

Module 6: AST-1

TITLE: Project with GitHub Repo and Continuous Integration

LEARNING OBJECTIVES

You will be able to understand and implement the following aspects:

1. Connect local repo with GitHub remote repo
2. Understanding & using cloud development environment: GitHub Codespaces
3. Using Linux Command Line
4. Continuous Integration: GitHub Actions for automated training and testing

INTRODUCTION

CI/CD

CI/CD stands for continuous integration and continuous deployment, which is fundamentally, about automating the stages of development and deployment.

The diagram below shows an overview of these stages. The **Build** refers to preparing everything need to run the code. In the case of Python, that's installing dependencies in the requirements.txt file and making sure that the operating system that's going to run the code is set up or the virtual machine or container is built and ready to go. The **Test** refers to automatically testing the code.

The **Merge** refers to merging in a feature branch, which automatically releases that code either to production or a testing environment where it can undergo further tests, or in the case of deploying automatically to production, then immediately be monitored and used by real users.



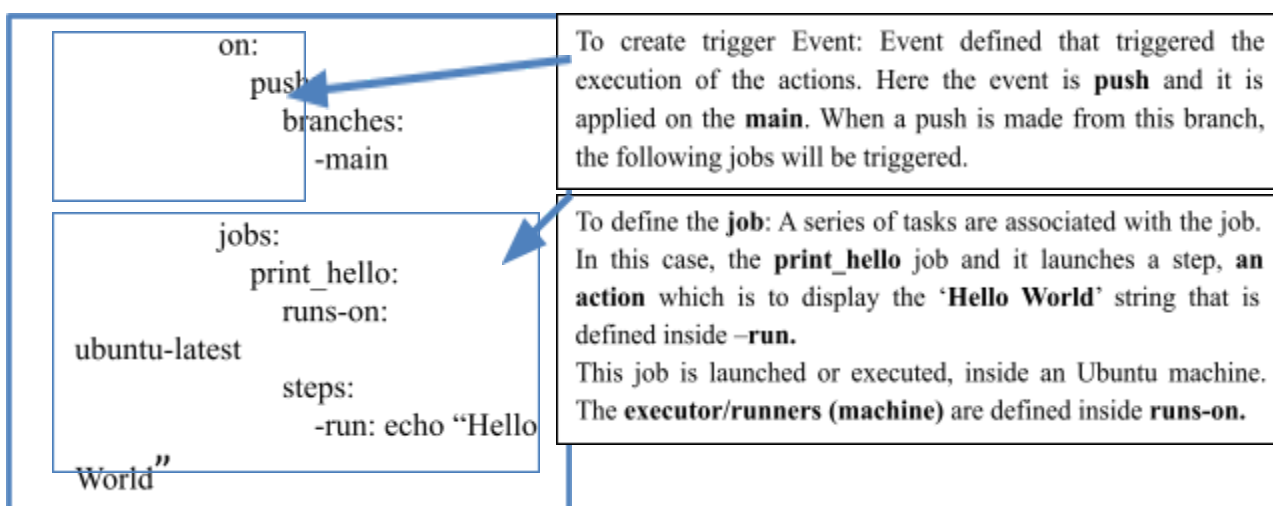
Automating all these steps ensures there's no need for a person to ever run a script or SSH into a machine, and that adds a huge amount of value. Why does automating these steps matter? Adopting this practice means that the system is always in a releasable state and allows us to react to issues quickly. Also, this ensures that the chances of breaking something upon changes or doing something that interferes with another system are reduced. This is more valuable for enterprises having regular and faster release cycles i.e. the code is being put out into production every day, or at least on a regular basis.

GitHub Actions

GitHub Actions is a platform that automates the creation, testing, and deployment of software. It also allows us to execute any code when a certain event occurs. The following three are the main components of GitHub actions, and this gives an understanding of the complete picture:

1. **Events:** An event refers to any action or occurrence that can take place within the repository. These events encompass a wide range of activities, such as submitting code changes, creating new branches, opening merge requests, commenting on issues, and more. Essentially, an event is anything that can trigger an action or initiate a process within the GitHub environment.
2. **Workflows:** A workflow is an automated process that is made up of a series of jobs that are executed when triggered by an event. Workflows are defined in YAML files and stored in a directory called GitHub Workflows.
3. **Jobs:** Jobs are a series of tasks that are executed within a workflow when triggered by an event. Each step in a script or a GitHub action would be a job at the end.

Basic workflow: The minimum components needed inside a GitHub action workflow are shown in figure below. This is an example of a workflow YAML file.



Explanation of different components:

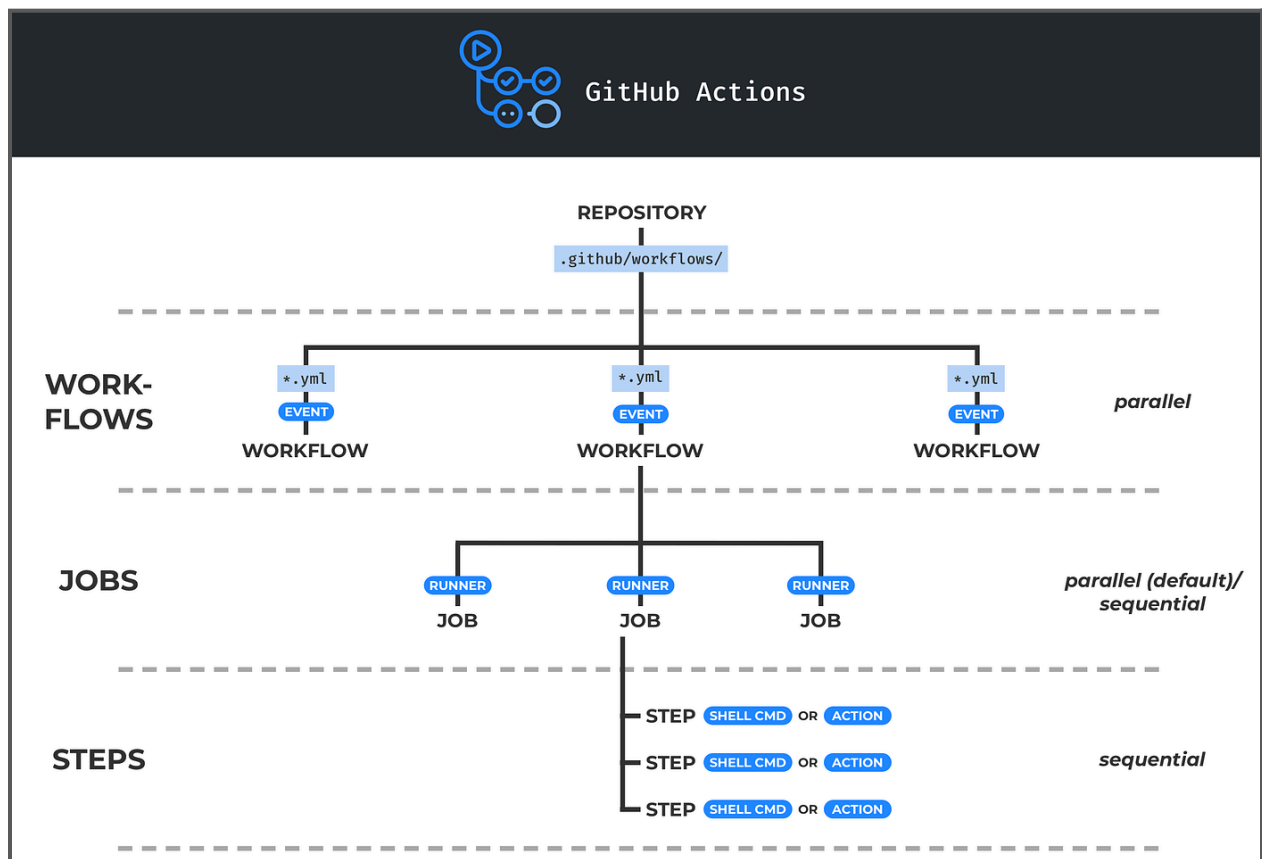
- **on** The event is defined that triggers the workflow.
- **runs on**: It indicates the machine where each job should be executed.
- **jobs** It contains the list of jobs that make up the workflow.
- **steps**: It contains a series of tasks that are executed within each job.

- **run** It indicates what is going to be executed.

Actions: An action is a command that is executed on a runner, the core element of GitHub Actions, which is named after it.

Runners: A runner is a GitHub Actions server. It listens for available jobs, runs each in parallel, and reports back progress, logs, and results. Each runner can be hosted by GitHub or self-hosted on a localized server. GitHub Hosted runners are based on Ubuntu Linux, Windows, and macOS.

GitHub Action Diagram



The diagram above depicts the GitHub actions. There is a GitHub repository and within that repository, there is a GitHub workflows folder. There are different workflows (separate YAML files) inside the GitHub workflows folder. The specified event is going to trigger a different workflow.

A workflow can have several jobs that run in parallel. Here, in the diagram, we can see a single workflow having three jobs running in parallel. Then within the job, we have different actions or steps. These actions, in the end, are individual tasks that are within a job. These actions can be very different and range from, for example, publishing a Python package in PyPI or sending a

notification email to configuring Python to a specific version before running a particular script.

Tools for Linting and Formatting

Linting highlights syntactical and stylistic problems in the Python source code, which often helps in identifying and correcting subtle programming errors or unconventional coding practices that can lead to errors. For example, linting detects the use of an uninitialized or undefined variable, calls to undefined functions, missing parentheses, and even more subtle issues such as attempting to redefine built-in types or functions. We are going to use the ‘**pylint**’ package as a linting tool.

Formatting makes code easier to read by humans. It applies specific rules and conventions for line spacing, indents, spacing around operators, and so on. Keep in mind, formatting doesn't affect the functionality of the code itself. We are going to use the ‘**black**’ package as a formatting tool and it not only reports format errors but also fixes them automatically.

Linting is distinct from formatting because linting analyzes how the code runs and detects errors, whereas formatting only restructures how code appears. Although there is a little overlap between formatting and linting, the two capabilities are complementary. The purpose of any kind of tooling like this is to manage the package, make the code easy for other people to use, and also reduce the chances of introducing bugs.

PROCEDURE

To conduct this experiment, familiarity with **Git and GitHub** accounts is required. Please review the reference document on the same topic [[Git Steps & Commands](#)]. The runtime for all cloud platforms is almost guaranteed to be the **Linux operating** system. Considering the same we need to set up the Linux terminal in the local VS Code. Follow the document [[Run Linux Terminal](#)] for setting the **Linux terminal in VS Code**. A few frequently used **Linux commands** are provided in the document [[Bash Shell & Commands](#)]. We don't have to do any complex shell scripting, knowing a few basic commands are sufficient for our purpose. First, we will conduct this experiment using local VS-code , repo, and GitHub repo.

The conducting of this experiment with cloud platforms like **GitHub Codespaces**, etc. remains the same, except that we have to follow a few extra steps for setting up the cloud accounts and environment. We will demonstrate that as well.

Follow the document [[GitHub Codespaces](#)] for getting started with **GitHub Codespaces**.

Follow the document [[AWS Account creation](#)] for creating an **AWS account**.

A. Experiment with VS Code, local repo and remote GitHub repo

Steps:


1. Go to your **GitHub account** and create a repo with a name, say '**github_action_demo**', choosing 'Add a README file' and 'Add .gitignore' with the Python template option and keeping all other default settings as it is.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner *

 yrajm1997


Repository name *

github-action-demo


✔ github-action-demo is available.

Great repository names are short and memorable. Need inspiration? How about **effective-parakeet** ?

Description (optional)

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore


.gitignore template: Python


Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set  **main** as the default branch. Change the default name in your [settings](#).

 You are creating a public repository in your personal account.

Create repository

2. Create a new folder on your system and Open it in VS Code. Start a New Terminal and if you are a Windows user, then type 'wsl' and enter. You should see the terminal opening in the Linux environment (given you have already configured WSL). In my case it is like:

- `rami@rami:/mnt/g/myproject$`
- `rami@rami:/mnt/g/myproject$ pwd`

which returns: `/mnt/g/myproject` [denoting 'myproject' folder in my system.]

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject$ pwd
/mnt/g/myproject
```

3. We want to clone the 'github_action_demo' repo in our system.

4. Generating SSH Key for Authentication of GitHub connection with the system:

- `rami@rami:/mnt/g/myproject$ ssh-keygen -t rsa`

Run the above command and it will prompt you to provide a passphrase, just keep everything to default by pressing enter.

Read the message and keep the default setting, press enter three times.

Copy the **public** key location and run the below command

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/yrajm1997/.ssh/id_rsa):
/home/yrajm1997/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/yrajm1997/.ssh/id_rsa
Your public key has been saved in /home/yrajm1997/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:ahIPVzaN09y3jtgCxe2oeIkcvYl+un/xwI0exGJGSTM yrajm1997@DESKTOP-DLM8PVA
The key's randomart image is:
+---[RSA 3072]-----+
|      .E.      |
|      = .00 .  |
|      * +... 0 |
|      o o .+00= |
|      o . S .00++o. |
|      = . . 0 0* . |
|      . +      * X..= |
|      o . . . 0 . |
|      +==..    |
+---[SHA256]-----+
```

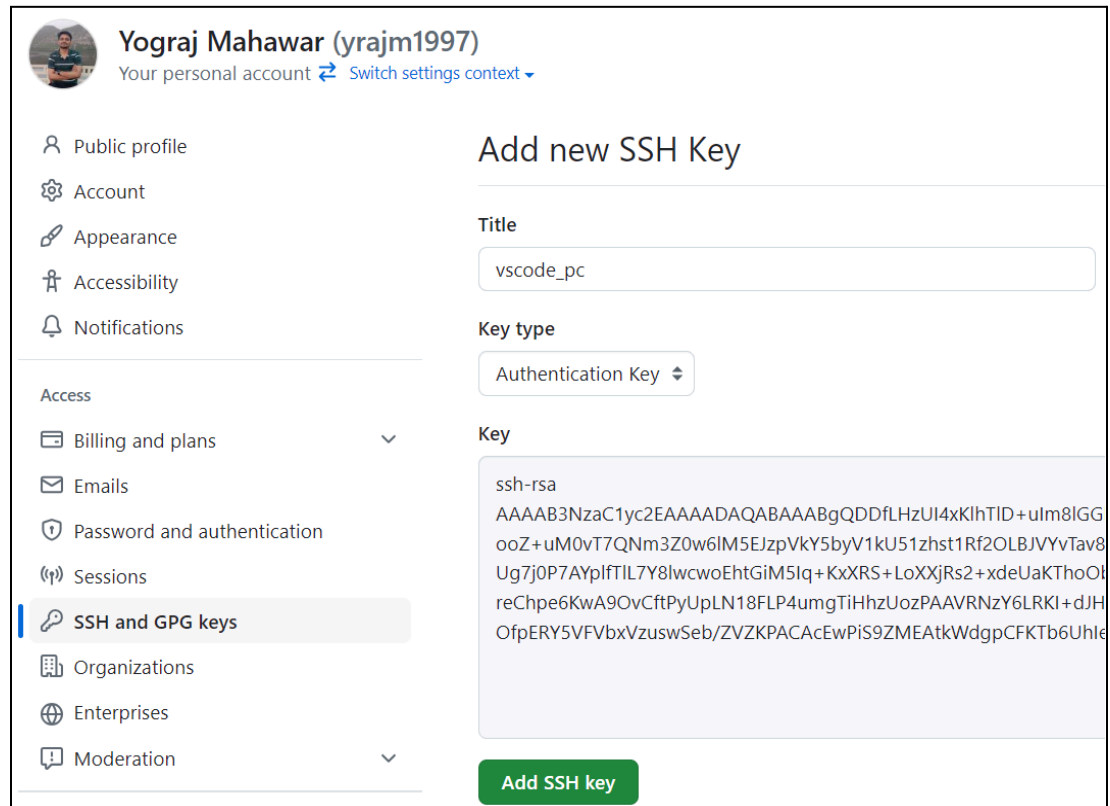
- `rami@rami:/mnt/g/myproject$ cat /home/rami/.ssh/id_rsa.pub`

Copy the entire output which would be something like this:

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject$ cat /home/yrajm1997/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQGDGfLHzUI4xKlTlD+uIm8lGGlF6ZJDl94uwl8cKVEqu0dv2aH+oqNwEe
BJVYvTav8V6iuhXJ7CiqVElatg0ztyVo0k12gYPSAbk+aWCElZ/DuASF2NXxmMr6frP61DdUg7j0P7AYp1fTlL7Y8lwcwo
YAA1kxloH0kpMgNHvbPmMIYreChpe6KwA90vcftPyUpLN18FLP4umgTiHhzUozPAAVRNzY6LRKI+dJHd+wUveDU6QtWtCP
ACAcEwPiS9ZMEatkwDgpcFKTb6UhIeCuJbNLPf1480RMj9XoLkWD0= yrajm1997@DESKTOP-DLM8PVA
```

Go to your GitHub account → Setting → SSH & GPG keys → New SSH key

→ Provide **Title** as 'vscode_pc' and paste that copied text from above rsa.pub inside **Key box**. Keep **Key type** as **Authentication Key**. Finally click **Add SSH key**. This will prompt you to provide a password for your GitHub account.



Yograj Mahawar (yrajm1997)
Your personal account [Switch settings context](#)

- Public profile
- Account
- Appearance
- Accessibility
- Notifications
- Access
 - Billing and plans
 - Emails
 - Password and authentication
 - Sessions
 - SSH and GPG keys**
 - Organizations
 - Enterprises
 - Moderation

Add new SSH Key

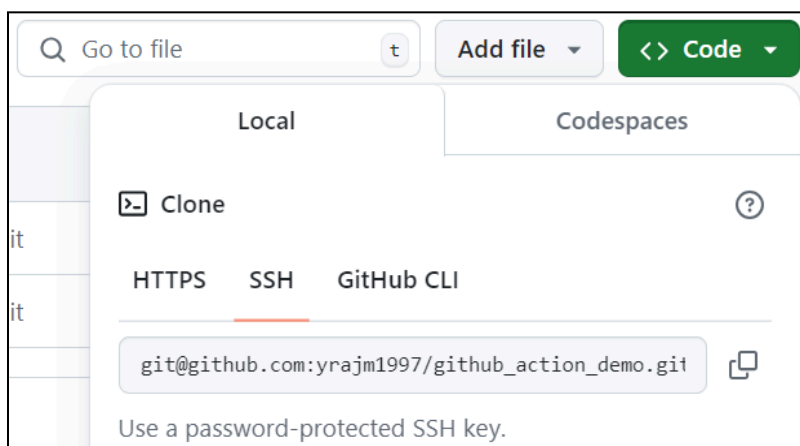
Title
vscode_pc

Key type
Authentication Key

Key
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQgQDDfLHzUI4xKlHtID+ulm8lGG
ooZ+uM0vT7QNm3Z0w6IM5EJzpVky5byV1kU51zhst1Rf2OLBJVYvTav8
Ug7j0P7AYplfTIL7Y8lwcwoEhtGim5lq+KxXRS+LoXXjRs2+xdeUaKThoO
reChpe6KwA9OvCftPyUpLN18FLP4umgTiHhzUozPAAVRNzY6LRKI+dJH
OfpERY5VFVbxVzuswSeb/ZVZKPACAcEwPiS9ZMEAtkWdgpCFKTb6Uhle

Add SSH key

5. **Cloning the GitHub repo:** Go to the particular repo in github ['github_action_demo'] and copy the clone url for ssh: click on → **Code** → **SSH** → and copy the URL.



Go to file [t](#) [Add file](#) [Code](#)

Local **Codespaces**

Clone ?

HTTPS **SSH** GitHub CLI

git@github.com:yrajm1997/github_action_demo.git

Use a password-protected SSH key.

Finally run the following command in terminal: `git clone your_copied_repo_url`

For example:

- `rami@rami:/mnt/g/myproject$ git clone git@github.com:yrajm1997/github_action_demo.git`

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject$ git clone git@github.com:yrajm1997/github_action_demo.git
Cloning into 'github_action_demo'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

Now you can see the 'github_action_demo' folder on your system.

6. Then move inside the 'github_action_demo' folder and do git status.

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject$ ls
github_action_demo
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject$ cd github_action_demo
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ ls
README.md
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$
```

Now we are ready to create and write different sub-folders & scripts that can be executed and pushed back to GitHub remote repository.

7. Creating and activating virtual environment:

- `.....$ python3 -m venv venv`
- `.....$ source venv/bin/activate`

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ python3 -m venv venv
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ source venv/bin/activate
(venv) yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$
```

Here, a virtual environment has been created and activated. You can notice the → (venv) at the start of the new command line.

8. Creating requirement file and installing the requirements: Run the command below:

-\$ touch requirements.txt

A blank text file is created. Open that in VS Code, type these libraries as given below and save it.

requirements.txt

```
# requirements.txt
black
pylint
pytest
```

Check the installed packages in venv:

-\$ pip freeze

Nothing will be displayed. Now install the requirements:

-\$ pip install -r requirements.txt

Check the installed packages in venv:

-\$ pip freeze

Note: Ctrl + L command clears the terminal screen.

9. Writing python script:

-\$ touch script.py

A blank python file is created. Open that in vs code, type the code given below and save it.

script.py:

```
# script.py
def addition (A, B):
    return A + B
```

Create a blank __init__.py file so that the folder can be treated as a module and import is possible.

-\$ touch __init__.py

10. Creating scripts for test cases:

-\$ touch test_{1,2}.py

Open these files in vs code, type the code given below and save it.

test_1.py:

```
# test_1.py
import os, sys
```

```
sys.path.insert(0, os.getcwd())

from script import addition
def test_add():
    subj = addition(2, 3)
    assert subj == 5
```

test_2.py:

```
# test_2.py
import os, sys
sys.path.insert(0, os.getcwd())

from script import addition

def test_data_type():
    subj = addition(2, 3)
    assert isinstance(subj, int)
```

11. Creating test directory and moving test cases inside that:

-\$ **mkdir tests**

Notice a blank folder name 'tests' is created. Now the following command moves test files into the tests folder.

-\$ **mv test*.py ./tests**

Go inside the tests folder and you can see test_1.py and test_2.py files inside it.

12. Running the test cases: Pytest automatically is able to find a folder with name as 'test' and run scripts starting with 'test'.

-\$ **pytest**

You will notice the result of test cases.

```
(venv) yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ pytest
===== test session starts =====
platform linux -- Python 3.10.6, pytest-8.2.2, pluggy-1.5.0
rootdir: /mnt/g/myproject/github_action_demo
collected 2 items

tests/test_1.py .
tests/test_2.py .

===== 2 passed in 0.32s =====
```

13. Understanding linting and formatting: Run the command below for linting:

-\$ `pylint *.py`

You can see the complaints about style guides and conventions of best practices of writing codes.

```
(venv) yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ pylint *.py
***** Module github_action_demo.script
script.py:3:0: C0304: Final newline missing (missing-final-newline)
script.py:1:0: C0114: Missing module docstring (missing-module-docstring)
script.py:2:0: C0116: Missing function or method docstring (missing-function-docstring)
script.py:2:14: C0103: Argument name "A" doesn't conform to snake_case naming style (invalid-name)
script.py:2:17: C0103: Argument name "B" doesn't conform to snake_case naming style (invalid-name)

-----
Your code has been rated at 0.00/10
```

Now run the command below for formatting:

-\$ `black *.py`

```
(venv) yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ black *.py
reformatted script.py

All done! ✨ 🍰 ✨
1 file reformatted, 1 file left unchanged.
```

Notice the above result that reformatting is done.

Now, again re-write the code in script.py as given below and save it.

script.py:

```
# script.py
def addition      (A,                B):

    return      A      +      B
```

Now run the below command again for formatting. You will notice the code is reformatted automatically.

-\$ `black *.py`

14. Understanding and creating Makefile: Create a file with a name ‘**Makefile**’ without any extension, open and write the syntax as given below:

-\$ touch Makefile

Makefile:

```
# Makefile
install:
    pip install --upgrade pip &&\
    pip install -r requirements.txt
format:
    black *.py
lint:
    pylint --disable=R,C script.py
test:
    python -m pytest tests/test_*.py

all: install lint test format
```

A Makefile runs “recipes” via the Make system, which comes with Unix-based operating systems. Therefore, a **Makefile** is an ideal choice to simplify the steps involved in continuous integration and **saves us from writing the same tedious code again and again.**

Run the commands like:\$ make test

```
(venv) yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ make test
python -m pytest tests/test_*.py
===== test session starts =====
platform linux -- Python 3.10.6, pytest-8.2.2, pluggy-1.5.0
rootdir: /mnt/g/myproject/github_action_demo
collected 2 items

tests/test_1.py .
tests/test_2.py .

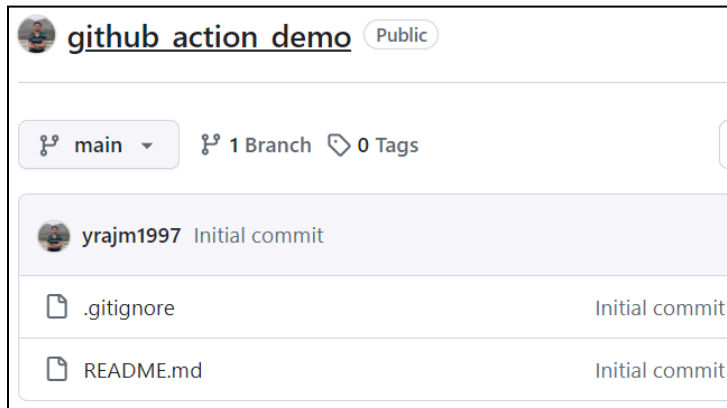
===== 2 passed in 0.14s =====
```

Try all commands given below, explanation is written.

- **make install:** This step installs software via the make install command
- **make format:** This step reformats the code if required via make format command.
- **make lint:** This step checks for syntax errors via the make lint command
- **make test:** This step runs tests via the make test command:
- **make all:** This step runs all above step one by one in one make all command.

15. Pushing all the changes from local repo to remote GitHub repo: Whatever files and folders have been created, those are in local repo only and need to be pushed into GitHub repo.

First check the remote repo, it contains only `Readme` and `.gitignore` file.



Now, run following commands and the local repo should be pushed to the remote repo.

-\$ `git status` [Get information about untracked files and folders]

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Makefile
    __init__.py
    requirements.txt
    script.py
    tests/

nothing added to commit but untracked files present (use "git add" to track)
```

-\$ `git add .` [Stage the untracked files and folders]
-\$ `git commit -m "Adding all files"` [Snapshot of staged files]

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ git commit -m "Adding all files"
[main 510d2b9] Adding all files
6 files changed, 37 insertions(+)
create mode 100644 Makefile
create mode 100644 __init__.py
create mode 100644 requirements.txt
create mode 100644 script.py
create mode 100644 tests/test_1.py
create mode 100644 tests/test_2.py
```

-\$ **git push** [Finally pushing all contents from local to remote repo]

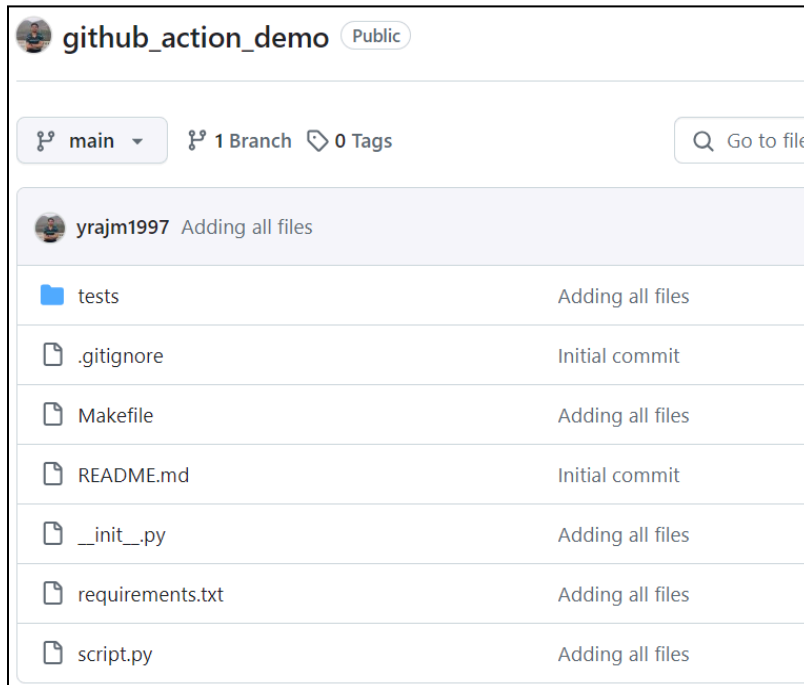
```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 983 bytes | 42.00 KiB/s, done.
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To github.com:yrajm1997/github_action_demo.git
fb7b4b0..510d2b9 main -> main
```

Check the status again.

```
yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Now again check the remote repo, all files are pushed.

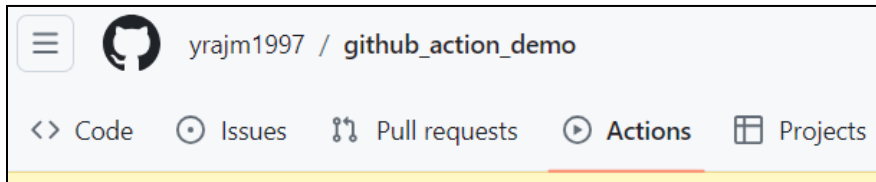


The screenshot shows the GitHub repository page for 'github_action_demo' (Public). The main branch is selected, showing 1 Branch and 0 Tags. A commit by yrajm1997 is highlighted with the message 'Adding all files'. Below the commit, a list of files is shown with their commit status:

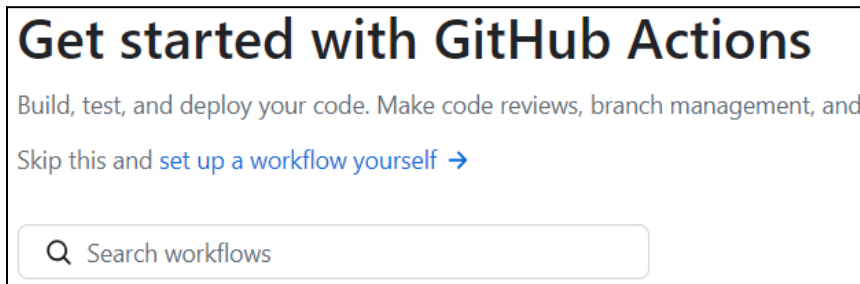
File	Status
tests	Adding all files
.gitignore	Initial commit
Makefile	Adding all files
README.md	Initial commit
__init__.py	Adding all files
requirements.txt	Adding all files
script.py	Adding all files

16. GitHub Action:

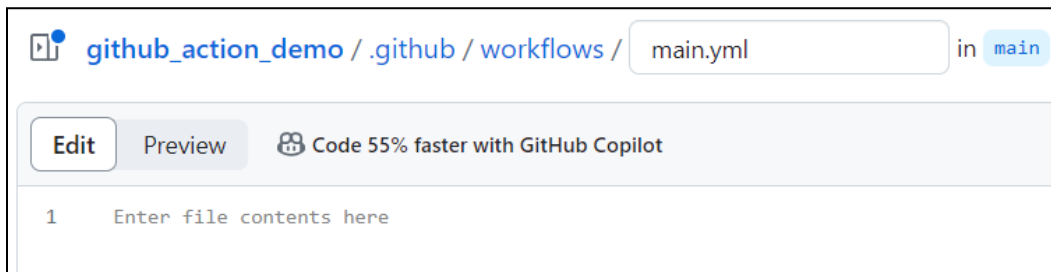
Go to your repo '**github_action_demo**' on GitHub and click on the '**Actions**' tab.



There are many suggested templates, but we are going with the **set up a workflow yourself** → option. Click on it.



It will open a blank yaml file inside '**github_action_demo/.github/workflows**' directory.



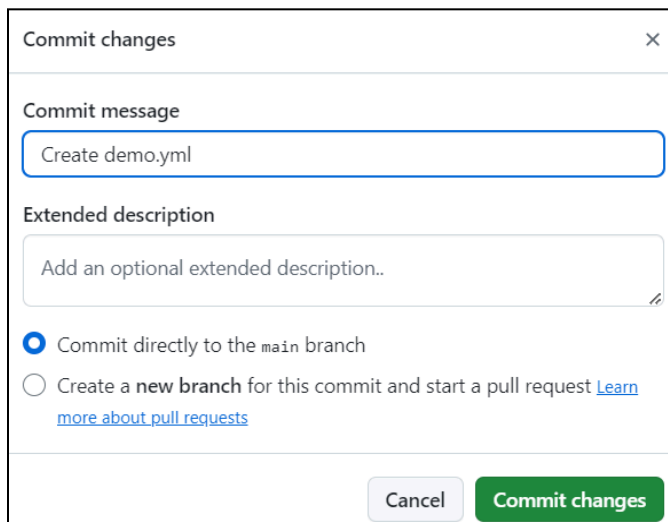
We can rename the .yaml file, say it is **demo.yaml**. Now copy the below code and paste inside that blank yaml file. Then click on '**Commit changes**'.

demo.yaml:

```
# demo.yaml
# This workflow will install Python dependencies, run tests with a variety of
Python versions
name: Python app
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
```

```
build:
  runs-on: ubuntu-latest
  strategy:
    fail-fast: false
    matrix:
      python-version: ["3.8", "3.9", "3.10"]
  steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${ matrix.python-version }
      uses: actions/setup-python@v3
      with:
        python-version: ${ matrix.python-version }
    - name: Install dependencies
      run: |
        make install
    - name: Lint with pylint
      run: |
        make lint
    - name: test with pytest
      run: |
        make test
    - name: Format code with Black
      run: |
        make format
```

Give a commit message, select ‘Commit directly to the main branch’, then click on ‘Commit changes’.



The image shows a 'Commit changes' dialog box with the following fields and options:


- Commit message:** A text input field containing 'Create demo.yml'.
- Extended description:** A text input field with the placeholder text 'Add an optional extended description..'. There is a small edit icon (pencil) to the right of the field.
- Commit options:**
 - ☒ Commit directly to the main branch
 - ☐ Create a new branch for this commit and start a pull request [Learn more about pull requests](#)
- Buttons:** 'Cancel' and 'Commit changes' (highlighted in green).

Again, go to the **Actions** tab. You will see the workflow running. Click on it.

All workflows

Showing runs from all workflows

1 workflow run


Create demo.yml
 Python app #1: Commit [f2d1e18](#) pushed by yrajm1997




You will see the status of the workflow, total duration it took to complete. To see the steps executed, click on **3 jobs completed**, then click on any one **build**.

Re-run triggered 1 minute ago

Status

Total duration


Artifacts

 yrajm1997  f2d1e18  main	Success	20s	—
---	----------------	------------	---

demo.yml

on: push


Matrix: build


3 jobs completed
[Show all jobs](#)


demo.yml

on: push


Matrix: build


build (3.10)

8s

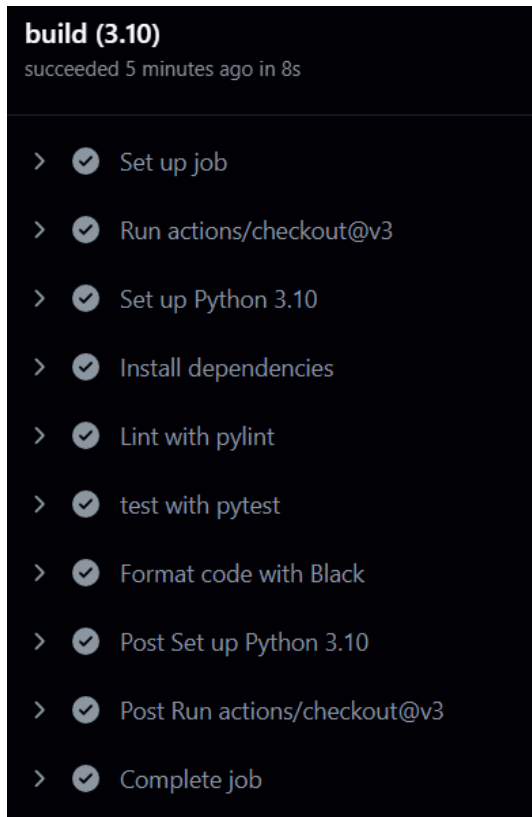

build (3.8)

9s

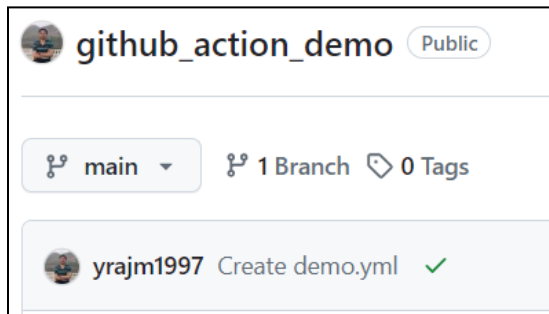

build (3.9)

10s

You can see each step and actions executed.



Now go to the **Code** tab of your repo and notice a green tick (✓). Click on it.



It would display “All checks have passed”. We can see the details as well.



Note: Makefile is not necessary for creating GitHub Actions workflow. Instead of **make** we can directly write commands in workflow yml file. GitHub actions workflow without a Makefile is implemented in later steps below for the Titanic example.

To update the local repo as per the changes in the remote repo run:\$ **git pull**

```

yrajm1997@DESKTOP-DLM8PVA:/mnt/g/myproject/github_action_demo$ git pull
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), 1.33 KiB | 16.00 KiB/s, done.
From github.com:yrajm1997/github_action_demo
  510d2b9..f2d1e18  main      -> origin/main
Updating 510d2b9..f2d1e18
Fast-forward
 .github/workflows/demo.yml | 33 ++++++++++++++++++++++
 1 file changed, 33 insertions(+)
 create mode 100644 .github/workflows/demo.yml

```

Whenever there is any push on the GitHub repo, the GitHub workflow would be triggered to run all the tests along with all other actions. To check this, update the **Readme** file in the local repo like given in the text box below.

Open Readme.md file in VS Code and type the explanation below and save it.

Readme.md:

```

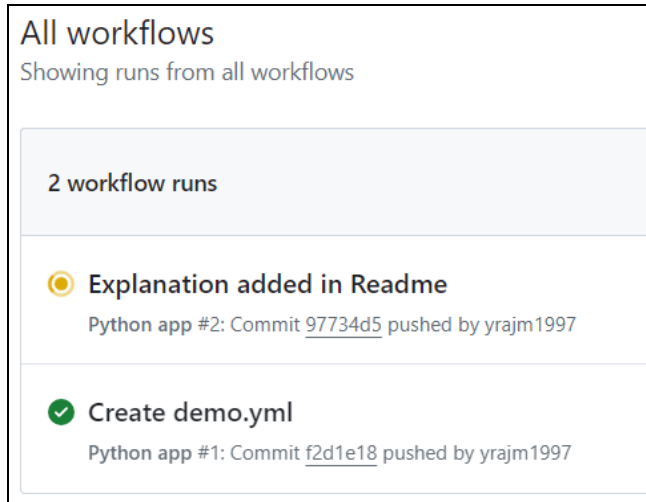
### github_action_demo
This repo is created for the demonstration of GitHub Actions.
Connecting local repo with GitHub repo and working with VS Code Linux
terminal.

```

Now run the following commands to push the changes to remote repo:

-\$ **git status**
-\$ **git add .**
-\$ **git commit -m "Explanation added in Readme"**
-\$ **git push**

Open the GitHub Actions you will see a new workflow has started to run. So, every push or commit in the remote repo will trigger the workflow.



The complete example above is a demonstration of Continuous integration (CI) along with the best practices of writing code in a production environment. This is a solid foundation for further implementation of CI in any other projects.

In a nutshell, CI is the process of continuously testing a software project and improving the quality based on these test results. It is automated testing using open source and SaaS build servers such as GitHub Actions, Jenkins, Gitlab, CircleCI, or cloud-native build systems like AWS Code Build. We have used GitHub Actions in our demonstration.

17. Now we will implement the CI concept in the ‘**Titanic Project**’ from **Module 3**. Since the project is ready and we don’t have to write any new script, we have to push it into our GitHub account and write the GitHub Action workflows. Follow the **steps** given below:
 - a. Download the project folder ***project_with_test***, shared along with this document, in your local system.
 - b. Go to your GitHub account and create a repo having the name ‘**MLOps**’ with Readme and .gitignore (choose python template) file.
 - c. Clone the remote ‘**MLOps**’ folder in your local system and move the local project folder inside this ‘**MLOps**’ folder. Finally, push this folder into the remote GitHub repo. You can create a virtual environment, run a train pipeline and carry out all the tests inside VS Code as done previously.
 - d. Go to the remote repo (inside ‘**MLOps**’) and through the **Actions** tab create the following workflow yml file with the name ‘**python-package**’ and commit it.

python-package.yml:

```
# This workflow will install Python dependencies, run tests with a
variety of Python versions.
name: Python package
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.8", "3.9", "3.10"]
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${ matrix.python-version }
        uses: actions/setup-python@v3
        with:
          python-version: ${ matrix.python-version }
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r
project_with_test/requirements/test_requirements.txt
      - name: Train pipeline
        run: |
          python project_with_test/titanic_model/train_pipeline.py
      - name: Test with pytest
        run: |
          pytest
```

Now click on the **Actions** tab again and check the workflow run. Note that here, we are not using Makefile. You can try with Makefile on your own.

Be careful Linux commands are different from Windows powershell commands, and the same command may behave differently in both environments.

B. Experiment with GitHub Codespaces

Steps:

1. Create a new GitHub repository. You can use the same GitHub repository you created at step Part-A 17. b
2. Create a new GitHub Codespace by clicking on **Code** dropdown >> **Create Codespace on main**.
It will open an IDE almost equivalent to VS Code.
3. Now you can create a Python virtual environment, activate it, perform training and testing. Add, update, or delete your application files.
4. Later you can commit the changes and push to the remote GitHub repository.
5. **Terminate Codespaces after the experiment:**
GitHub Codespaces are free for 120 hours per month. Make sure you stop and delete your codespaces once you no longer need them, that is, after committing and pushing the changes to your GitHub repository.
You can check all your Codespaces whether they are in active or stopped state by going to <https://github.com/codespaces>.