Comparing Python to Julia
- Demo
- Poll

Setting Up a Julia Kernel in Google Colab

Step by step instructions

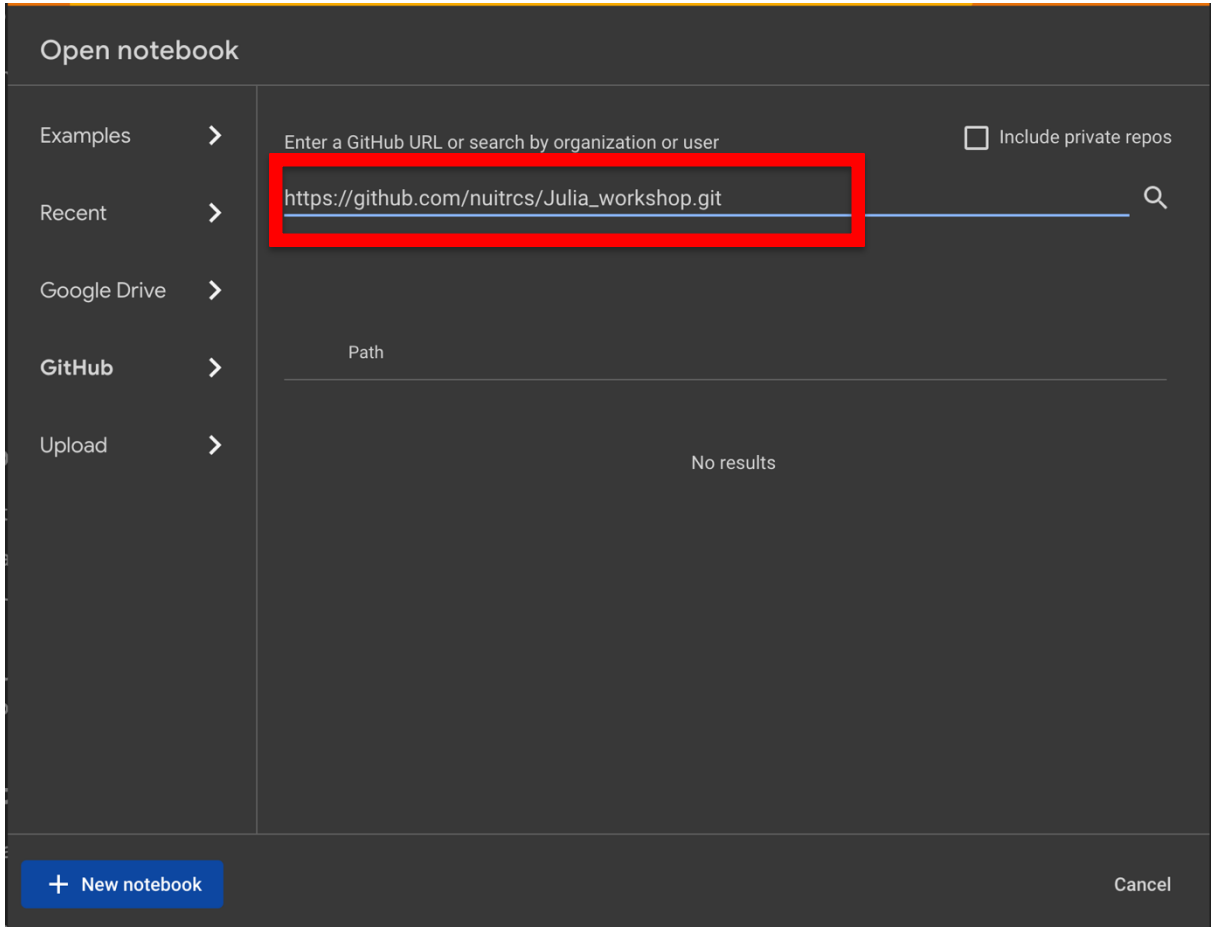
- Go to https://colab.research.google.com
- Log-in with a google account (probably your Northwestern GSuite)

Setting Up a Julia Kernel in Google Colab



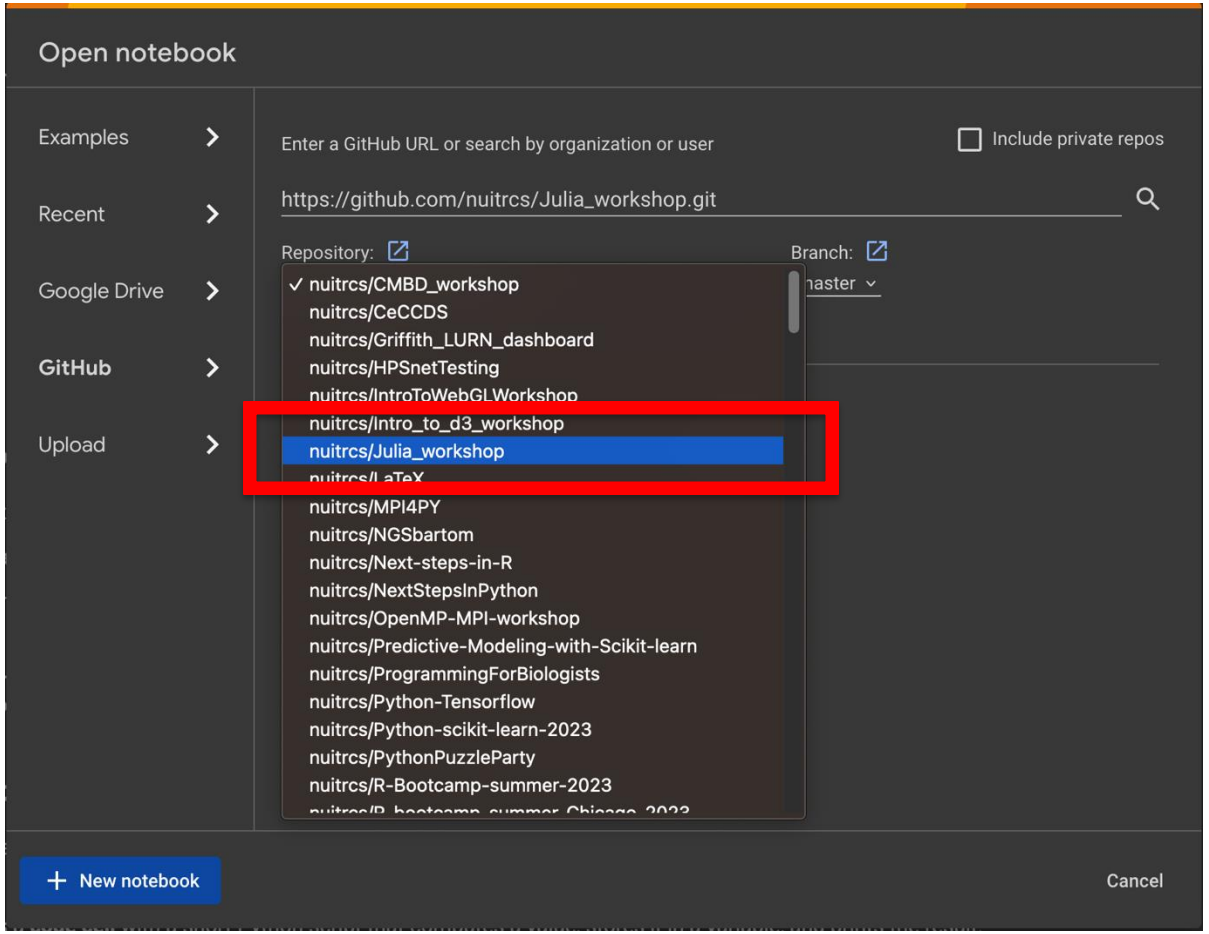
Setting Up a Julia Kernel in Google Colab

- Please enter the following URL: https://github.com/nuitrcs/Julia_workshop
- Then hit “Enter”



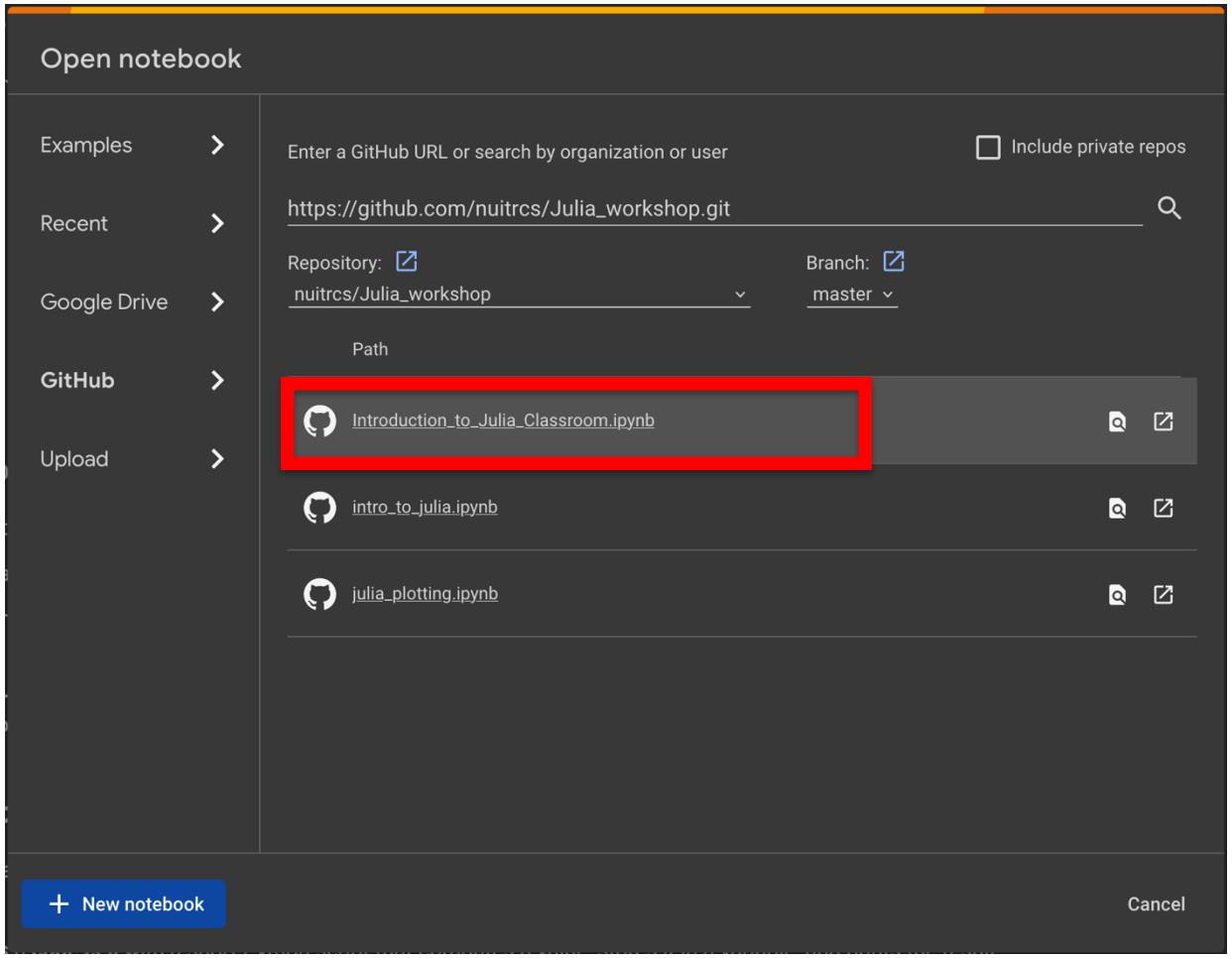
Setting Up a Julia Kernel in Google Colab

- Under repository, find ***nuitrcs/Julia_workshop***
- Under branch, select ***master***



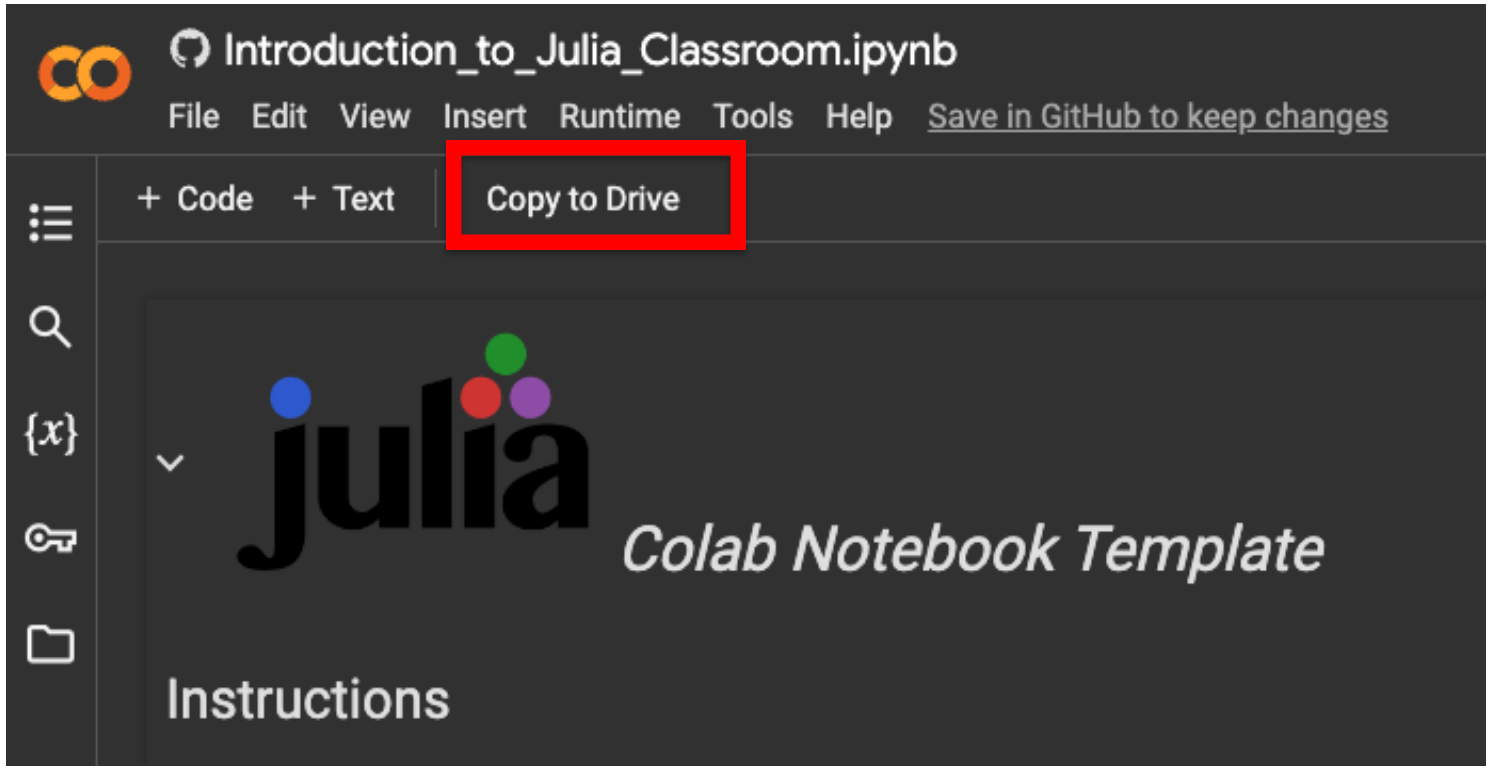
Setting Up a Julia Kernel in Google Colab

- Next, select ***Introduction_to_Julia_Classroom.ipynb***



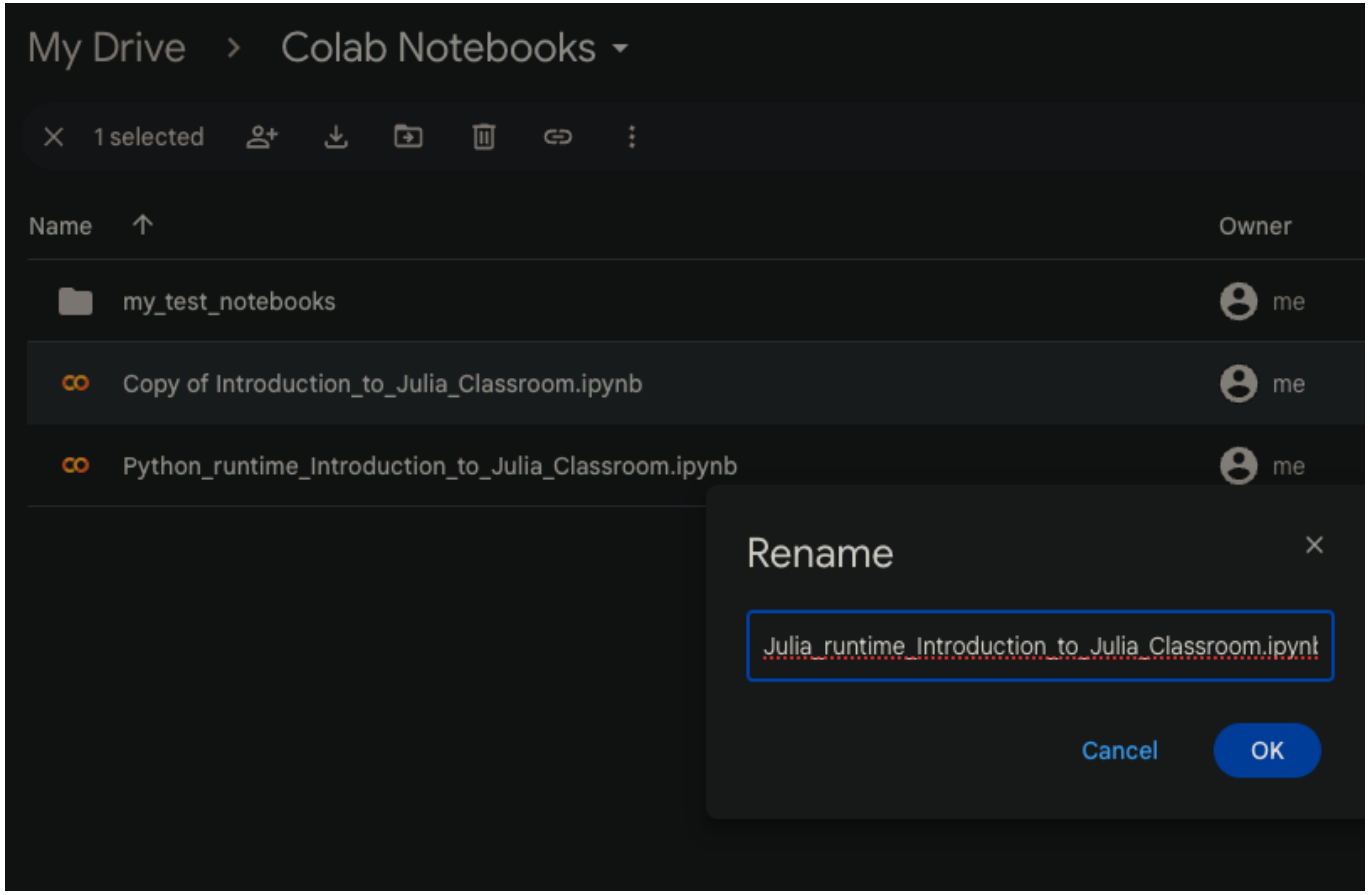
Setting Up a Julia Kernel in Google Colab

- In the notebook, select ***Copy to Drive***

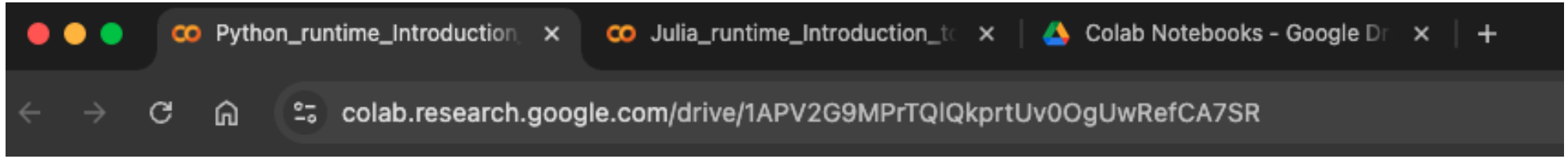


Setting Up a Julia Kernel in Google Colab

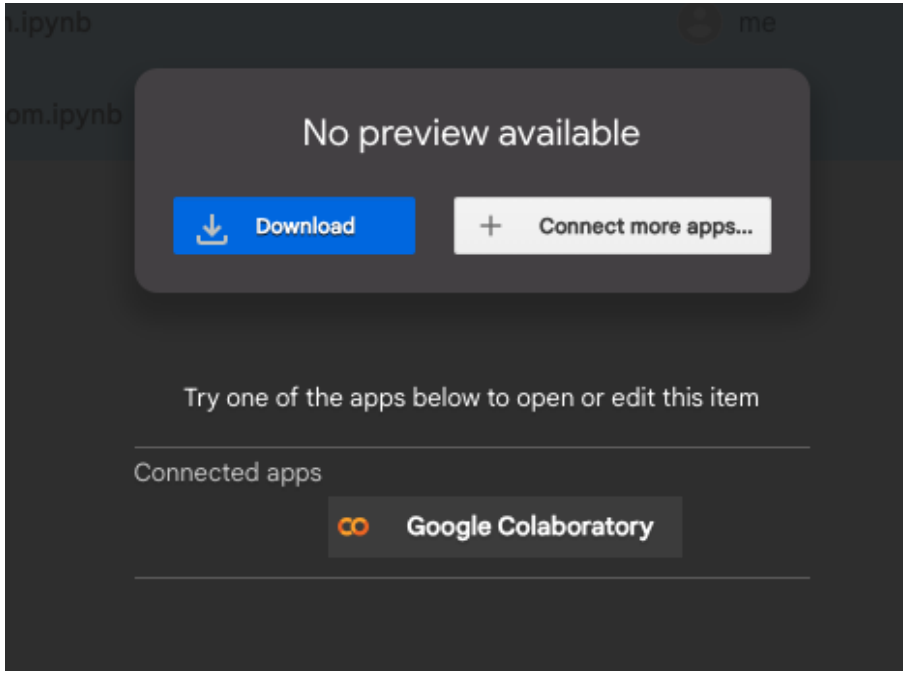
- In your ***Google Drive***, under ***Colab Notebooks***, you can make *two copies* of the notebook ***Introduction_to_Julia_Classroom.ipynb***:
  - ***Python_runtime_Introduction_to_Julia_Classroom.ipynb***
    - (required)
  - ***Julia_runtime_Introduction_to_Julia_Classroom.ipynb***
    - (optional)
- Open both notebooks in Colab



Setting Up a Julia Kernel in Google Colab

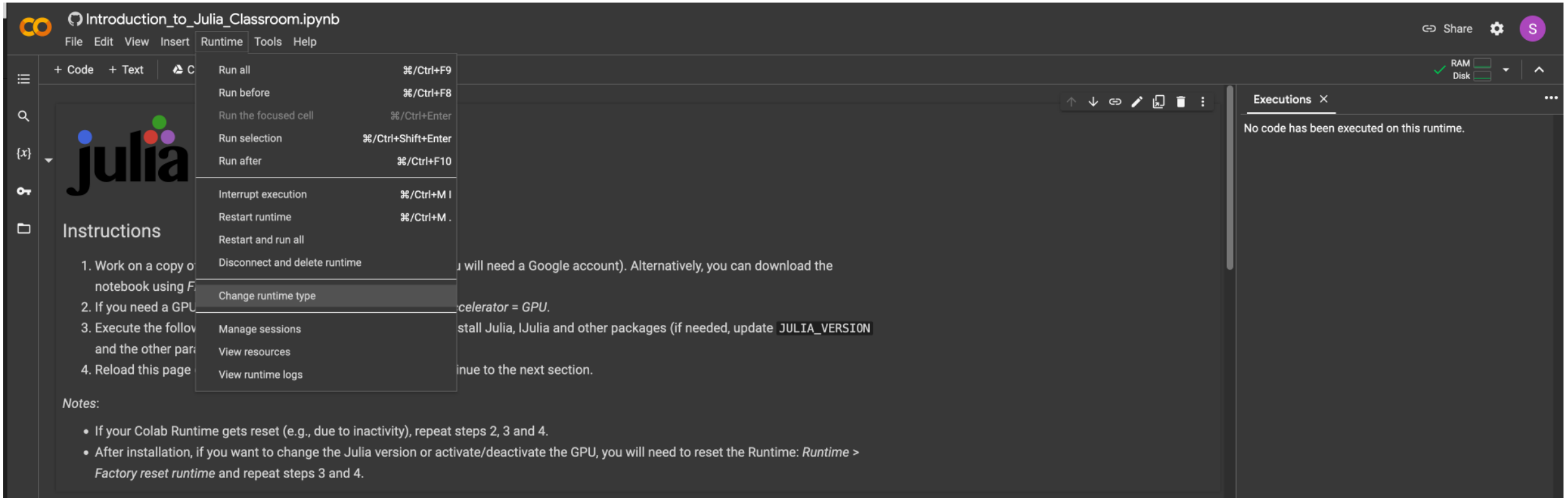


- You would now have a tab open for each copy of the notebook (***Julia_runtime....ipynb*** and ***Python_runtime....ipynb***)
  - You can close other Colab tabs if you like (such as the original ipynb from GitHub)
  - If you are opening the notebooks from Google Drive do so by selecting ***Google Colaboratory***



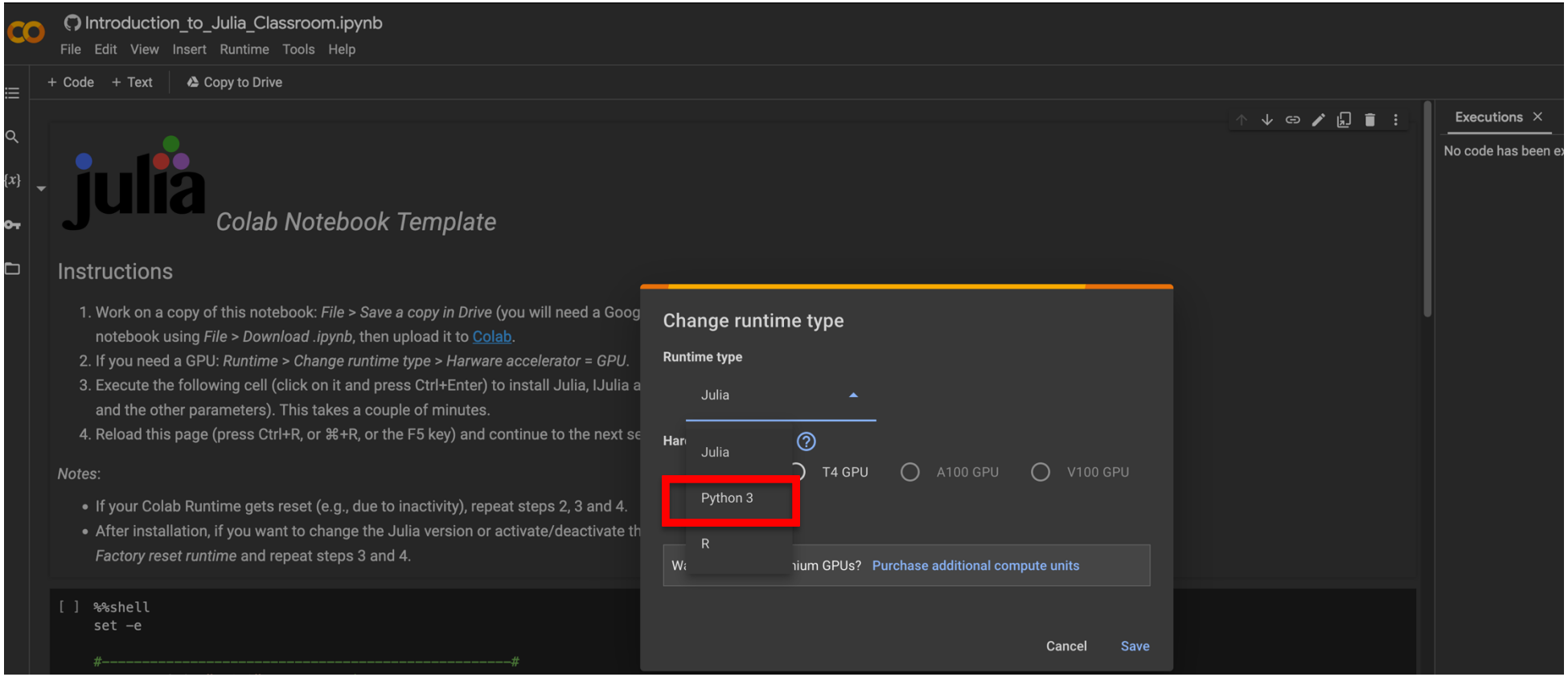
Setting Up a Julia Kernel in Google Collab

In your **Julia runtime notebook** select *Runtime* ➔ *Change runtime type*



Setting Up a Julia Kernel in Google Collab

In your **Julia runtime notebook** change the Runtime Type from Julia to Python3 (or if it is already Python3, leave it as is).



Setting Up a Julia Kernel in Google Collab

- Finally in your **Julia runtime notebook** run the first cell which is install Julia *1.10.4* and a Julia kernel,
  - It may take 10-15 minutes
- *After the installation completes, refresh the tab*

Notes:

- If your Colab Runtime gets reset (e.g., due to inactivity), repeat steps 2, 3 and 4.
- After installation, if you want to change the Julia version or activate/deactivate the GPU, you will need to reset the Runtime: *Runtime > Factory reset runtime* and repeat steps 3 and 4.

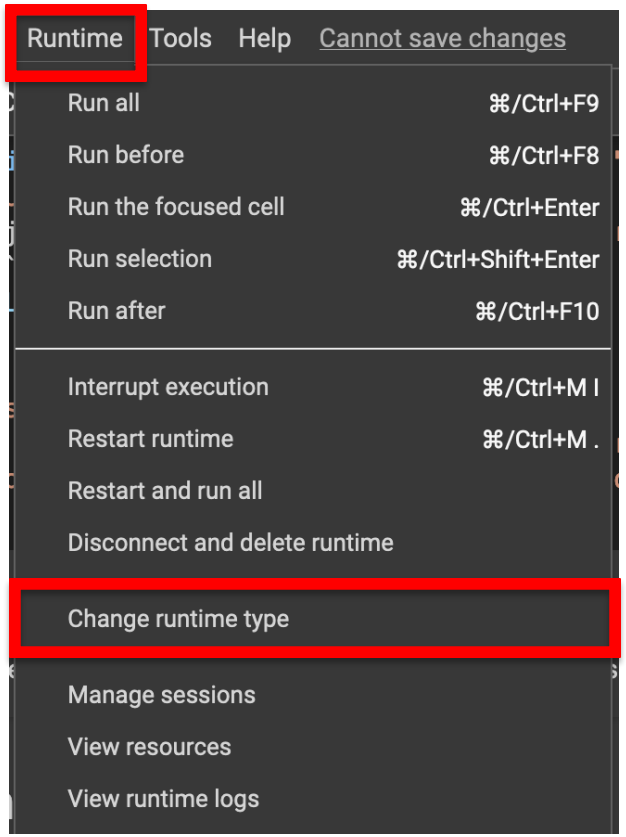
```
[ ] %%shell
    set -e

    #----------------------------------------------------#
    JULIA_VERSION="1.9.3" # any version ≥ 0.7.0
    JULIA_PACKAGES="IJulia BenchmarkTools"
    JULIA_PACKAGES_IF_GPU="CUDA" # or CuArrays for older Julia versions
    JULIA_NUM_THREADS=2
    #----------------------------------------------------#

    if [ -z `which julia` ]; then
      # Install Julia
      JULIA_VER=`cut -d '.' -f -2 <<< "$JULIA_VERSION"`
      echo "Installing Julia $JULIA_VERSION on the current Colab Runtime..."
      BASE_URL="https://julialang-s3.julialang.org/bin/linux/x64"
      URL="$BASE_URL/$JULIA_VER/julia-$JULIA_VERSION-linux-x86_64.tar.gz"
      wget -nv $URL -O /tmp/julia.tar.gz # -nv means "not verbose"
      tar -x -f /tmp/julia.tar.gz -C /usr/local --strip-components 1
      rm /tmp/julia.tar.gz

      # Install Packages
      nvidia-smi -L &> /dev/null && export GPU=1 || export GPU=0
      if [ $GPU -eq 1 ]; then
        JULIA_PACKAGES="$JULIA_PACKAGES $JULIA_PACKAGES_IF_GPU"
      fi
      for PKG in `echo $JULIA_PACKAGES`; do
        echo "Installing Julia package $PKG..."
        julia -e 'using Pkg; pkg"add '$PKG'; precompile;"' &> /dev/null
      done

      # Install kernel and rename it to "julia"
      echo "Installing IJulia kernel..."
      julia -e 'using IJulia; IJulia.installkernel("julia", env=Dict(
          "JULIA_NUM_THREADS"=>"'"$JULIA_NUM_THREADS"'"))'
      KERNEL_DIR=`julia -e "using IJulia; print(IJulia.kerneldir())"`
      KERNEL_NAME=`ls -d "$KERNEL_DIR"/julia*`
      mv -f $KERNEL_NAME "$KERNEL_DIR"/julia

      echo ''
      echo "Successfully installed `julia -v`!"
      echo "Please reload this page (press Ctrl+R, ⌘+R, or the F5 key) then"
      echo "jump to the 'Checking the Installation' section."
    fi
```

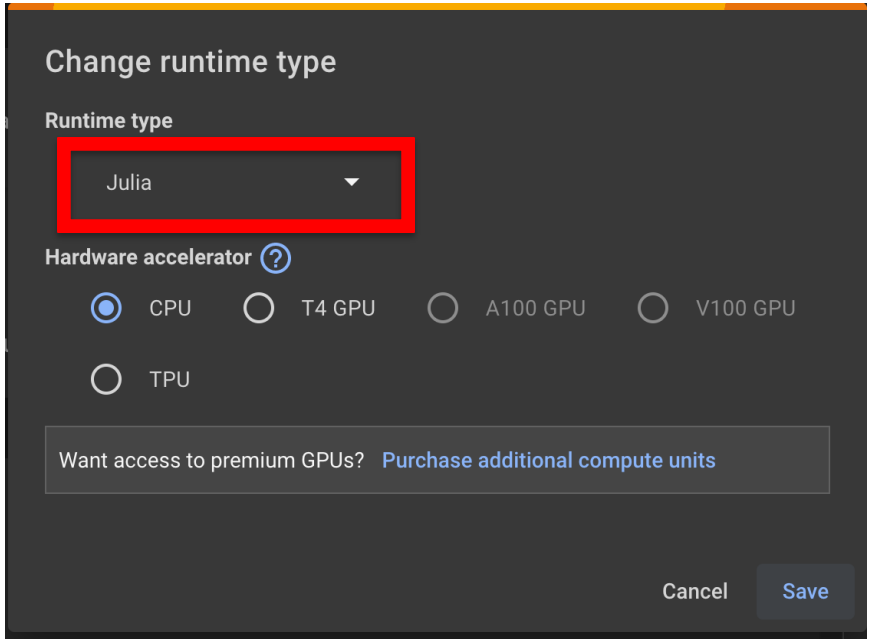
```
Installing Julia 1.9.3 on the current Colab Runtime...
2023-11-13 20:21:02 URL:https://storage.googleapis.com/julialang2/bin/linux/x64/1.9/julia-1.9.3-linux-x86_64.tar.gz [146268149/146268149] -> "/tmp/julia.tar.gz" [1]
Installing Julia package IJulia...
Installing Julia package BenchmarkTools...
Installing IJulia kernel...
[ Info: Installing julia kernelspec in /root/.local/share/jupyter/kernels/julia-1.9

Successfully installed julia version 1.9.3!
Please reload this page (press Ctrl+R, ⌘+R, or the F5 key) then
jump to the 'Checking the Installation' section.
```

Checking the Installation

Setting Up a Julia Kernel in Google Collab

After refreshing the page, in your **Julia runtime notebook** check to make sure that the `runtime type` is now Julia





Setting Up a Julia Kernel in Google Collab

After refreshing the page, in your **Julia runtime notebook** please run the cell which should print the `version` information about Julia.

Checking the Installation

The `versioninfo()` function should print your Julia version and some other info about the system:

```julia
versioninfo()
```

```
Julia Version 1.9.3
Commit bed2cd540a1 (2023-08-24 14:43 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 2 × Intel(R) Xeon(R) CPU @ 2.20GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-14.0.6 (ORCJIT, broadwell)
  Threads: 3 on 2 virtual cores
Environment:
  LD_LIBRARY_PATH = /usr/local/nvidia/lib:/usr/local/nvidia/lib64
  JULIA_NUM_THREADS = 2
```

Julia

- Installing a Julia Kernel on Google Colab
- **Why Julia?**
- Comparing Python to Julia
- Demo
- Poll

Why Julia?

- Superior performance for numerical analysis and scientific computing because of “Just-In-Time” (JIT) compilation vs. Interpreted
  - Compiled vs. Interpreted languages
- Built-in parallelism, great for heavy computations
- No external libraries needed for Mathematics
- Data Models

Julia

- Installing a Julia Kernel on Google Colab
- Why Julia?
- **Comparing Python to Julia**
- Demo
- Poll

Comparing Python to Julia

Key Indicator	Julia	Python
Maturity	Created in 2012	Created in 1991
Scope	General-purpose, but data-oriented	General-purpose and used for almost everything
Language Type	High Level, (Just in Time) Compiled	High Level, Interpreted
Typing	Dynamically-typed language, but also offers the ability to specify types (Static)	Dynamic, the type for a variable is decided at runtime
Open-source	Yes	Yes
Usage	Data Science and Machine Learning – especially work with data models	Mobile/web Dev, AI, Data Science, web scripting, game development, security ops.
Data Science	Math functions are easy to write and understand – no external libraries are needed for math functions	Requires NumPy or other libraries for advanced math
Performance	Fast development and production, high speed runtime, can handle millions of data threads	Fast for development, slow for production

Comparing Python to Julia

- Indexing
- Loops
- Functions / Methods
- Installing / Loading / Using external packages
- Dictionaries
- Global
- Modules
- Classes

Comparing Python to Julia

- **Indexing**
- Loops
- Functions / Methods
- Installing / Loading / Using external packages
- Dictionaries
- Global
- Modules
- Classes

Comparing Python to Julia - **Indexing**

- Julia: 1, 2, 3 (indexing is 1-based)
- Python: 0, 1, 2 (indexing is 0-based)

Python

```
list_numbers = [1,
2, 5, 10, 4, 9, 7, 12, 9]
element = 10
list_numbers.index(element)

> 3
```

Julia

```
str = "Hello, world!"
println(str[1])
println(str[4:9])

> H
> lo, wo
```

Comparing Python to Julia

- Indexing
- **Loops**
- Functions / Methods
- Installing / Loading / Using external packages
- Dictionaries
- Global
- Modules
- Classes

Comparing Python to Julia - **Loops**

Python

Loops

```
while
for

(if/else)
```

Control Statements

```
continue
break

(pass)
```

Julia

Loops

```
while
for

(foreach)
```

Control Statements

```
continue
break
```

Comparing Python to Julia - **Loops**

Python

```
for i in range(10):
        i += 5
        print(i)
```

```
for i in range(5):
    if i % 2 == 0:
        print(f"{i} is even")
    else:
        pass  # Do nothing for
odd numbers
```

Julia

```
for i in 1:10
        i += 5
        print(i)
end
```

```
foreach(i -> begin
    if i % 2 == 0
        println("$i is
even")
    end
end, 0:4)
```

https://docs.julialang.org/en/v1/manual/control-flow/#man-loops

Comparing Python to Julia

- Indexing
- Loops
- **Functions / Methods**
- Installing / Loading / Using external packages
- Dictionaries
- Global
- Modules
- Classes

Comparing Python to Julia - **Functions**

Python

```python
def new_function():
     print("Hello!")

new_function()

def name_function(name):
     print("Hello, " + name +
"!")

name_function("James")
```

Julia

```julia
new_function(x,y)
     x + y
end

new_function(2,3)
```

==

```julia
f(x,y) = x + y

f(2,3)
```

https://docs.julialang.org/en/v1/manual/functions/

Comparing Python to Julia - **Methods**

Python

A method in Python is a function that belongs to an object/class.

```
class C:
    def my_method(self):
        print("I am a C")

c = C()
c.my_method()  # Prints("I am a C")
```

Julia

A method in Julia is the implementation of **multiple dispatch**, which allows the call/execution of different definitions of a function based on the data types of the arguments.

(example in Google Colab)

https://docs.julialang.org/en/v1/manual/methods/

Comparing Python to Julia

- Indexing
- Loops
- Functions / Methods
- **Installing / Loading / Using external packages**
- Dictionaries
- Global
- Modules
- Classes

Comparing Python to Julia - **Packages**

Julia:

<u>To install a package:</u>

In Julia command line, type "]" and then

```
add <Package_name>
Pkg> add JSON StaticArrays
```

<u>To Remove a package:</u>

```
Pkg> rm <Package>
```

<u>To get out of Pkg>, use Ctrl+C</u>

<u>To load installed packages:</u>

```
import <Package>
using <Package>
```

Alternatively, in a script (or on the Julia command line)

```
using Pkg
Pkg.add("<package_name1>")

## For multiple packages
Pkg.add(["<package_name1>","<package_name2>
"])

using <package_name1>
```

Comparing Python to Julia - **Packages**

Python

To install a package:

```
Pip/pip3/mamba install <package_name>
```

To remove a package:

```
Pip/pip3/mamba uninstall <package_name>
```

To load installed packages:

```
import numpy as np
from time import time
```

https://services.northwestern.edu/TDClient/30/Portal/KB/ArticleDet?ID=2064

Comparing Python to Julia – **Pin Packages**

Julia

- Pinned packages will never be updated, they're "frozen" on that version
  - To pin: `Pkg> pin <Package name>`
  - To unpin: `Pkg> free <Package name>`
  - Be mindful of dependencies – when pinning packages, it will install all dependencies needed. Version conflicts may occur because of this. There are ways to address this.

- `.toml` files (these are written to the path `~/.julia/environments/v<version_no.>/` when you install packages)
  - `Manifest.toml`
  - `Project.toml`

- Can install multiple packages from `Project.toml`
  - `julia --project=/path/to/myproject`
  - If `myproject/Project.toml`

Comparing Python to Julia

- Indexing
- Loops
- Functions / Methods
- Installing / Loading / Using external packages
- **Dictionaries**
- Global
- Modules
- Classes

Comparing Python to Julia - **Dictionaries**

Python

Creating an empty dictionary:

```
dict_ex = {}
```

Creating a filled dictionary with different types:

```
dict_ex = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
print(type(dict_ex)) ⟶ class 'dict'
```

`dict()` method to make a dictionary:

```
dict_ex = dict(brand= "Ford",
electric= False, year= 1964, colors
= ["red", "white", "blue"])
```

Referring to values using keys:

```
print(dict_ex["brand"])
> Ford
```

`.keys()` returns the keys through a dict_keys object.
`.values()` returns the values through a dict_values object.
`.items()` returns both the keys and values through a dict_items object.

Comparing Python to Julia - **Dictionaries**

Julia

Creating an Empty dictionary:

```
Dict1 = Dict()
println("Empty Dictionary = ", Dict1)
```

Creating an Untyped Dictionary:

```
Dict2 = Dict("a" => 101, "b" => 102,
"c" => "Hello")
println("\nUntyped Dictionary = ",
Dict2)
```

Creating a Typed Dictionary:

```
Dict3 = Dict{String, Integer}("a" =>
101, "c" => 102)
println("\nTyped Dictionary = ",
Dict3)
```

Accessing dictionary values using keys:

```
println(Dict1["b"])
println(Dict1["c"])
```

Creating a Dictionary with Integer keys:

```
Dict2 = Dict(1 => 10, 2 => 20, 3
=> "Geeks")
println(Dict2[1])
println(Dict2[3])
```

Creating a Dictionary with Symbols:

```
Dict3 = Dict(:a => 1, :b => "one")
println(Dict3[:b])
```

`Keys = keys(Dictionary_name)` Returns all the keys of the dictionary

`Values = values(Dictionary_name)` Returns all the values of the dictionary

https://docs.julialang.org/en/v1/base/collections/

Comparing Python to Julia

- Indexing
- Loops
- Functions / Methods
- Installing / Loading / Using external packages
- Dictionaries
- **Global**
- Modules
- Classes

Comparing Python to Julia – **Global variables**

Python

Any variable defined outside the scope of a function is a global variable.

To create or modify the value of a global variable inside the scope of a function, use the keyword `global`

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

The original value of the global variable `x` is `awesome`. This returns:

```
Python is fantastic
```

Comparing Python to Julia – **Global variables**

Julia

It's complicated, and has to do with the fact that Julia is JIT compiled.

A `global` variable is *reasonable* if it's a constant (`const`). It is *borderline reasonable* if it is a constant type.

https://gist.github.com/flcong/2eba0189d7d3686ea9633a6d14398931
https://docs.julialang.org/en/v1/manual/performance-tips/#Avoid-untyped-global-variables
https://docs.julialang.org/en/v1/manual/variables-and-scoping/#man-typed-globals

```
global x::Int = 1000
global const z::Float64 = rand(1)[1]
## This is nice for the compiler 🙂

function my_func()
    ## stuff
end
```

Will lead to (much) better performance than

```
x = 1000
y = rand(1)[1]
## If one of these changes type
throughout the program, it will be
difficult for the compiler 😫

function my_func()
    ## stuff
end
```

Comparing Python to Julia

- Indexing
- Loops
- Functions / Methods
- Installing / Loading / Using external packages
- Dictionaries
- Global
- **Modules**
- Classes

Comparing Python to Julia – **Modules** (Python)

In the command line:

```
[user@computer ~/py_dir]$ python
./load_fibo.py
0 1 1 2 3 5 8
Done
```

Python caches the compiled version of each module in the `__pycache__` directory

```
[user@computer ~/py_dir]$ ls
fibo.py  load_fibo.py  __pycache__
```

https://docs.julialang.org/en/v1/manual/modules/
https://docs.python.org/3/tutorial/modules.html

Definition of module - **`fibo.py`**

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series
up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

Use the module - **`load_fibo.py`**

```
# load my module
import fibo

# call a function defined in that
module
fibo.fib(10)
print('done')
```

Comparing Python to Julia – **Modules** (Julia)

In the command line:

```
[user@computer ~/julia_dir]$ julia
./load_Fibo.jl
0 1 1 2 3 5 8
done
```

Julia does not cache compiled versions of modules

```
[user@computer ~/julia_dir]$ ls
Fibo.jl  load_Fibo.jl
```

https://docs.julialang.org/en/v1/manual/modules/
https://docs.python.org/3/tutorial/modules.html

Definition of module - **`Fibo.jl`**

```
module Fibo

function fib(n::Int) # write Fibonacci series
    a, b = 0, 1
    while a < n
        print(a, " ")
        a, b = b, a + b
    end
    println()
end

# End of module
end
```

Use the module - **`load_Fibo.jl`**

```
# load my module
include("Fibo.jl")
using .Fibo

# call a function defined in that module
Fibo.fib(10)
println("done")
```

Comparing Python to Julia

- Indexing
- Loops
- Functions / Methods
- Installing / Loading / Using external packages
- Dictionaries
- Global
- Modules
- **Classes**

Comparing Python to Julia - **Classes**

Python

```
class Person:
  def __init__(pyobject, name, age):
    pyobject.name = name
    pyobject.age = age

  def myfunction(x):
    print("Hello my name is " + x.name)

p = Person("Matthew", 46)
p.myfunction()

>> Hello my name is Matthew
```

Julia

Julia has no class!

Seriously though, Julia isn't an object-oriented language. Using structs, generic functions, and constructors you can create something class and object-like but it's a pain.

If an OOP paradigm is what you seek, look elsewhere.

Julia

- Installing a Julia Kernel on Google Colab
- Why Julia?
- Comparing Python to Julia
- **Demo**
- Poll

Live Demo!

Live Demo will Cover…

- Python vs. Julia vs. Python w/ Numba
- Python vs. Julia Linear Regression
- Python vs. Julia DataFrames

Note: we will use the **Python runtime notebook** to compare the performance of Julia and Python – follow along if you like

Julia

- Installing a Julia Kernel on Google Colab
- Why Julia?
- Comparing Python to Julia
- Demo
- **Poll**

Poll

- Link to poll:
  - Slido.com
  - #27269017

- What are you most interested in covering next session?

Thank You!