Workflow

Goals

- access first lab
- review the structure of labs for 101
- define command line interface
- talk about three ways to run code
- walk through first lab together

Accessing Labs

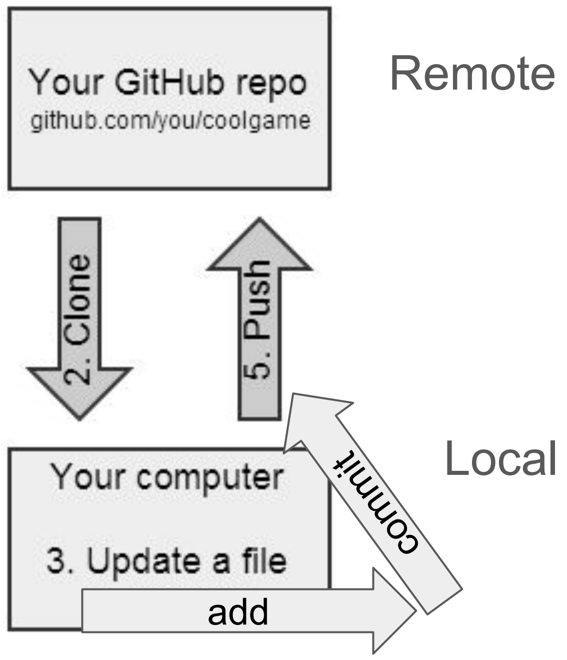
GitHub gives you access

- All the files for labs are initially hosted remotely on GitHub
- Git can be used to transfer the remote files to your local computer!

Dealing with Repos on GitHub (GH)

A repository (or **repo**) that is on GH is a set of files that is stored in **online**.

- **git** is a **local** program (not online) that allows you to copy and update the **remote** files (online)
- copying is done with **clone**
- updating is done with a sequence of three things: **add**, **commit**, **push**

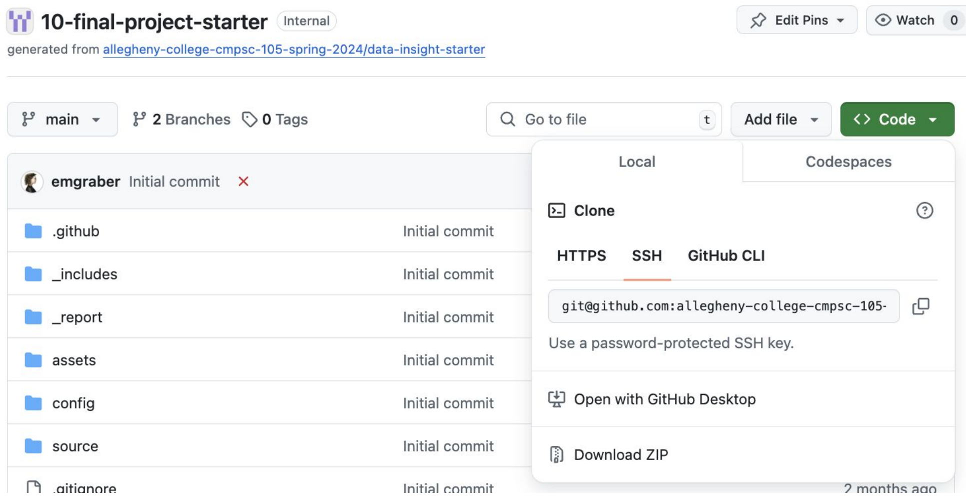


GitHub Classroom Link

In the course assignments schedule, the first lab for this course is linked

Git

- from your repo, click the green code button, and copy the SSH url
- On your computer, open gitbash or a mac terminal
- type
  - >> git clone
  - then paste the copied link
- finally cd into the directory that was just created
- type
  - >> subl .



10-final-project-starter Internal

generated from allegheny-college-cmpsc-105-spring-2024/data-insight-starter

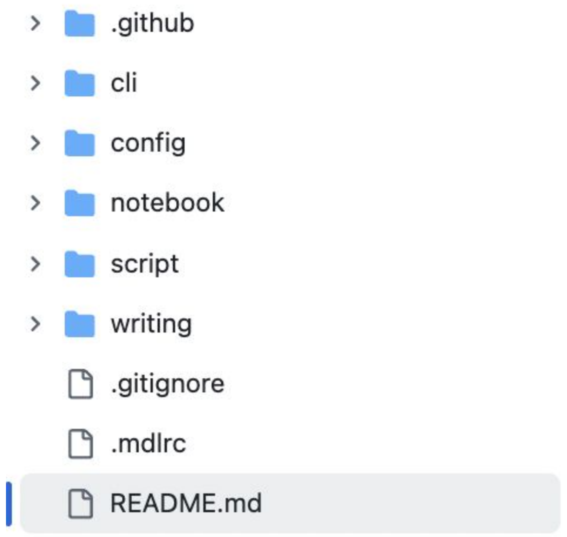
emgraber Initial commit	
.github	Initial commit
_includes	Initial commit
_report	Initial commit
assets	Initial commit
config	Initial commit
source	Initial commit
.gitignore	Initial commit

```
git@github.com:allegheny-college-cmpsc-105-
```

Use a password-protected SSH key.

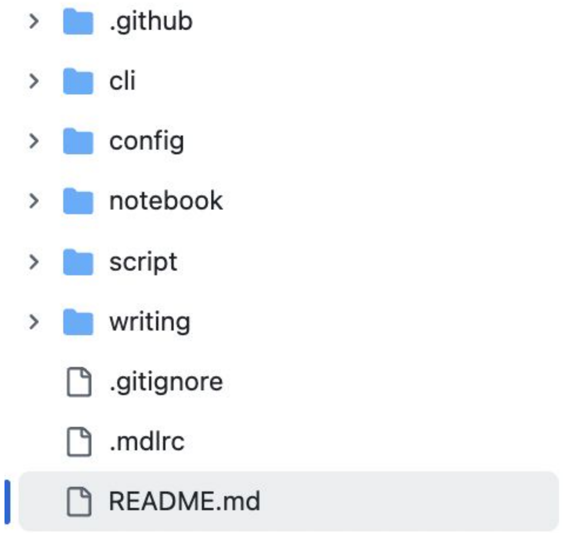
Structure of 101 Labs

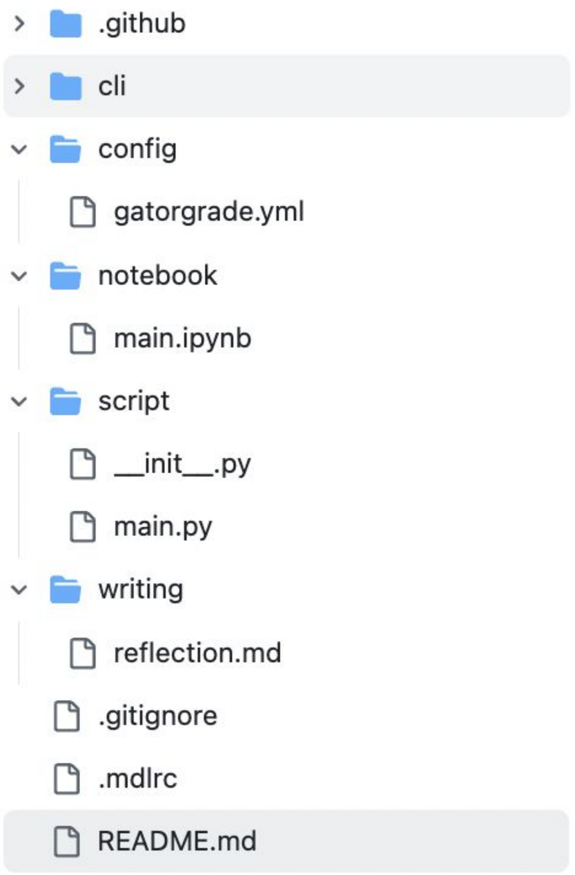
Each lab may have many files and folders!



```
.github
cli
config
notebook
script
writing
.gitignore
.mdlrc
README.md
```

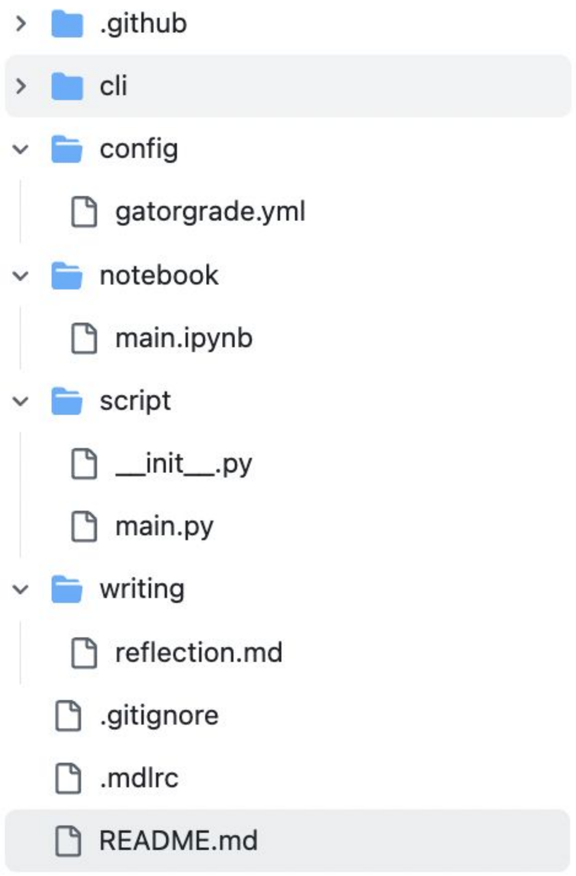
Each folder has other files!


```
> .github
> cli
> config
> notebook
> script
> writing
  .gitignore
  .mdlrc
  README.md
```


```
> .github
> cli
v config
    gatorgrade.yml
v notebook
    main.ipynb
v script
    __init__.py
    main.py
v writing
    reflection.md
  .gitignore
  .mdlrc
  README.md
```

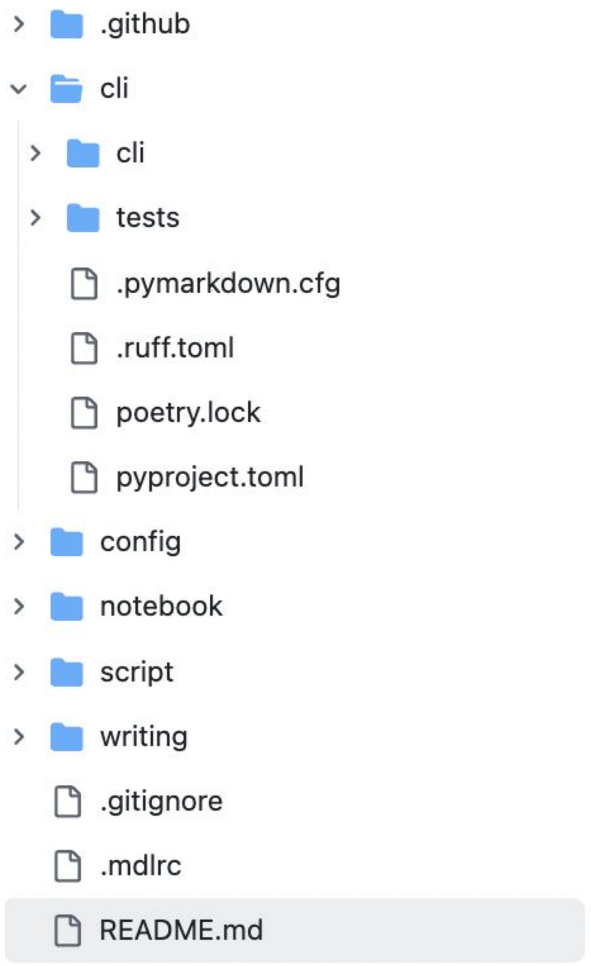
Each folder has other files!

- writing contains the reflection file
- notebook contains a python notebook
  - ipynb ending
- script contains a python script
  - py ending
- config contains the gatorgrade file
  - ylm ending



Each folder has other files!

- **cli** contains another folder called cli
- **cli** contains another folder called tests
- **cli** contains the pyproject.toml

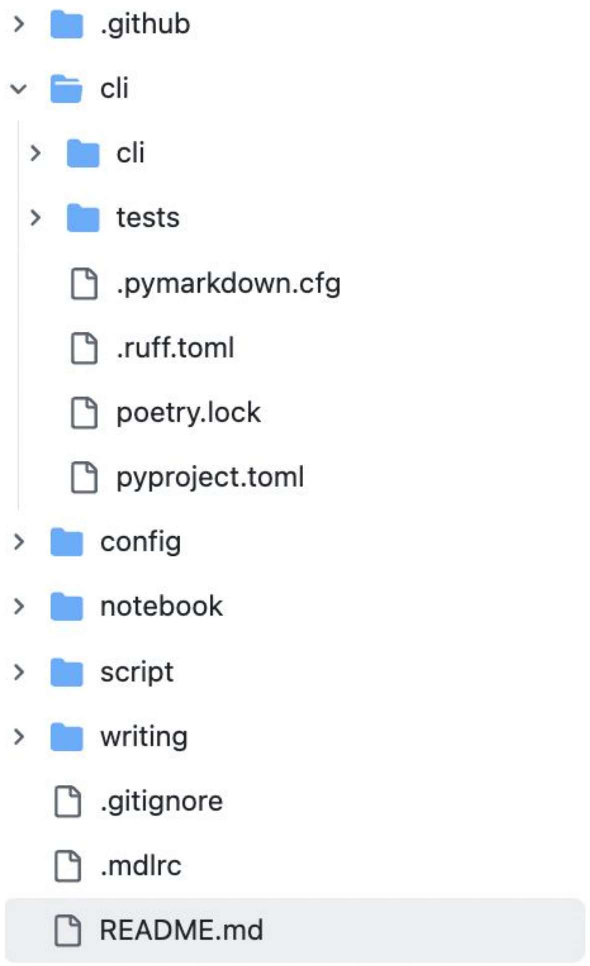


```
> .github
v cli
    > cli
    > tests
      .pymarkdown.cfg
      .ruff.toml
      poetry.lock
      pyproject.toml
> config
> notebook
> script
> writing
  .gitignore
  .mdlrc
  README.md
```

Command Line Interfaces

Command Line Interface

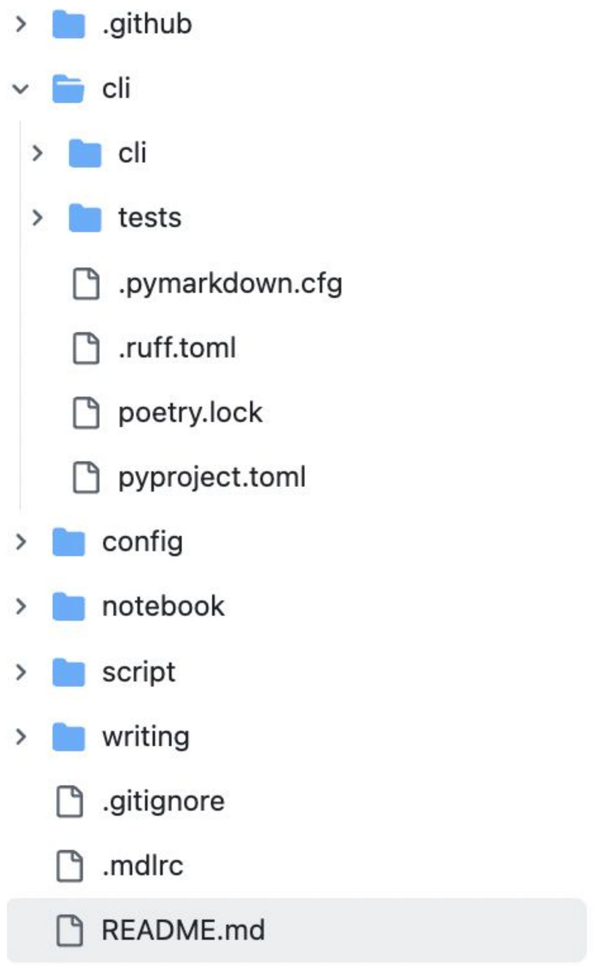
- the files and folders in cli define a COMMAND LINE INTERFACE (**CLI**)
- a **CLI** can run programs in the command line and pass information to the program from the user
- The **interface** of a **CLI** is the command line
- The **interface** is called an interface because it mediates between the user and program



```
> .github
v cli
    > cli
    > tests
      .pymarkdown.cfg
      .ruff.toml
      poetry.lock
      pyproject.toml
> config
> notebook
> script
> writing
  .gitignore
  .mdlrc
  README.md
```

Command Line Interface

- Some CLIs that you have already used during installation process include
  - **brew** install pipx
  - **pipx** install gatorgrade
  - **cd** ~/Documents/path/to/some/other/folders
  - **gatorgrade** --help
  -

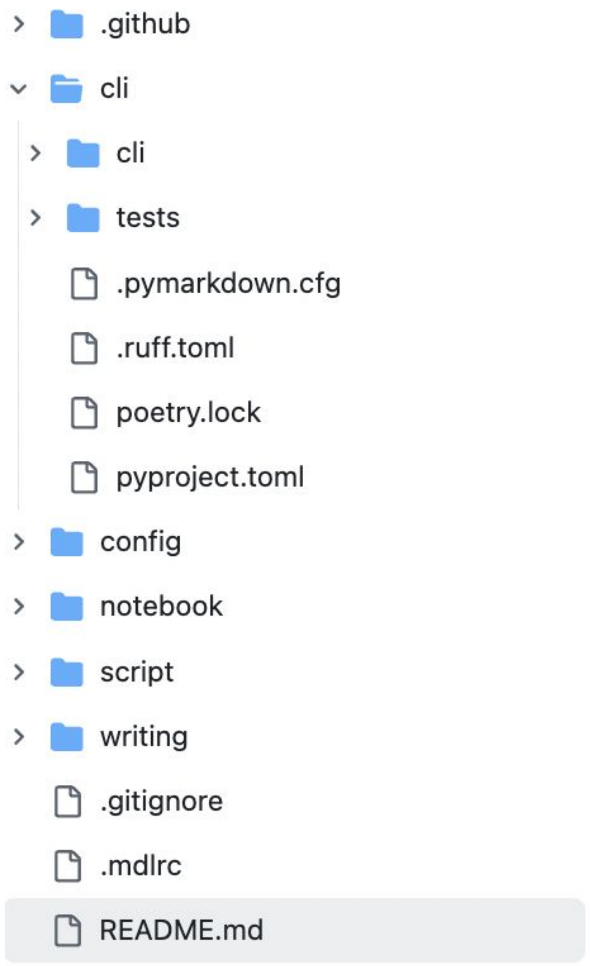


```
> .github
v cli
  > cli
  > tests
    .pymarkdown.cfg
    .ruff.toml
    poetry.lock
    pyproject.toml
> config
> notebook
> script
> writing
  .gitignore
  .mdlrc
  README.md
```

Command Line Interface

- Regular python scripts are not CLIs
- If user input is needed, a regular script has to specifically request the information WHILE the script is RUNNING
- A CLI takes the user input BEFORE the program runs

gatorgrade --help

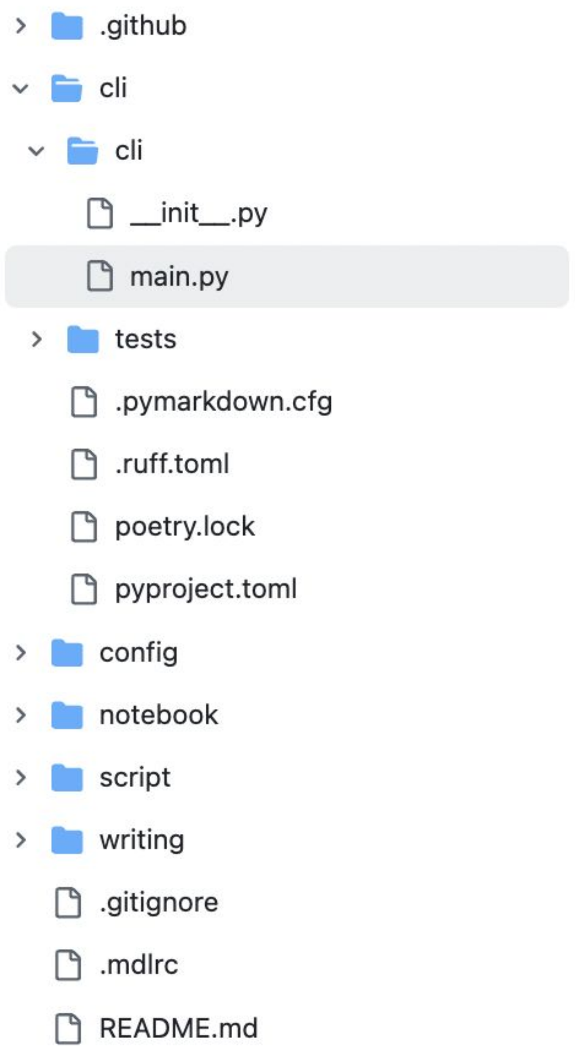


```
> .github
v cli
  > cli
  > tests
    .pymarkdown.cfg
    .ruff.toml
    poetry.lock
    pyproject.toml
> config
> notebook
> script
> writing
  .gitignore
  .mdlrc
  README.md
```

Inside cli

- the main.py that is inside a cli package always tends to have the same structure.

1. import section
2. global variables section
3. algorithm definition section
4. function that defines the cli
    - displays preliminary messages
    - runs predefined algorithms
    - displays results



```
> .github
v cli
  v cli
      __init__.py
      main.py
  > tests
    .pymarkdown.cfg
    .ruff.toml
    poetry.lock
    pyproject.toml
> config
> notebook
> script
> writing
  .gitignore
  .mdlrc
  README.md
```

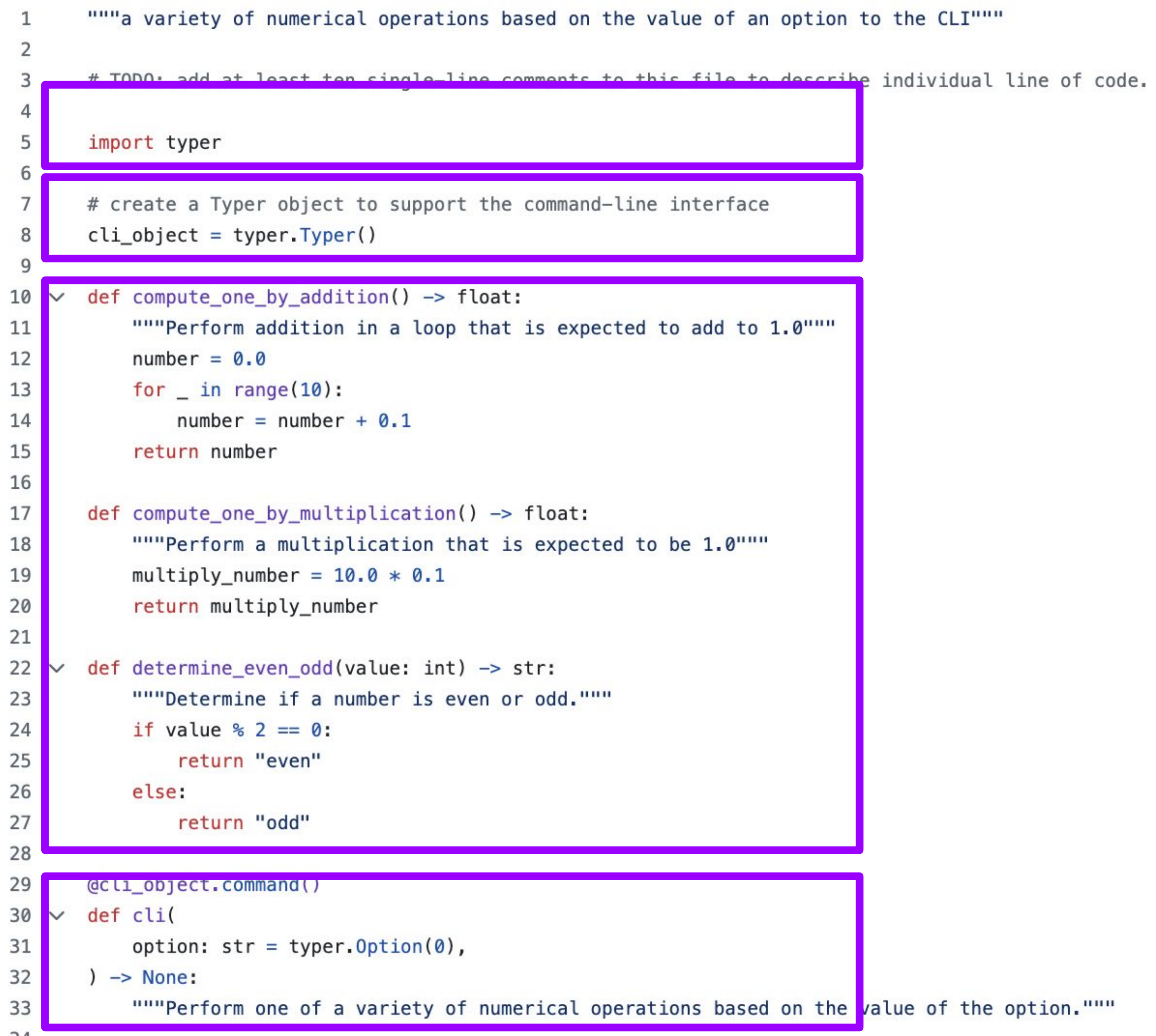
- import section——---->
- global variables——-->
- algorithm definitions

——--------------->

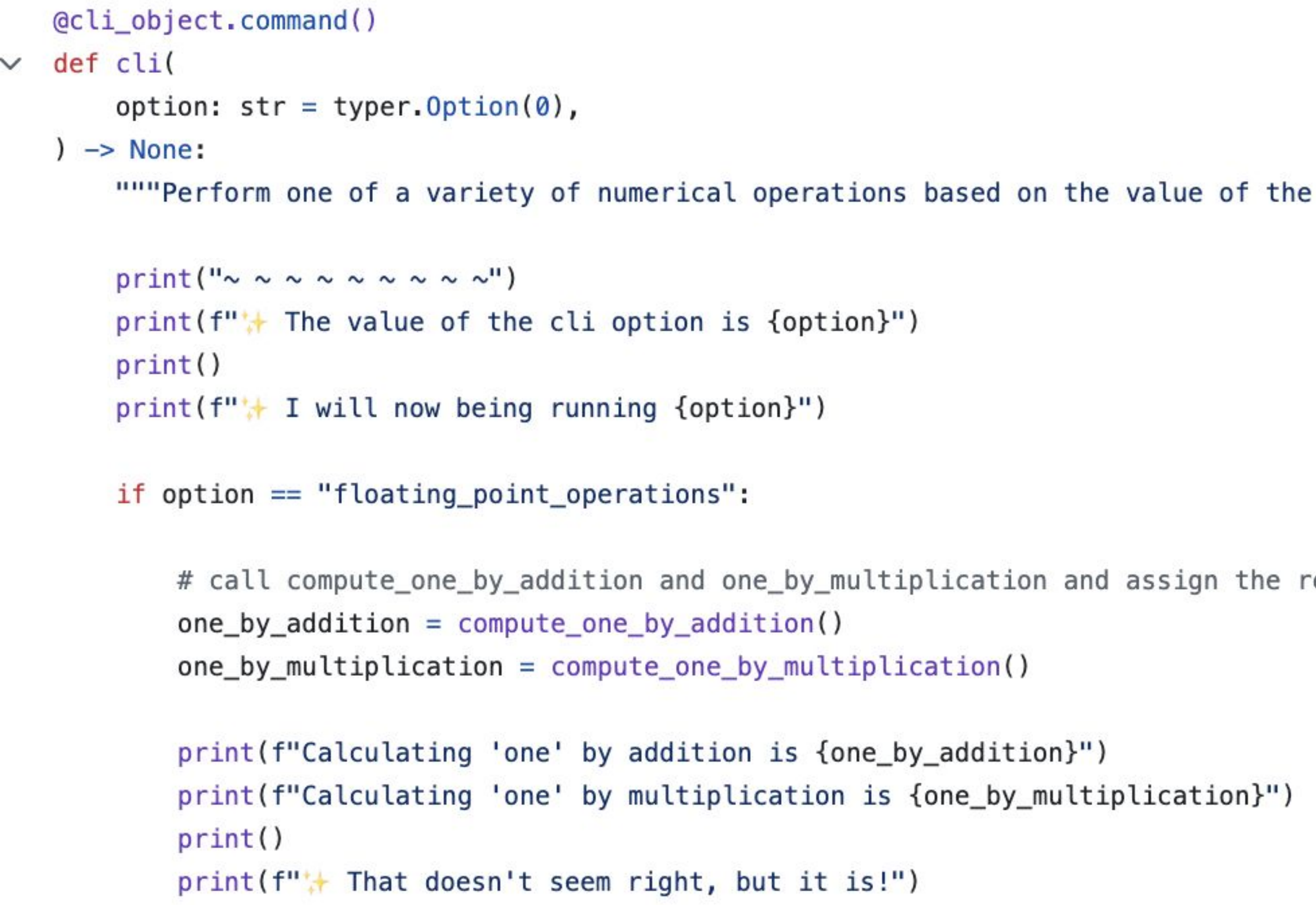
——--------------->

——--------------->

- function that defines the cli



```python
"""a variety of numerical operations based on the value of an option to the CLI"""

# TODO: add at least ten single-line comments to this file to describe individual line of code.


import typer

# create a Typer object to support the command-line interface
cli_object = typer.Typer()

def compute_one_by_addition() -> float:
    """Perform addition in a loop that is expected to add to 1.0"""
    number = 0.0
    for _ in range(10):
        number = number + 0.1
    return number

def compute_one_by_multiplication() -> float:
    """Perform a multiplication that is expected to be 1.0"""
    multiply_number = 10.0 * 0.1
    return multiply_number

def determine_even_odd(value: int) -> str:
    """Determine if a number is even or odd."""
    if value % 2 == 0:
        return "even"
    else:
        return "odd"

@cli_object.command()
def cli(
    option: str = typer.Option(0),
) -> None:
    """Perform one of a variety of numerical operations based on the value of the option."""
```



```python
@cli_object.command()
def cli(
    option: str = typer.Option(0),
) -> None:
    """Perform one of a variety of numerical operations based on the value of the option."""

    print("~ ~ ~ ~ ~ ~ ~ ~ ~")
    print(f"✨ The value of the cli option is {option}")
    print()
    print(f"✨ I will now being running {option}")

    if option == "floating_point_operations":

        # call compute_one_by_addition and one_by_multiplication and assign the return value into variables
        one_by_addition = compute_one_by_addition()
        one_by_multiplication = compute_one_by_multiplication()

        print(f"Calculating 'one' by addition is {one_by_addition}")
        print(f"Calculating 'one' by multiplication is {one_by_multiplication}")
        print()
        print(f"✨ That doesn't seem right, but it is!")
```

- displays preliminary messages
- runs predefined algorithms
- displays results

Three Ways to Run Code

Poetry for CLIs

We use poetry to help us run the programs in the command line

In the command line, you would type

- cd cli
- poetry install
- poetry run cli --option floating_point_operations

Python for Scripts

We can use the regular python interpreter to run regular python scripts

In the command line, you would type

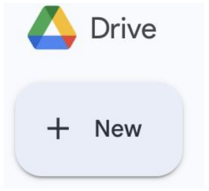
- cd script
- python main.py
    - you may have put in information when the code prompts you

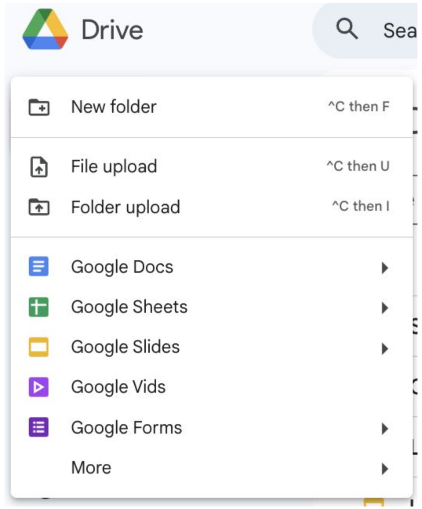
Colab for notebooks

We can use the Google Colab to run python notebooks easily in the browser

You would

- open you google drive
- click new
- the upload main.ipynb from the notebook directory
- run each cell in order, one at a time
    - you may have put in information when the code prompts you





Walkthrough