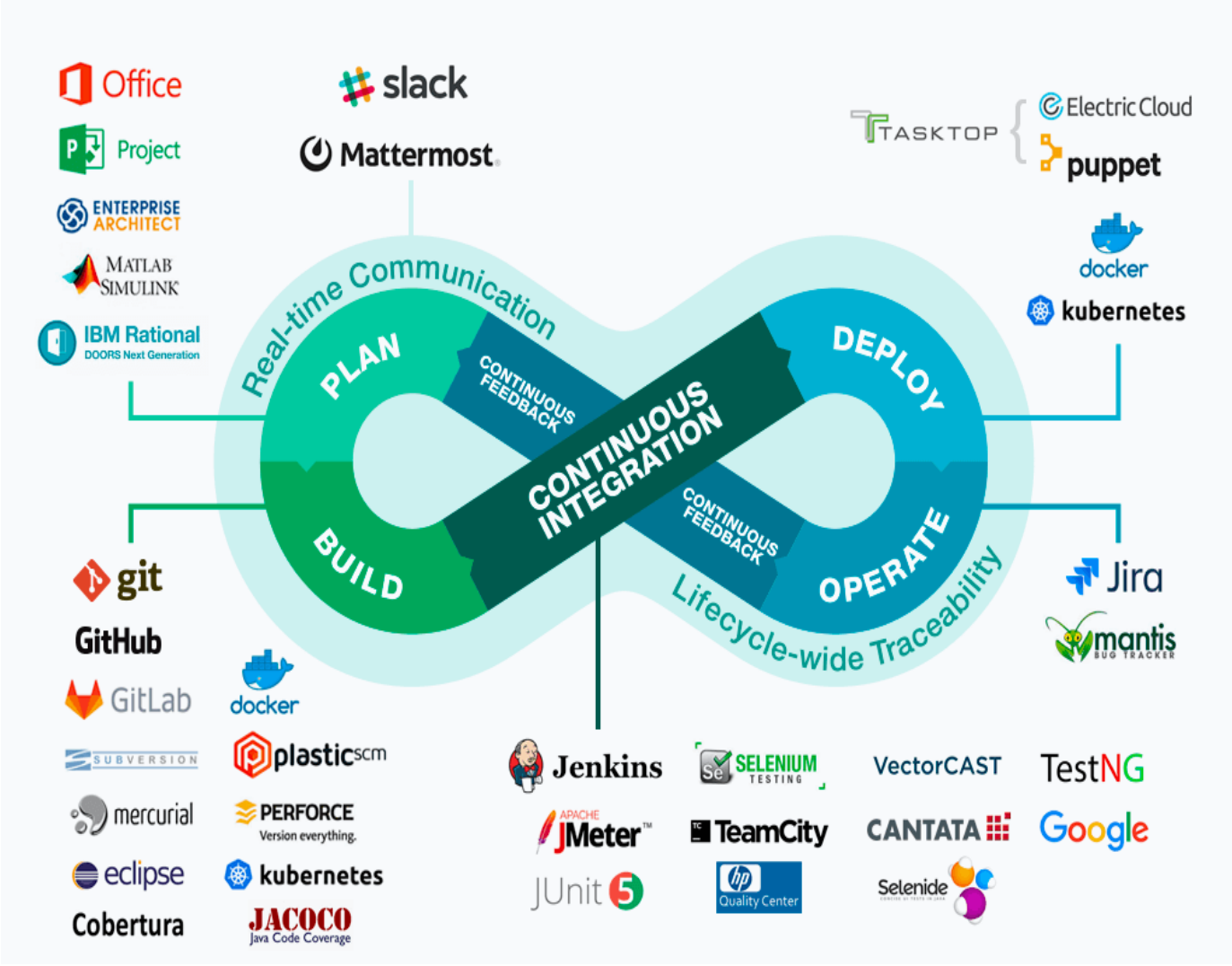
BASIC SCIENTIFIC CALCULATOR- REPORT



By Mohd Asad Ansari
(MT2020147)
SOFTWARE PRODUCTION ENGINEERING

PROJECT STATEMENT

Creating a terminal based basic calculator program with operations such calculating square root, factorial, natural log, and power, adding each functionality incrementally using devops tools like CircleCI, Docker, Git, and Github. The main objective of the project is to learn devops concept CI/CD which is achieved by creating a circleci pipeline.

DevOps:

DevOps is the combination of software development and operations. This practice allows a single team to handle the entire application lifecycle-from development, to testing, to deployment, and operations.

DevOps Pipeline :

Here we are automating the integration chain which includes source control management, and continuous integration. The cycle would include pushing latest changes or code to git (GitHub), building code includes using tools, pushing build docker image onto docker hub, this part of continuous automation is called continuous integration.
Then we integrate circleci with docker and github here does the continuous integration. CircleCI pulls the pushed docker image from docker hub and deploys it on a docker container. Tools used includes

- GitHub - https://github.com/asadiiitb/Dev-Ops-MiniProject
- Docker image - https://hub.docker.com/repository/docker/imasadansari/devops_calculator
- Continuous Integration – CircleCI
- Kibana (For Visualizing and Monitoring)

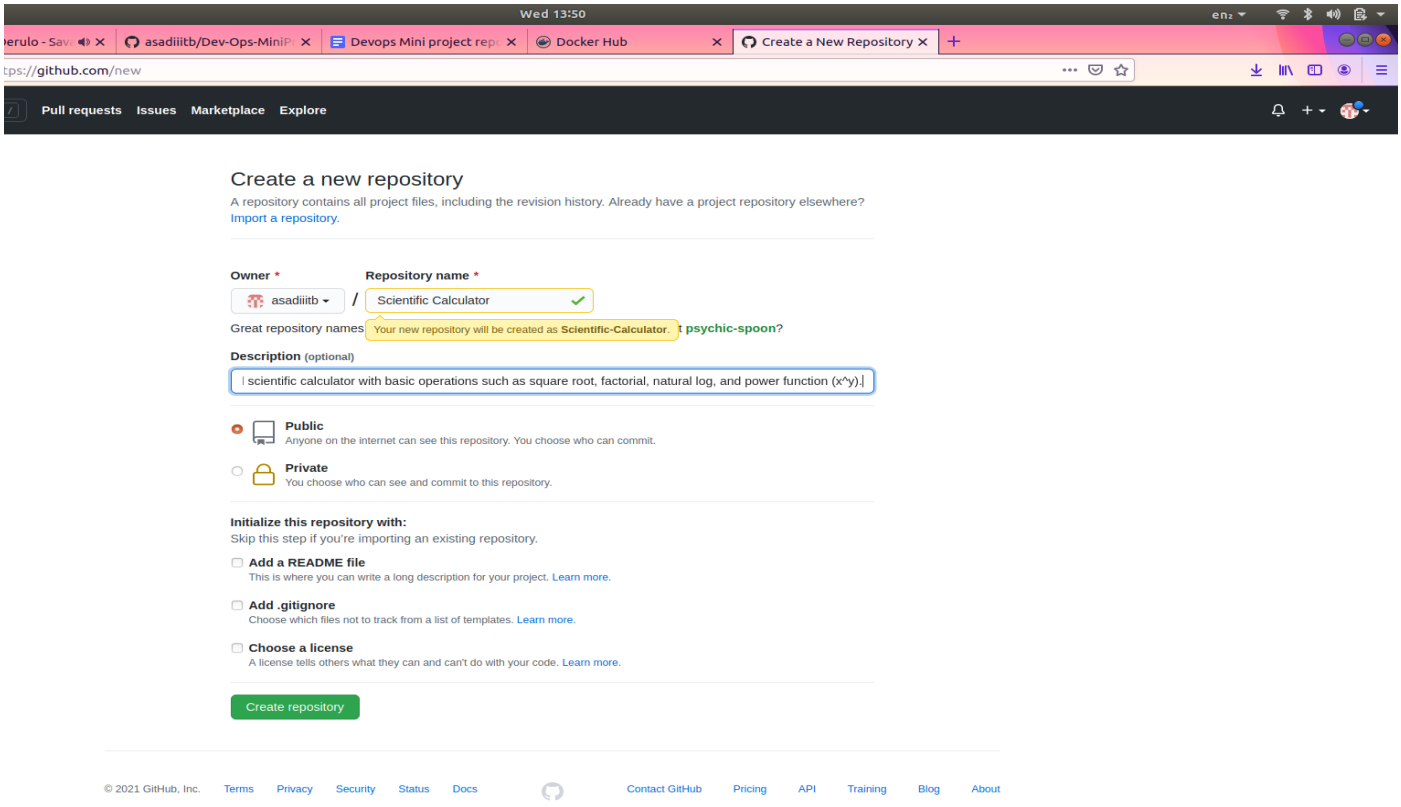


Fig 1 :- Creating a Repository on github

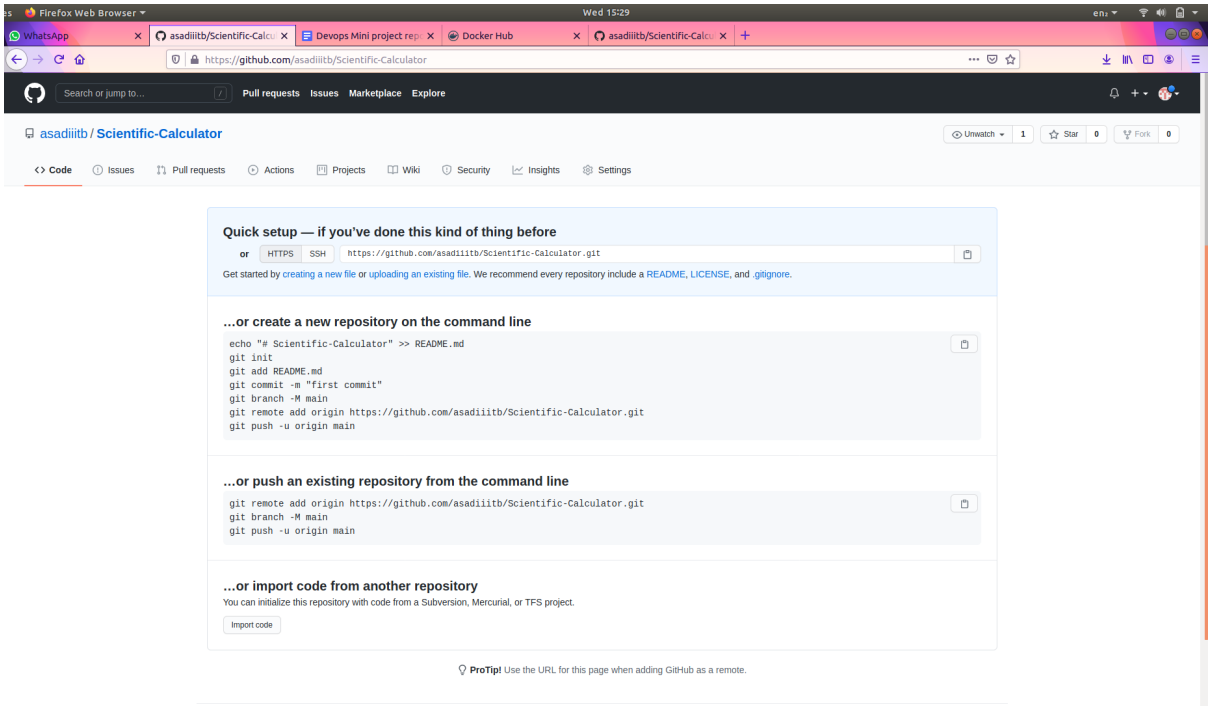


Fig 2:- Empty repository Created

Source Control Management :-

Creating a new repository on to https://www.github.com. This includes adding repository name and description. The repository name should be unique for the signed in user. The similar is done and one such repository is created for the calculator dev-ops project and can be found at https://github.com/asadiiitb/Dev-Ops-MiniProject. The SCM handles our code and is used to connect as input to CircleCI. Other SCM are gitlab, bitbucket.

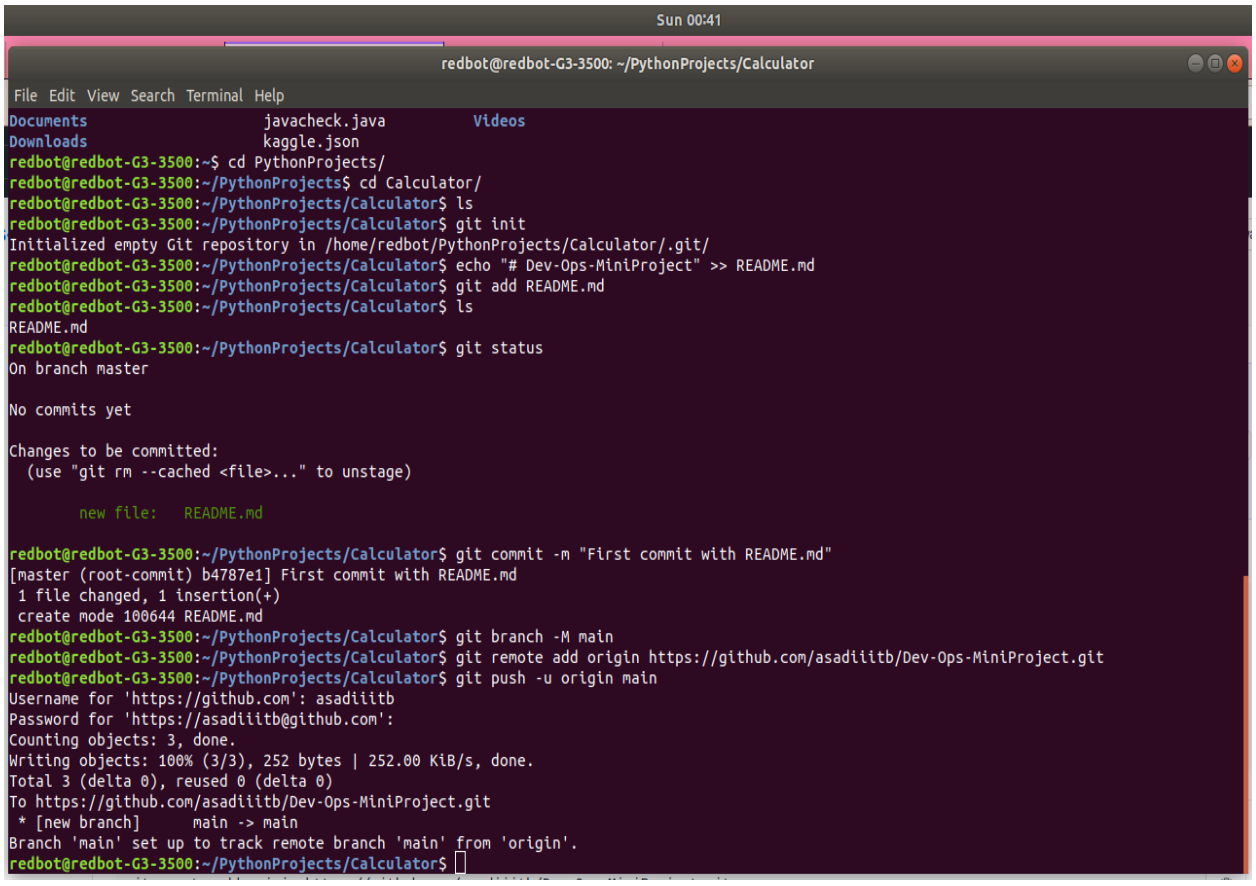


Fig 3: Initializing SCM

Installing Git and its Commands:

To Install Git use commands:

- sudo apt-get update
- sudo apt-get install git
- git –version

```
zipp==3.4.1
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git --version
git version 2.17.1
```

We need to tell git who are we, so the username will be used for pushing into repository

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git config --global user.name "Mohammed Asad Ansari"
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git config --global user.email "mohammadasad.ansari@iiitb.org"
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Now if you have already developed the project and want to push to yours create repo, you can initiate the current directory –

- git init :- To initiate the current directory
- git remote add origin <git_repo_url>
- git push -f origin <branch_name>

Or if you creating a new project I suggest doing a

- git clone <git_repo_url>

```
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git init
Initialized empty Git repository in /home/redbot/PythonProjects/Calculator/.git/
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ echo "# Dev-Ops-MiniProject" >> README.md
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git add README.md
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ ls
README.md
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md

redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git commit -m "First commit with README.md"
[master (root-commit) b4787e1] First commit with README.md
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git branch -M main
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git remote add origin https://github.com/asadiiitb/Dev-Ops-MiniProject.git
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ git push -u origin main
Username for 'https://github.com': asadiiitb
Password for 'https://asadiiitb@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 252 bytes | 252.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/asadiiitb/Dev-Ops-MiniProject.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

To push the code on to repository follow following commands

- git add <changed_files_path>
- git commit -m “Commit message name”
- git push (only if you are on master branch, otherwise It is suggested merging your branch with master first, resolving conflicts and then do git push).
- git status :- To check status of the tracked and untracked file.
- git add <File_name> :- to add file to staging area.

Setting Up Python and Working with Virtual Environment:

Here we are using python so we will install python using command:

- sudo apt-get install python3

Then to check whether it is installed or not check version using command:

- python3 -V

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip3 -V
pip 9.0.1 from /home/redbot/PythonProjects/Calculator/venv/lib/python3.6/site-packages (python 3.6)
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip -V
pip 9.0.1 from /home/redbot/PythonProjects/Calculator/venv/lib/python3.6/site-packages (python 3.6)
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ python3 -V
Python 3.6.9
```

Since we are working on the local system currently, and to cope up with the updates it is always advised and better to use a virtual environment for your project. Advantage of it will be that whenever in future if we want to use this project and we upgrade a certain library or package or it got deleted we don't need to cope up with environment we will save the libraries and their versions we are working with while developing this code, in a **'requirements.txt'** file. Using this file we can install those libraries anytime in future and work with it (Virtual Environment also has certain more advantages).

To manage software packages for Python, install **pip**, a tool that will install and manage libraries or modules to use in your projects.

- sudo apt install -y python3-pip

To check pip version use command:

- pip -V or pip3 -V

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip3 -V
pip 9.0.1 from /home/redbot/PythonProjects/Calculator/venv/lib/python3.6/site-packages (python 3.6)
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip -V
pip 9.0.1 from /home/redbot/PythonProjects/Calculator/venv/lib/python3.6/site-packages (python 3.6)
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ python3 -V
Python 3.6.9
```

To install any Package use :

- pip3 install <package name>

Installing Venv: Virtual environments enable you to have an isolated space on your server for Python projects. We'll use **venv**, part of the standard Python 3 library, which we can **install** by:

- sudo apt install -y python3-venv

Creating a Virtual Environment : You can create a new environment with the **pyvenv** command. Here, we can call new environment by any name, whatever you want:

- python3.6 -m venv <Environment Name>

Activate the environment using the command below, where my_env is the name of your programming environment.

- Source <Environment Name>/bin/activate [This works for linux and mac users]

```
redbot@redbot-G3-3500:~/PythonProjects/Calculator$
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ source venv/bin/activate
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Now, we can start working in this environment, isolated from our actual environment.

Note that we have installed external packages (pytest) and others might not have these installed. The other developers need the correct version of libraries and packages we have used in our current environment to replicate our work. This is where requirements.txt comes into play. If we provide a file listing the necessary packages, others can simply pip install -r requirements.txt and all the program's required packages and libraries will be downloaded, installed, and ready to go in one shot.

To save current environment use

- pip3 freeze > requirements.txt

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip3 freeze
appdirs==1.4.4
attrs==20.3.0
coverage==5.5
distlib==0.3.1
filelock==3.0.12
flake8==3.8.4
importlib-metadata==3.7.2
importlib-resources==5.1.2
iniconfig==1.1.1
mccabe==0.6.1
packaging==20.9
pkg-resources==0.0.0
pluggy==0.13.1
py==1.10.0
pycodestyle==2.6.0
pyflakes==2.2.0
pyparsing==2.4.7
pytest==6.2.2
pytest-cov==2.11.1
six==1.15.0
toml==0.10.2
typing-extensions==3.7.4.3
virtualenv==20.4.2
zipp==3.4.1
```

Fig : Pip freeze output showing used libraries and corresponding versions

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip3 freeze > requirements.txt
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ ls
 Calculator.log       Dockerfile            'Logs last 5 hours.png'   README.md          Scientific_Calculator.py       venv
 Calculator_main.py  'Logs last 3 days.png'   __pycache__             requirements.txt   test_ScientificCalculator.py
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Fig: requirements.txt now present in directory

To install the saved libraries use

- Pip3 install -r requirements.txt

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip3 install -r requirements.txt
Requirement already satisfied: appdirs==1.4.4 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 1))
Requirement already satisfied: attrs==20.3.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 2))
Requirement already satisfied: coverage==5.5 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 3))
Requirement already satisfied: distlib==0.3.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 4))
Requirement already satisfied: filelock==3.0.12 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 5))
Requirement already satisfied: flake8==3.8.4 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 6))
Requirement already satisfied: importlib-metadata==3.7.2 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 7))
Requirement already satisfied: importlib-resources==5.1.2 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 8))
Requirement already satisfied: iniconfig==1.1.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 9))
Requirement already satisfied: mccabe==0.6.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 10))
Requirement already satisfied: packaging==20.9 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 11))
Requirement already satisfied: pluggy==0.13.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 12))
Requirement already satisfied: py==1.10.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 13))
Requirement already satisfied: pycodestyle==2.6.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 14))
Requirement already satisfied: pyflakes==2.2.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 15))
Requirement already satisfied: pyparsing==2.4.7 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 16))
Requirement already satisfied: pytest==6.2.2 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 17))
Requirement already satisfied: pytest-cov==2.11.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 18))
Requirement already satisfied: six==1.15.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 19))
Requirement already satisfied: toml==0.10.2 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 20))
Requirement already satisfied: typing-extensions==3.7.4.3 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 21))
Requirement already satisfied: virtualenv==20.4.2 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 22))
Requirement already satisfied: zipp==3.4.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 23))
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Note: Since I'm already in my virtual environment it says the requirements are already satisfied but for others it will start downloading that package and installing it if that package is not already installed.

To exit the virtual Environment use command:

- deactivate

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ deactivate
redbot@redbot-G3-3500:~/PythonProjects/Calculator$
redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Since this will be a python terminal-based calculator, I am using ubuntu's editor, a text editor to write source code and test cases. First, I created a python file called Scientific_Calculator.py which contains the main functions of the calculator. The reason for this is that while testing, we will not have to input from stdin and there will not be errors or warnings during testing.

Open

*Scientific_Calculator.py
~/PythonProjects/Calculator

```
# Scientific Calculator functions
import math
import logging
logging.basicConfig(filename='Calculator.log', level=logging.DEBUG, format='%(asctime)s:%(levelname)s:%(message)s')

# Functions For the operations
def square_root(num):
        # GEtting Exception in case of taking sqare root of Negative numbers
        if num >= 0:
                ans = math.sqrt(num)
                logging.debug('Square Root of {} = {}'.format(num, ans))
                return ans
        else:
                # Prevent Exception : ValueError: math domain error
                # Occured due to taking sqare root of negative numeber
                # num == num
                print("Cannot calculate Sqaure root of Negative numbers")
                logging.debug('Square Root of {} = Cannot be caculated'.format(num))
                return float(num)

def factorial(num):
        temp = float(num)
        # TO prevent ValueError: factorial() not defined for negative values EXCEPTION
        if temp >= 0:
                # print(temp, "    ", round(temp))
                if temp == float(round(temp)):
                        ans = math.factorial(float(temp))
                        logging.debug('Factorial of {} = {}'.format(temp, ans))
                else:
                        ans = round(temp)
                        print("Cannot take Factorial of Decimal Numbers, SO we converted It to INT")

                        logging.debug('Factorial of {} = Cannot be calculated as it is Decimal number'.format(temp))
        else:
                ans = 0.0
                print("Factorial of negative number cannot be taken")
                logging.debug('Factorial of {} = Cannot be taken as it is negative number'.format(num))
        return ans

def natural_log(num):
        temp = float(num)

        if(temp > 0):
                temp = math.log(float(temp))
                logging.debug('Natural Log of {} = {}'.format(num, temp))
                #return temp
        else:
                temp = 0
                logging.debug('Natural Log of {} = Cannot be taken as it is wither Zero or Negative number'.format(num))
        return temp

def power(num1, num2):
        ans = (num1 ** num2)
        logging.debug('{} to power of {} = {}'.format(num1, num2, ans))
        return ans
```

Fig 4: Scientific_Calculator functions

Then in a separate python file, we write the driving code.

```python
import Scientific_Calculator
import os

os.system('cls' if os.name == 'nt' else 'clear')

while True:
        print("Select Operation.")
        print("1: Sqaure Root - √x")
        print("2: Factorial - x!")
        print("3: Natural Log - ln(x)")
        print("4: Power - x^y")
        print("e: Exit")

        operation = input("Enter your choice : ")

        if operation in ('1', '2', '3', '4'):
                num = float(input("Enter the number: "))

                if operation == '1':
                        print("Answer : ", Scientific_Calculator.square_root(num))

                elif operation == '2':
                        print("Answer : ", Scientific_Calculator.factorial(num))

                elif operation == '3':
                        print("Answer : ", Scientific_Calculator.natural_log(num))

                elif operation == '4':
                        exp = float(input("Enter the power : "))
                        print("Answer : ", Scientific_Calculator.power(num, exp))
                # Not using Clear screen here as it will also clear the result as well
                # os.system('cls' if os.name == 'nt' else 'clear')

        elif operation == 'e' or 'E':
                break

        else:
                print("Please Enter a valid Input")
        print("\n")

```

Fig 5 : Code of Caluclator_main.py which uses the Scientific Calculator functions

Testing:

For unit tests in python, we use pytest. A unit test is designed to check a single function, or unit, of code. Pytest supports execution of unittest test cases. The real advantage of pytest comes by writing pytest test cases. pytest test cases are a series of functions in a Python file starting with the name test_. To install pytest, we simply do

- pip install -U pytest

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip3 install -U pytest
Requirement already up-to-date: pytest in ./venv/lib/python3.6/site-packages
Requirement already up-to-date: iniconfig in ./venv/lib/python3.6/site-packages (from pytest)
Requirement already up-to-date: toml in ./venv/lib/python3.6/site-packages (from pytest)
Requirement already up-to-date: attrs>=19.2.0 in ./venv/lib/python3.6/site-packages (from pytest)
Collecting importlib-metadata>=0.12; python_version < "3.8" (from pytest)
  Downloading https://files.pythonhosted.org/packages/80/5d/0bbca82b16e01313cf0343167d4cfb90f6fade747cd4d10d368094b2883a/importlib_metadata-3.7.3-py3-none-any.whl
Requirement already up-to-date: packaging in ./venv/lib/python3.6/site-packages (from pytest)
Requirement already up-to-date: py>=1.8.2 in ./venv/lib/python3.6/site-packages (from pytest)
Requirement already up-to-date: pluggy<1.0.0a1,>=0.12 in ./venv/lib/python3.6/site-packages (from pytest)
Requirement already up-to-date: typing-extensions>=3.6.4; python_version < "3.8" in ./venv/lib/python3.6/site-packages (from importlib-metadata>=0.12; python_version < "3.8"->pytest)
Requirement already up-to-date: zipp>=0.5 in ./venv/lib/python3.6/site-packages (from importlib-metadata>=0.12; python_version < "3.8"->pytest)
Requirement already up-to-date: pyparsing>=2.0.2 in ./venv/lib/python3.6/site-packages (from packaging->pytest)
Installing collected packages: importlib-metadata
  Found existing installation: importlib-metadata 3.7.2
    Uninstalling importlib-metadata-3.7.2:
      Successfully uninstalled importlib-metadata-3.7.2
Successfully installed importlib-metadata-3.7.3
```

pytest has some other great features:

- Support for the built-in assert statement instead of using special self.assert*() methods
- Support for filtering for test cases
- Ability to rerun from the last failing test
- An ecosystem of hundreds of plugins to extend the functionality

A standard practice that goes hand in hand with testing is calculating code coverage. Code coverage is the percentage of source code that is “covered” by your tests. pytest has an extension, pytest-cov, that helps us understand the code coverage. To install this, we do pip install pytest-cov. Now we can run the tests.

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ pip install pytest-cov
Requirement already satisfied: pytest-cov in ./venv/lib/python3.6/site-packages
Requirement already satisfied: pytest>=4.6 in ./venv/lib/python3.6/site-packages (from pytest-cov)
Requirement already satisfied: coverage>=5.2.1 in ./venv/lib/python3.6/site-packages (from pytest-cov)
Requirement already satisfied: importlib-metadata>=0.12; python_version < "3.8" in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: pluggy<1.0.0a1,>=0.12 in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: toml in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: iniconfig in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: packaging in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: attrs>=19.2.0 in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: py>=1.8.2 in ./venv/lib/python3.6/site-packages (from pytest>=4.6->pytest-cov)
Requirement already satisfied: typing-extensions>=3.6.4; python_version < "3.8" in ./venv/lib/python3.6/site-packages (from importlib-metadata>=0.12; python_version < "3.8"->pytest>=4.6->pytest-cov)
Requirement already satisfied: zipp>=0.5 in ./venv/lib/python3.6/site-packages (from importlib-metadata>=0.12; python_version < "3.8"->pytest>=4.6->pytest-cov)
Requirement already satisfied: pyparsing>=2.0.2 in ./venv/lib/python3.6/site-packages (from packaging->pytest>=4.6->pytest-cov)
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Writing the TestSum test case example for pytest would look like this, It will consist of both positive to false negative results as well:

```python
"""
Unit tests for the calculator library
"""
# Import the Scientific Calculator Function File

import Scientific_Calculator


class TestCalculator:

    def test_sqrt(self):
        assert Scientific_Calculator.square_root(196) == 14.0
        assert Scientific_Calculator.square_root(0) == 0.0
        assert Scientific_Calculator.square_root(-9) == -9.0
        assert Scientific_Calculator.square_root(1) == 1
        assert Scientific_Calculator.square_root(3) == 1.7320508075688772

    def test_factorial(self):
        assert Scientific_Calculator.factorial(7) == 5040
        assert Scientific_Calculator.factorial(5) == 120
        assert Scientific_Calculator.factorial(0) == 1
        assert Scientific_Calculator.factorial(1) == 1
        assert Scientific_Calculator.factorial(-5) == 0.0
        assert Scientific_Calculator.factorial(-5.5) == 0.0
        assert Scientific_Calculator.factorial(5.5) == 6

    def test_log(self):
        assert Scientific_Calculator.natural_log(0) == 0
        assert Scientific_Calculator.natural_log(-1) == 0
        assert Scientific_Calculator.natural_log(-10) == 0
        assert Scientific_Calculator.natural_log(2.7182818284590 45) == 1.0
        assert Scientific_Calculator.natural_log(20) == 2.995732273553991

    def test_power(self):
        assert Scientific_Calculator.power(2,9) == 512
        assert Scientific_Calculator.power(0, 12) == 0
        assert Scientific_Calculator.power(20, 0) == 1
        assert Scientific_Calculator.power(20, 1) == 20.0
        assert Scientific_Calculator.power(2, 3) == 8.0
        assert Scientific_Calculator.power(-2,3) == -8.0
        assert Scientific_Calculator.power(2,-3) == 0.125
        assert Scientific_Calculator.power(-2, -3) == -0.125
        assert Scientific_Calculator.power(-2, 4) == 16.0
        assert Scientific_Calculator.power(1, 23) == 1.0

```

Fig 6: Unit Test cases for scientific calculator

We are using the assert keyword to write our test cases. It lets us test the condition whether both sides give equal results (or same), else it will raise an AssertionError.

After creating the unit test cases, it's time to check whether our code passes out those test cases or not. To check run command :

- pytest -v --cov=<Name of main file> [Like i runned pytest -v --cov=Scientific_Calculator as 'Scientific_Calculator.py' is name of file which holds the functions of my calculator].

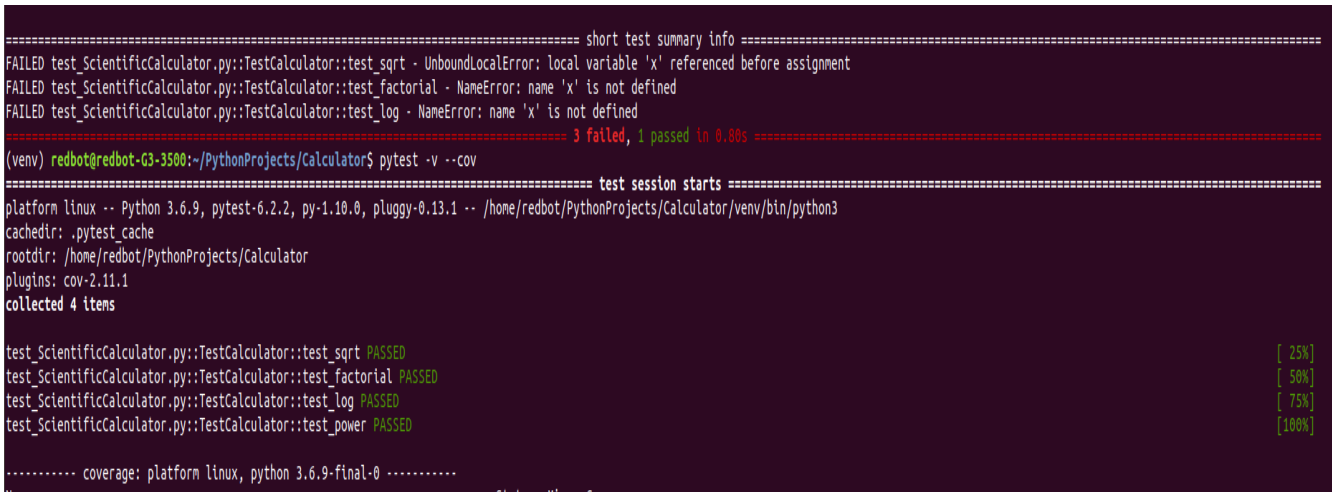


Fig 7: Pytest results

Continuous Integration:

One of the most important part of Project. Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day. In this development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. We can use CircleCI for Continuous Integration or continue till Continuous Delivery using the pipeline project.

The steps we did above were all manual. What if we want several developers to add hundreds of features every day, doing all these steps manually? Think about the time we would lose in this labor. And enter Continuous Integration. The idea is to automate the build. Now python is an interpreted language so it's 'builds' revolve around test execution, rather than compilation.

The point of this step is to have everyone working on a known stable base. When the build is automated, we are encouraged to commit frequently, usually multiple times per day. It allows people to quickly find out about changes and notice if there's a conflict between two developers. If there are numerous small changes instead of a few massive updates, it's much easier to locate where the error originated. It will also encourage you to break your work down into smaller chunks, which is easier to track and test.

Step 1: To use CircleCI we need to go to : https://circleci.com/ , then create an account there.
Step 2: Connect CircleCI account to Github or Bitbucket wherever you store your code.
Step 3: Then go to Projects > Setup Projects > setup "config.yml".
Now, Whenever we push change code in the github repository it will get build and tested.

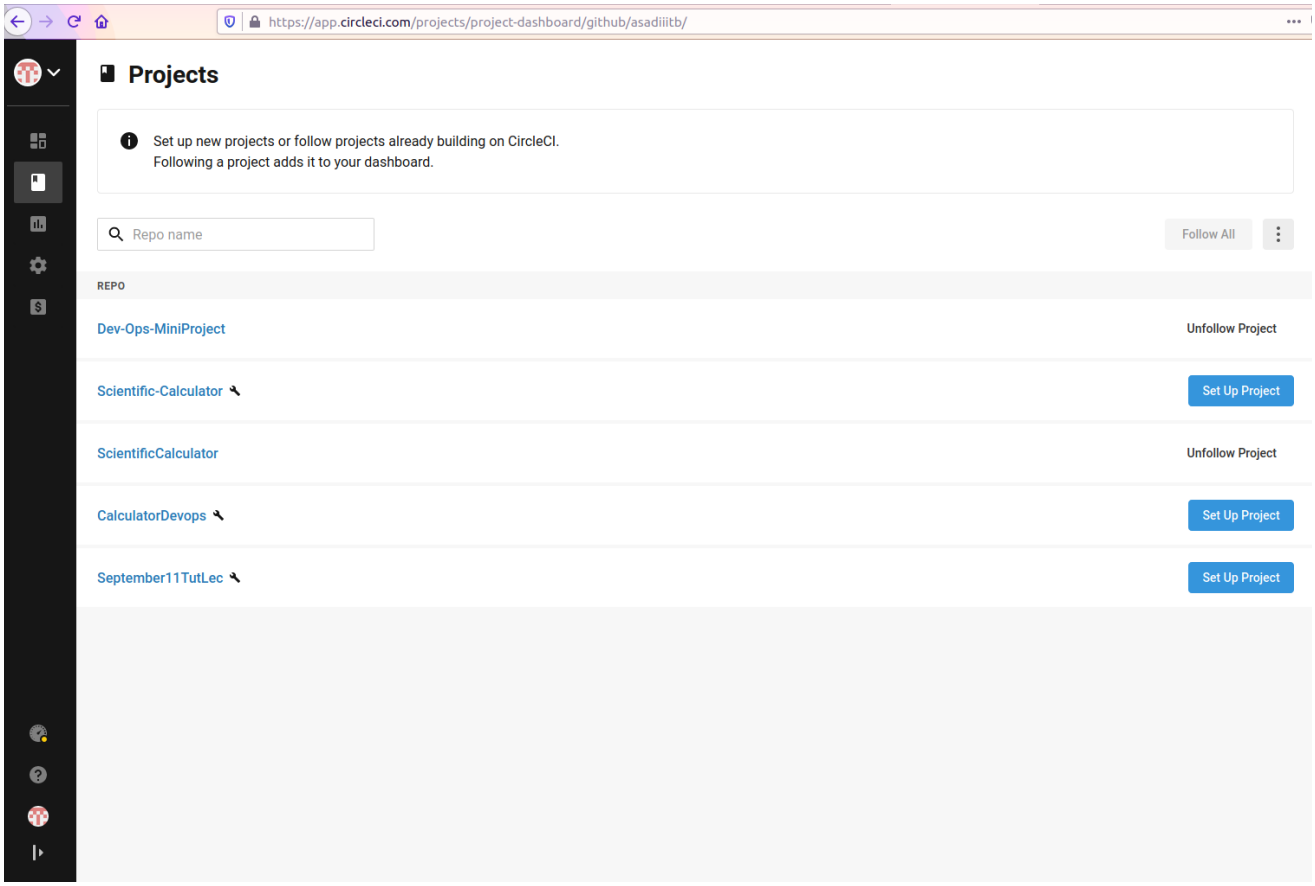


Fig 8 : CircleCI projects page of your account (already connected with Github or Bitbucket)

Before setting up a project in CircleCI we need to write the **config.yml** file in which we write the steps and commands instructing CircleCI server how to build and test our code.

1. The first step is to create a .circleci folder in our project directory. Inside this .circleci folder, we create a config.yml file. So its structure will be like ./Parent Repo/.circleci/config.yml
2. Now in the config.yml file, we need to define or state the following steps.

```yaml
# Python CircleCI 2.1 configuration file
version: 2.1
jobs:
  build:
    # executor: docker-publisher
    environment:
      IMAGE_NAME: devops_calculator
    docker:
      - image: circleci/python:3.7
    working_directory: ~/repo

    steps:
      # Step 1: obtain repo from GitHub
      - checkout
      # Step 2: setup remote docker engine
      - setup_remote_docker
      # Step 3: create virtual env and install dependencies
      - run:
          name: install dependencies
          command: |
            python3 -m venv v_env
            . v_env/bin/activate
            pip install -r requirements.txt
      # Step 3: run linter and tests
      - run:
          name: run tests
          command: |
            . v_env/bin/activate
            pytest -v --cov=Scientific_Calculator
      # Step 4: Build Docker Image
      - run:
          name: Build and Push Docker Image
          command: |
            docker build -t $IMAGE_NAME:latest .
            echo "$DOCKERHUB_PASSWORD" | docker login -u "$DOCKERHUB_USERNAME" --password-stdin
            docker tag $IMAGE_NAME $DOCKERHUB_USERNAME/$IMAGE_NAME:latest
            docker push $DOCKERHUB_USERNAME/$IMAGE_NAME:latest

```

Fig 9: Config.yml file code

Ignoring the docker part for now, we see the steps as:

- Checkout - This checks our github repo for the latest version of our code.
- Install dependencies – This creates a virtual environment and installs the packages from requirements.txt which we used in our local machine.
- Run tests – This runs unit tests in the similar fashion as we did it locally.

Now, we add this .circleci folder to our github repository. Back to setting up our CircleCI server.

We select our project repo and set up the project. Once we click on set up project, CircleCI automatically fetches the config.yml from the .circleci folder in our github repository.

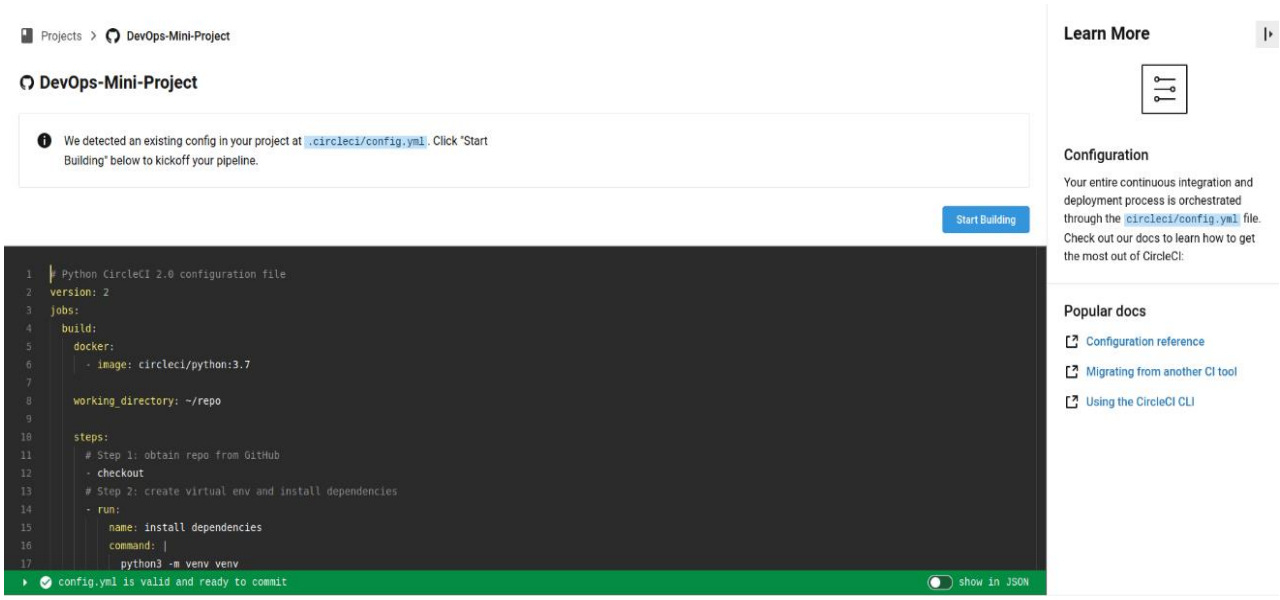


Fig 10: CircleCI fetched config.yml, presenting it and ready to start building

Now we will click on Start Building Button

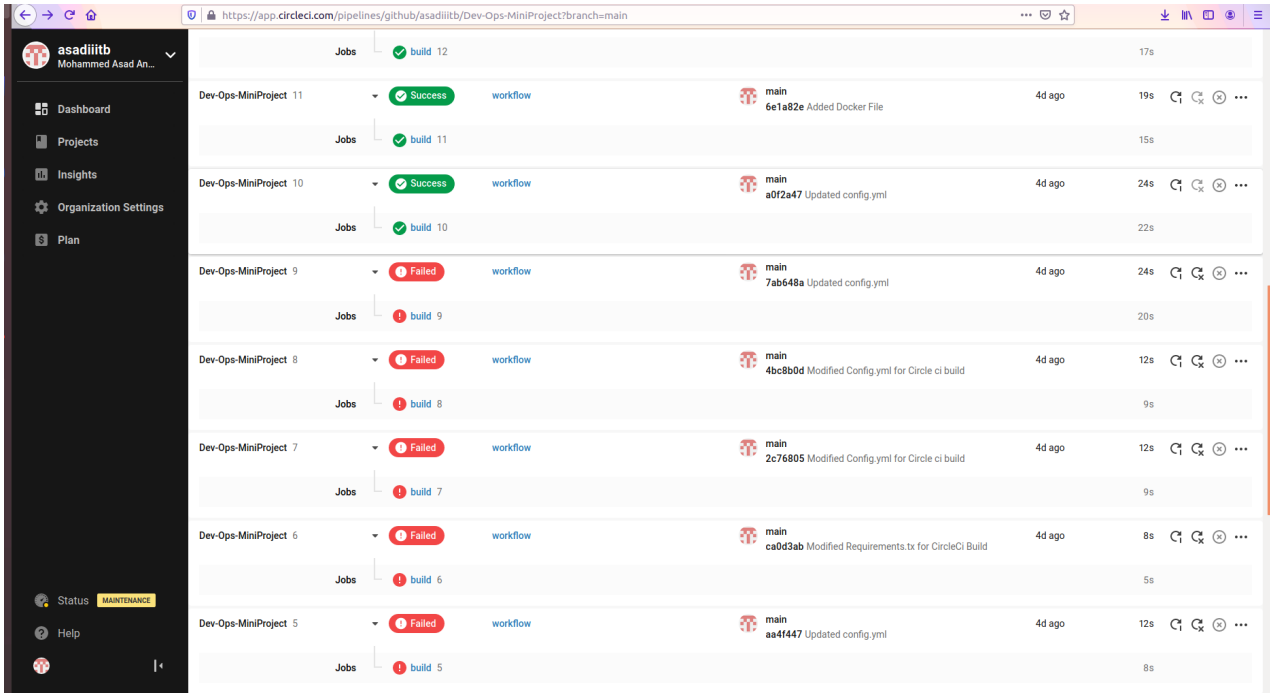


Fig 11: CircleCI builds success and fails pipeline

As it can be seen my certain number of builds failed as there were some virtual environment naming conflicts, also there were build fails because of spacing and indentation, as it is sensitive to spacing and indentation. But finally at the end it all worked

Workflow of CircleCI server in this successful build:

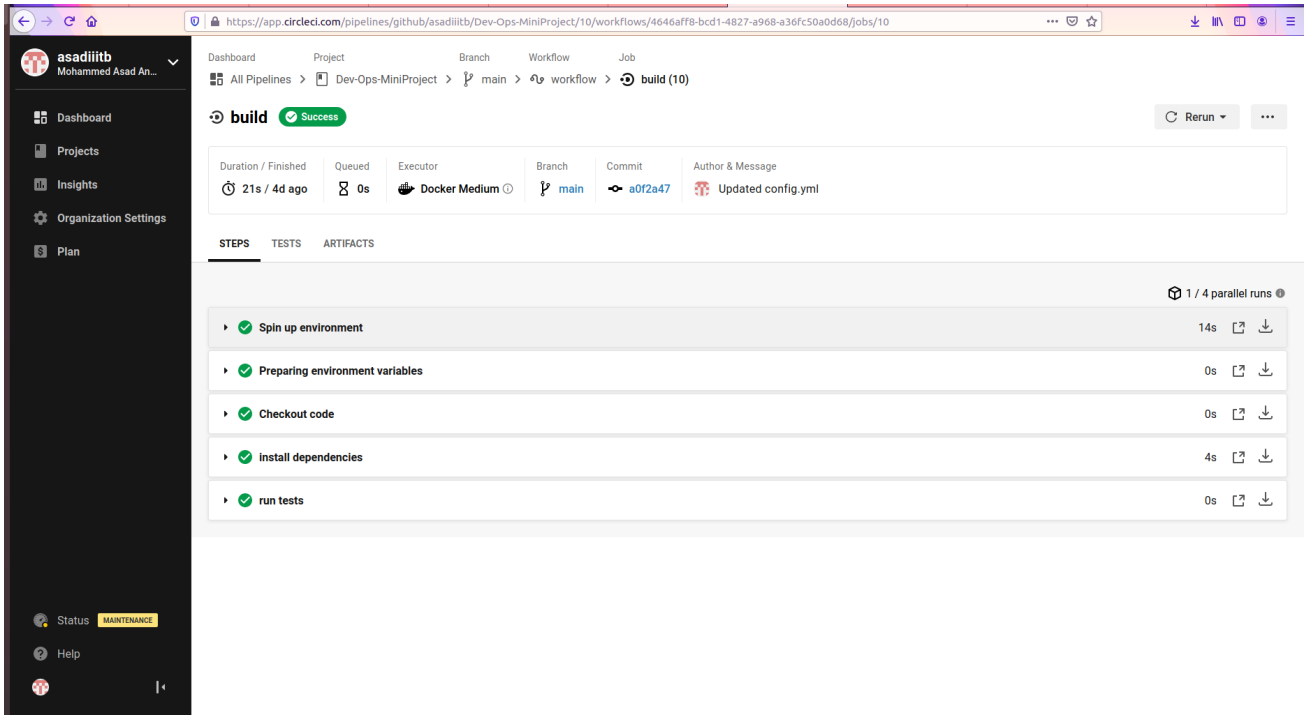


Fig 12: All steps of CircleCI build and test

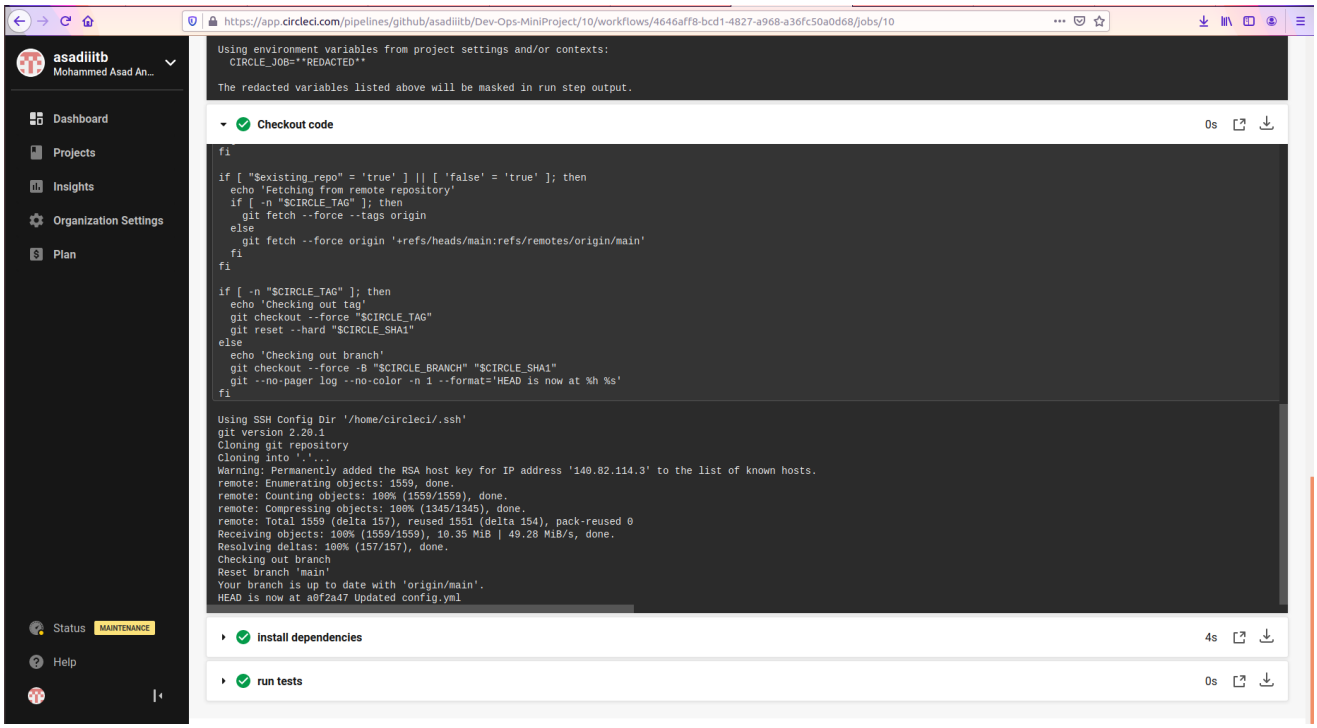


Fig 13: Checksout - pulls the latest code from the Github repo

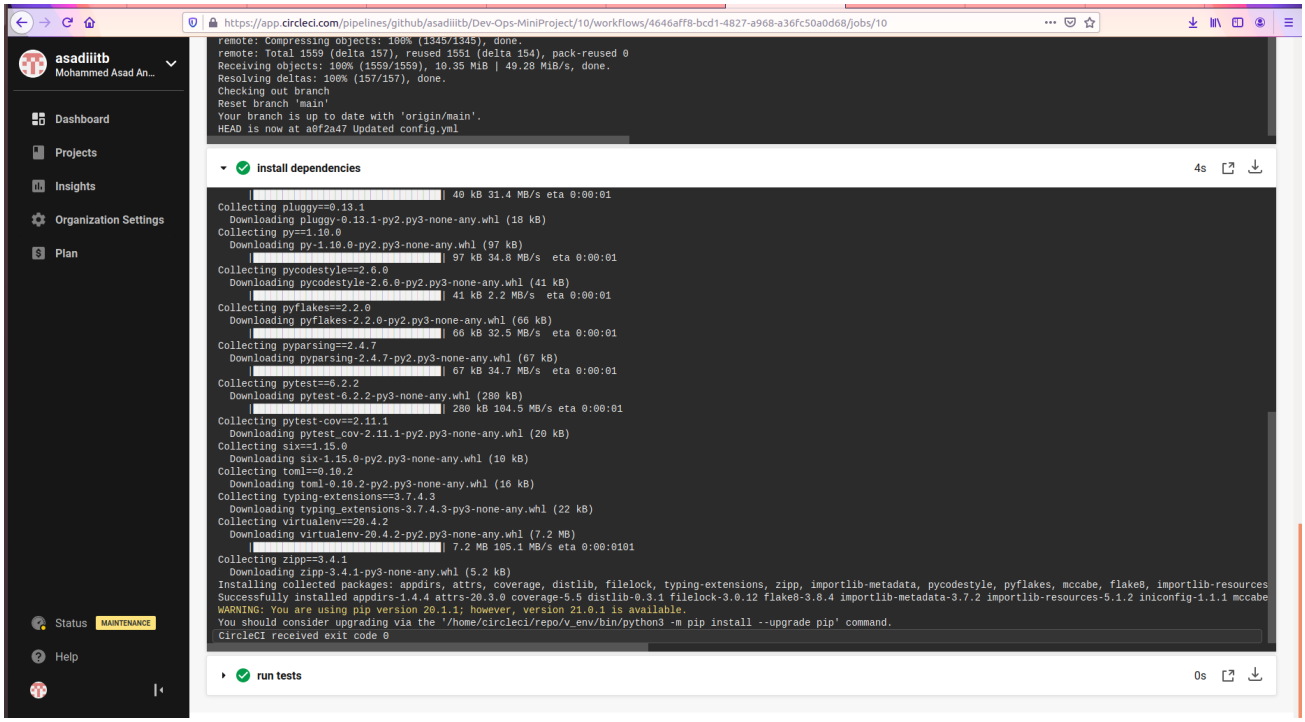


Fig 14: Installs dependencies, with the help of requirements.txt

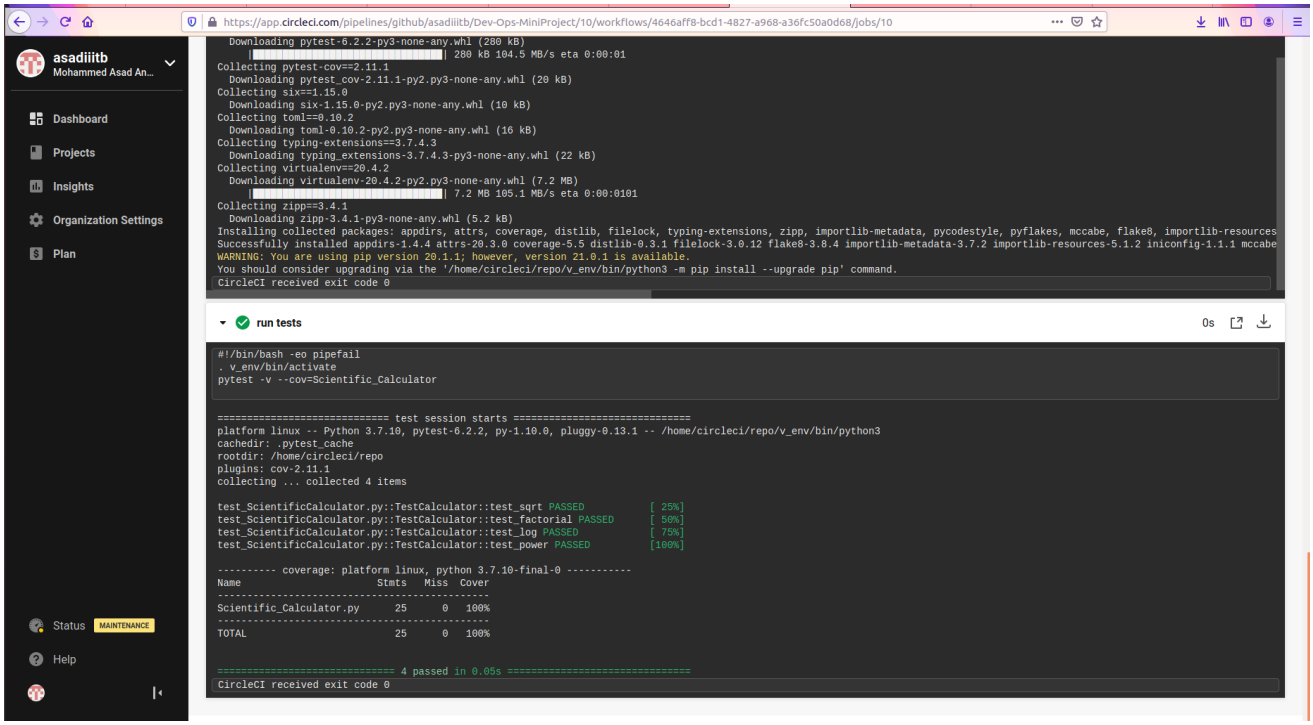


Fig 15: Runs Unit tests

One of features of CircleCI, as we push the code or made changes in repository it checks them and builds, and whatever is the result it sends the report / result to the developer via email, so developer dont need to look at the pipeline instantly and wait for result.

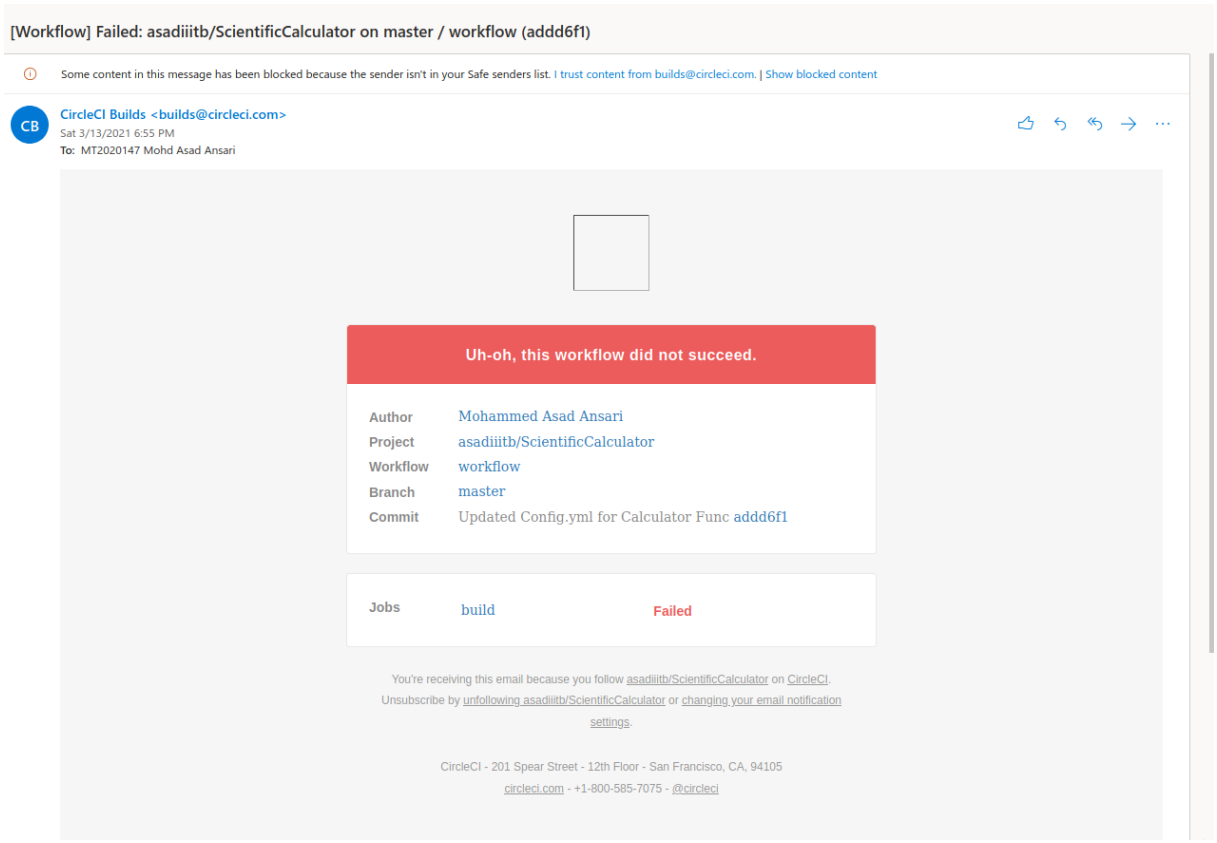


Fig 16: CircleCI build fail notification via mail

Docker:

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

The Docker platform:

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share

containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Installing Docker:

First things first, let's install docker engine on our local machine. Since I was using Ubuntu at the moment of this project, I followed this guide How to Install Docker on Ubuntu.
{ Keep in mind docker image need to be run in su mode so we need to use **sudo** with all the commands }.
Need of Docker : We simply want to create a docker image of our calculator application which can run on any Linux machine (Note that since I am working on Linux, the docker image created will be a Linux based image and cannot be run on Windows).

DOCKERFILE:

To build our docker image, we first need to make a Dockerfile in our project's top level directory.
A **Dockerfile** is a **text file** written in an easy-to-understand syntax that includes the instructions to build a Docker image. It specifies the operating system that will underlie the container, along with the languages, environmental variables, file locations, network ports, and other components it needs—and, of course, what the container will be doing once we run it.
Let's write the docker file for our calculator application:

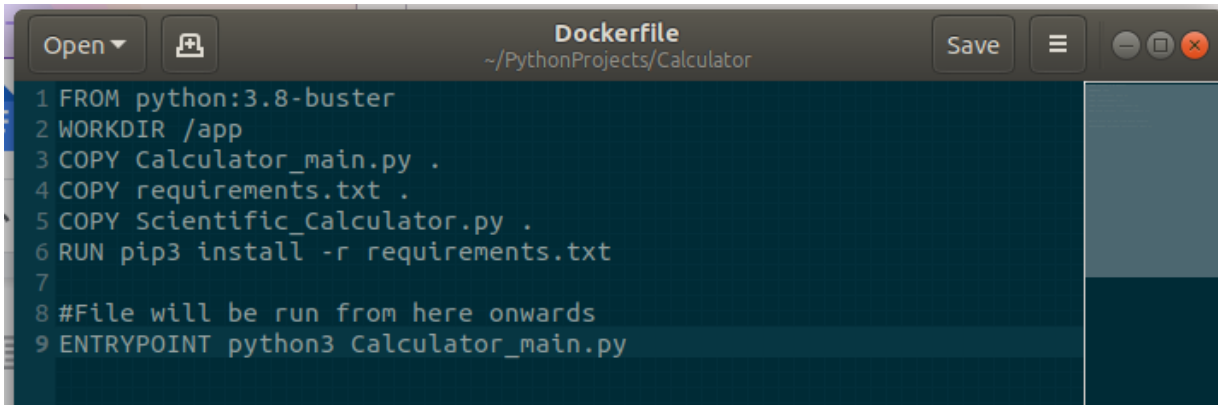


Fig 17 : Dockerfile

{Keep in mind u need to save this file as “Dockerfile”, I saved it in Camel Case and wasted 2-3 hrs to solve the error before realizing}
Since, it’s easy to understand let’s try to get what each line means:

1. **FROM python:3.8-buster** - This line specifies that from the python repository at Docker Hub, use the version 3.8. There can be multiple versions available for an image and we are free to choose at our own will.
2. **WORKDIR /app** - Now, we create a working directory. This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.
3. **COPY Calculator_main.py .** - The COPY command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We will copy the Calculator_main.py file into our working directory /app. Similarly, we do it for our remaining .py files and requirements.txt.
4. **RUN pip3 install -r requirements.txt** - Next, we can use the RUN command to execute the command pip3 install. This works the same as if we were running pip3 install locally on our machine, but this time the modules are installed into the image.
5. **ENTRYPOINT python3 Calculator_main.py** - This tells the Docker what to do with the image. In our case, we want to run our Calculator_main.py using the command **python3 Calculator_main.py.**

DOCKER IMAGE BUILD:

With our Dockerfile ready, we are ready to build our image. By command:

- **sudo docker build -t <Name/tag of image generated by Dockerfile>**

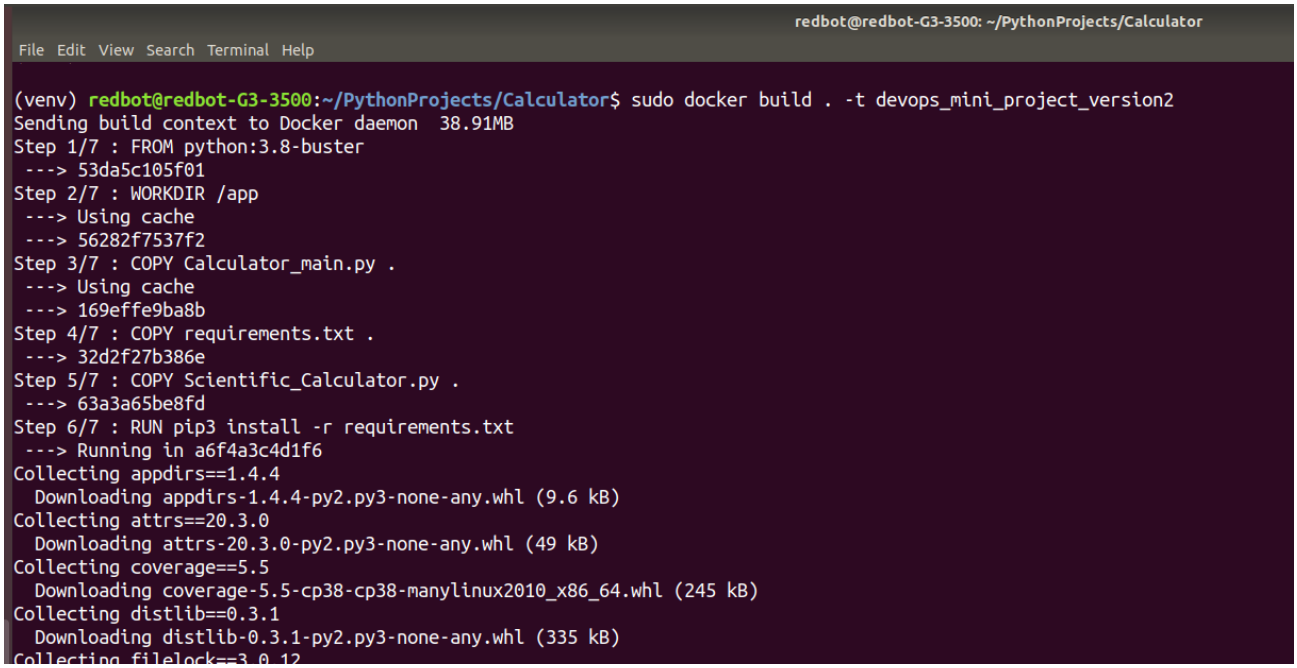


Fig 18: Building Docker image from Dockerfile

Now check in docker images on your system the generated image using command:

- **sudo docker images**

Fig 19: Build image “devops_mini_project_version2” can be seen in docker images.

As we have successfully built the docker image for our terminal based calculator. Let’s run it by command :

- Sudo docker run -i <tag/name of Image> {-i tag is used to run the image in an interactive terminal such that it expects a user}.

```
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$ sudo docker run -i devops_mini_project
TERM environment variable not set.
Select Operation.
1: Sqaure Root - √x
2: Factorial - x!
3: Natural Log - ln(x)
4: Power - x^y
e: Exit
Enter your choice : 2
Enter the number: 6
Answer :  720


Select Operation.
1: Sqaure Root - √x
2: Factorial - x!
3: Natural Log - ln(x)
4: Power - x^y
e: Exit
Enter your choice : 1
Enter the number: 169
Answer :  13.0


Select Operation.
1: Sqaure Root - √x
2: Factorial - x!
3: Natural Log - ln(x)
4: Power - x^y
e: Exit
Enter your choice : e
(venv) redbot@redbot-G3-3500:~/PythonProjects/Calculator$
```

Fig 20: Running docker image

Now we are ready to push our image to our Docker Hub repository using below steps:

1. Login into Docker Hub from command line - docker login --username=yourhubusername --email=youremail
2. Attaching a tag to our docker image:

```
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ ls
og       Dockerfile           'Logs last 5 hours.png'   README.md         Scientific_Calculator.py       venv
ain.py  'Logs last 3 days.png'   __pycache__             requirements.txt   test_ScientificCalculator.py
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ sudo docker tag 69s547321xa2 imasadansari/devops_mini_project:firsttry
```

3. Pushing the image to our repository:

```
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ sudo docker push imasadansari/devops_mini_project:firsttry
```

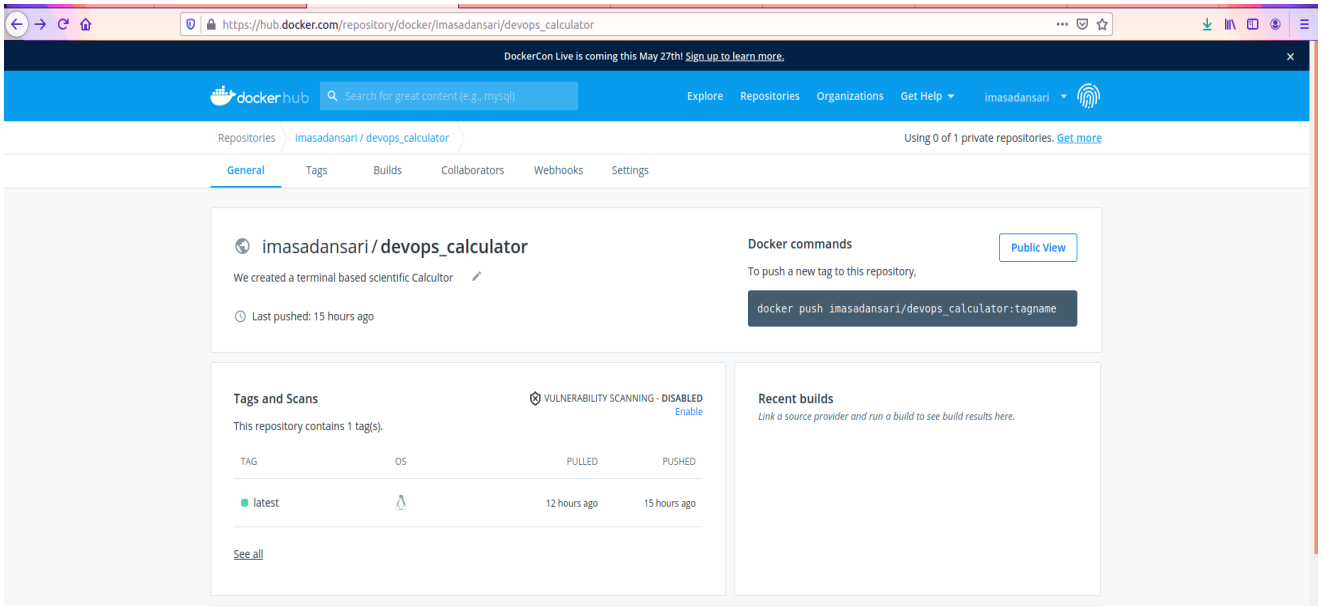
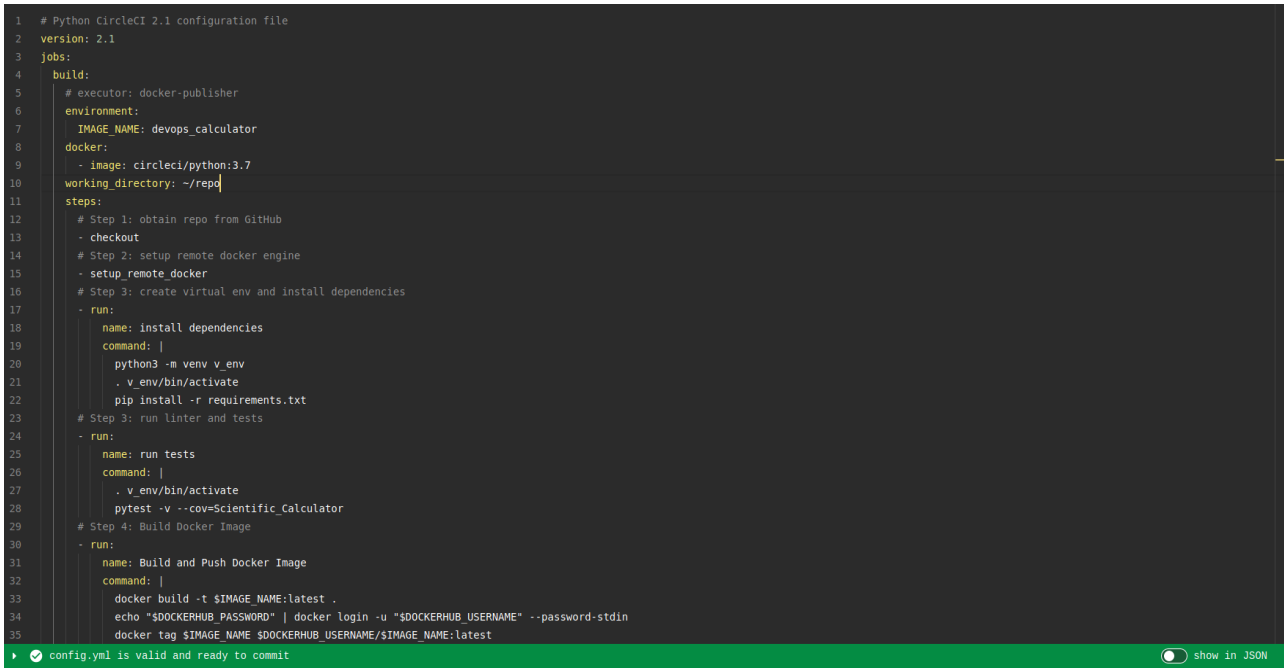


Fig 21: Docker hub

As we can see, our image was pushed to Docker Hub from our local machine. This is great but we do not want to manually build and push every time we make some changes or add new features. Once again, we will go with CircleCI for automating this.

Docker Hub and CircleCI Automation builds:

Since we have already written the Dockerfile and successfully pushed the image on Docker Hub, half of our work is already done.
To make CircleCI do this job for us, we need to again edit the config.yml file which was present in the .circleci folder of our project directory.



```
# Python CircleCI 2.1 configuration file
version: 2.1
jobs:
  build:
    # executor: docker-publisher
    environment:
      IMAGE_NAME: devops_calculator
    docker:
      - image: circleci/python:3.7
    working_directory: ~/repo
    steps:
      # Step 1: obtain repo from GitHub
      - checkout
      # Step 2: setup remote docker engine
      - setup_remote_docker
      # Step 3: create virtual env and install dependencies
      - run:
          name: install dependencies
          command: |
            python3 -m venv v_env
            . v_env/bin/activate
            pip install -r requirements.txt
      # Step 3: run linter and tests
      - run:
          name: run tests
          command: |
            . v_env/bin/activate
            pytest -v --cov=Scientific_Calculator
      # Step 4: Build Docker Image
      - run:
          name: Build and Push Docker Image
          command: |
            docker build -t $IMAGE_NAME:latest .
            echo "$DOCKERHUB_PASSWORD" | docker login -u "$DOCKERHUB_USERNAME" --password-stdin
            docker tag $IMAGE_NAME $DOCKERHUB_USERNAME/$IMAGE_NAME:latest
```


Fig 22 : Contains Config.yml code

In the above sections we ignored the docker part. Let's have a look at it now.

- setup_remote_docker - It tells CircleCI to allocate a new Docker Engine, separate from the environment that is running our job, specifically to execute the Docker commands.
- Build and Push Docker Image - This is the crucial part. After successful testing, we want to build our docker image and push it to Docker Hub. Note that I have used an environment variable called IMAGE_NAME which specifies the name of my docker image. Firstly, we build the docker image with the same command we used in our local machine `docker build -t $IMAGE_NAME:latest .`

Now, remember we logged into Docker Hub from our terminal when building image locally?

We need to do this via CircleCI now. But it is not safe and a secure practice to store passwords and sensitive keys in a text format simply in the config.yml file. So, we add these credentials to the environment variables in our CircleCI project.

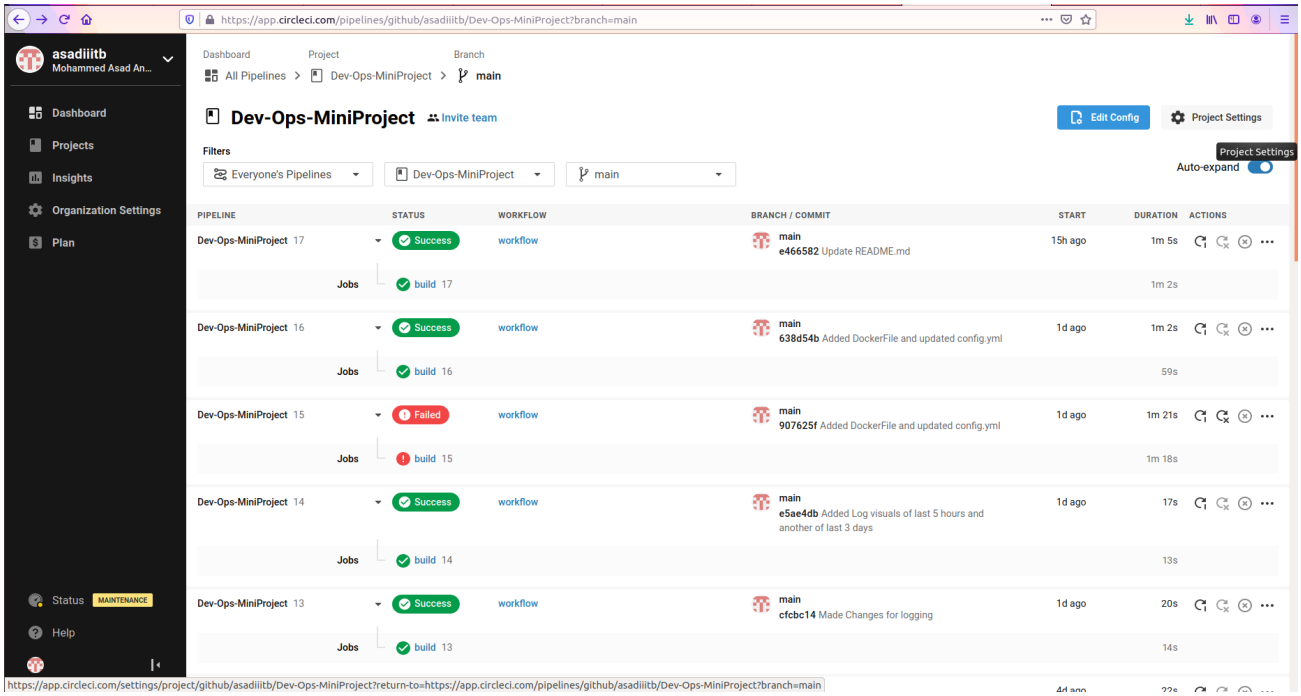


Fig : Go to Project Settings

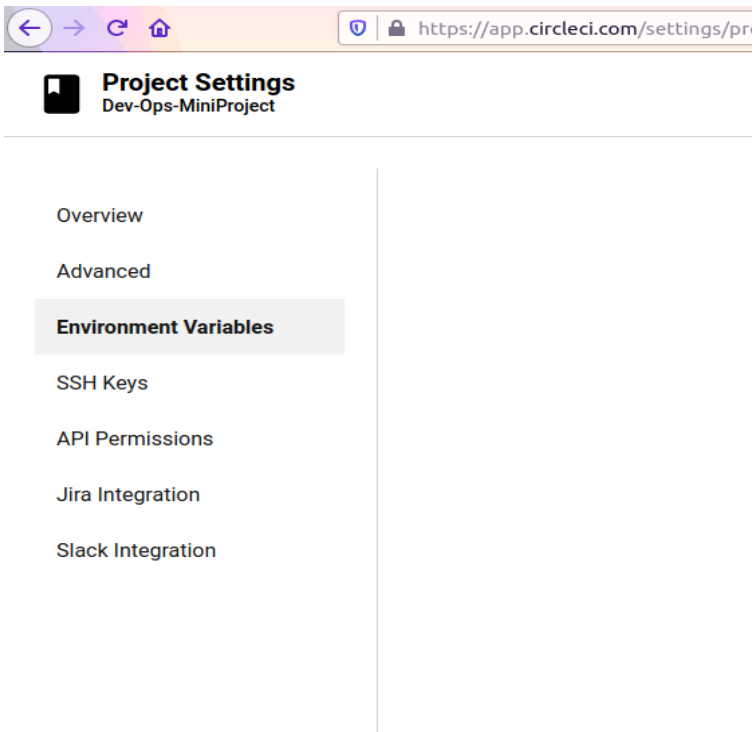


Fig : Select Environment Variables

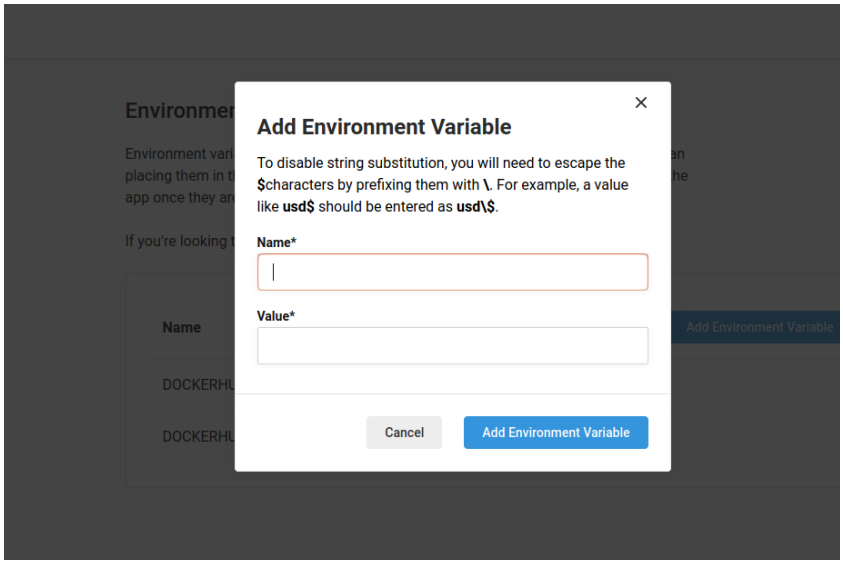
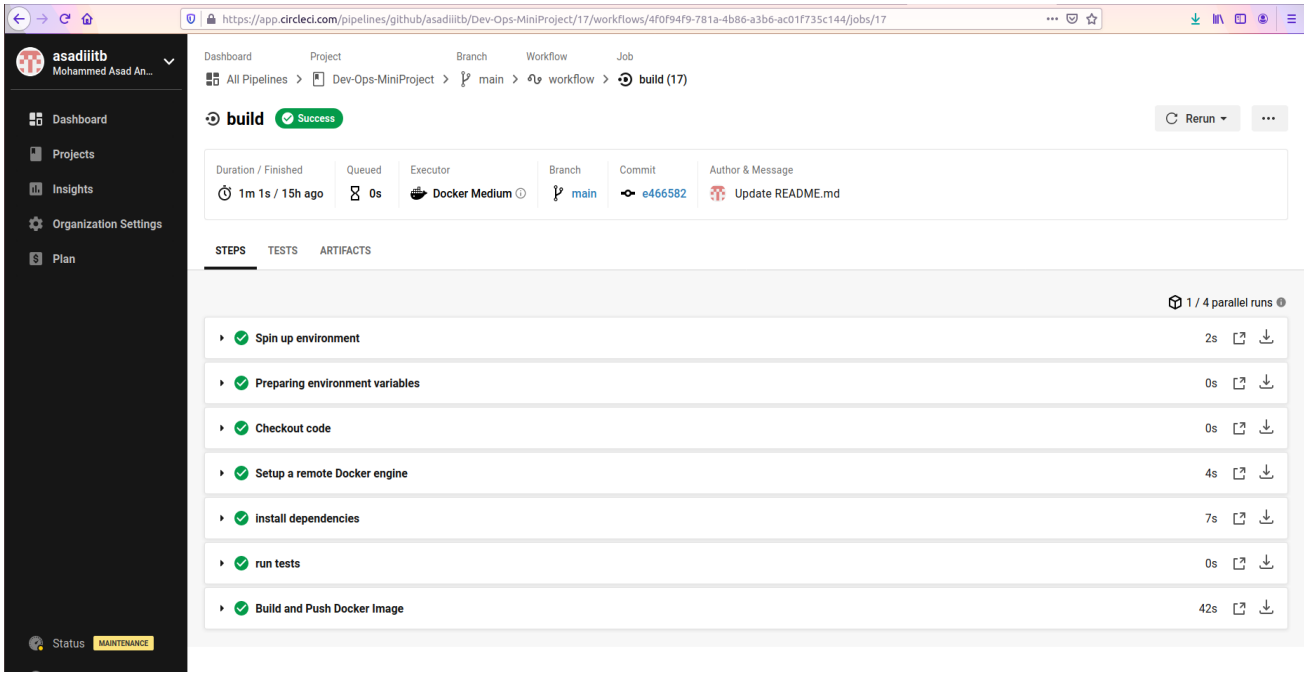


Fig : Then click on Add Environment Variable, and simply add the variable and its value.

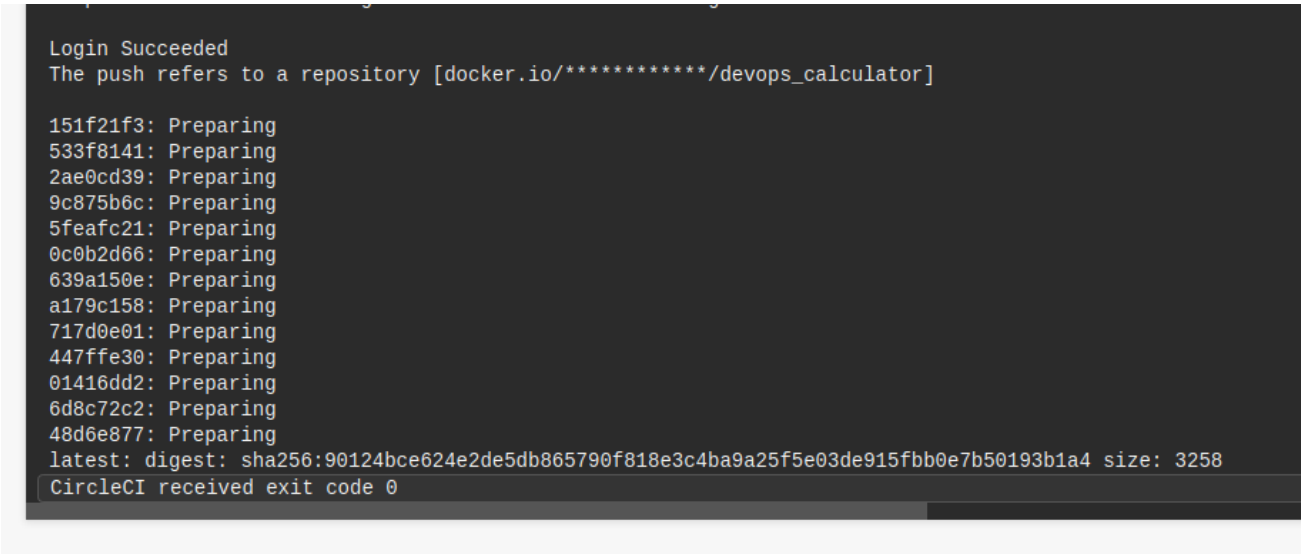
Now, we are ready to use our Docker Hub credentials safely in our config.yml file.

```
echo "$DOCKERHUB_PASSWORD" | docker login -u "$DOCKERHUB_USERNAME" --password-stdin
docker tag $IMAGE_NAME $DOCKERHUB_USERNAME/$IMAGE_NAME:latest
docker push $DOCKERHUB_USERNAME/$IMAGE_NAME:latest
```

We simply login into Docker Hub by using the environment variables (starts with '$') and follow the same procedure of tagging our image and then pushing it.
Let us now push the updated config.yml file and our Dockerfile to our GitHub repo. And then, this happens on CircleCI:



Let us checkout the Docker part. As we can see, it successfully ran our Dockerfile from Github which we pushed before. Also, the login and pushing to Docker Hub was Successful.



By doing this, we have successfully automated our calculator project. Let's just try to run this image we pushed from CircleCI on our local machine which will verify that our image will work on any linux machine:

```
redbot@redbot-G3-3500:~/PythonProjects/Calculator$ sudo docker run -i imasadansari/devops_calculator
[sudo] password for redbot:
Sorry, try again.
[sudo] password for redbot:
Unable to find image 'imasadansari/devops_calculator:latest' locally
latest: Pulling from imasadansari/devops_calculator
e22122b926a1: Already exists
f29e09ae8373: Already exists
e319e3daef68: Already exists
e499244fe254: Already exists
5a6ebed20e89: Already exists
56b703a5a371: Already exists
dec810778c22: Already exists
505aac614a24: Already exists
36deae172a15: Already exists
6042dfdeaf0f: Pull complete
b62408ca8804: Pull complete
682b113b4afd: Pull complete
096e526720b5: Pull complete
1ca281bf0b35: Pull complete
Digest: sha256:7d177ccdec73d0137d055dc4485ae49f8e84a9723fb947e834d8a30478aab037
Status: Downloaded newer image for imasadansari/devops_calculator:latest
TERM environment variable not set.
Select Operation.
1: Sqaure Root - √x
2: Factorial - x!
3: Natural Log - ln(x)
4: Power - x^y
e: Exit
Enter your choice : 3
Enter the number: 2
Answer :  0.6931471805599453


Select Operation.
1: Sqaure Root - √x
2: Factorial - x!
3: Natural Log - ln(x)
4: Power - x^y
e: Exit
Enter your choice : 4
Enter the number: 2
```

Fig : Pulling latest image pushed via CircleCI

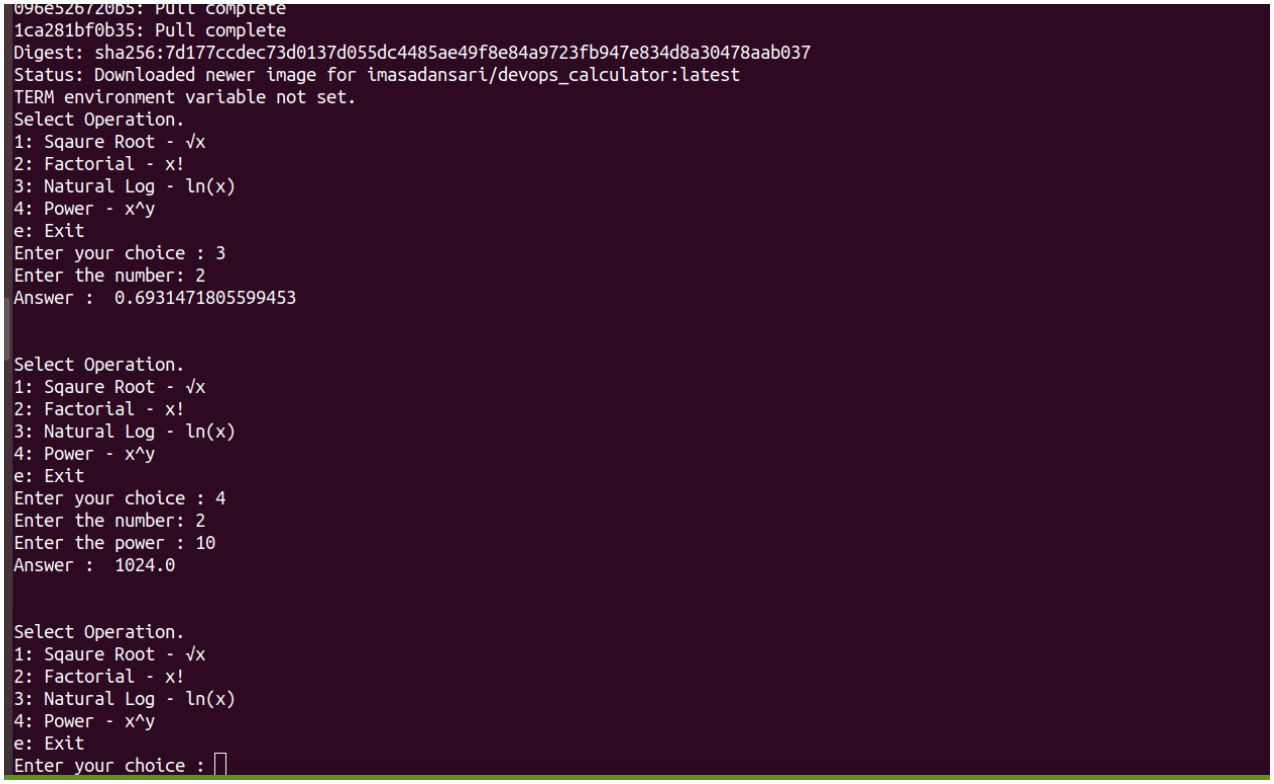


Fig : Updated image running

Logging:

Logs provide us with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information, like which user or IP accessed the application. If an error occurs, then they can provide more insights than a stack trace by telling you what the state of the program was before it arrived at the line of code where the error occurred.

Python provides a logging system as a part of its standard library, so we can quickly add logging to our application.

In Python, by default, there are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:

1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL

```
logging.basicConfig(filename='calcy.log', level=logging.DEBUG,
                    format='%(asctime)s:%(levelname)s:%(message)s')
```

For this project, we have changed the level to DEBUG, and the logs are being stored in a log file called Calculator.log in the specified format.
Remember the Scientific_Calculator.py module I created at the beginning which contained the main functions? I have added logging in those functions so that whenever these functions are called, the application will store the values entered by the user in these logs.

** Logging image of code

Now, whenever any function is called, the value entered by the user will be logged. For example:
** Log file image
After Running the application:
** Image of running code and log file

As we can see the new inputs are being recorded in log file

Monitoring:

For monitoring, we are going to use the ELK stack, mainly Kibana. Kibana is an open source browser based visualization tool mainly used to analyze large volumes of logs in the form of line graph, bar graph, pie charts, heat maps, region maps, coordinate maps, etc. The visualization makes it easy to predict or to see the changes in trends of errors or other significant events of the input source.
First, we setup our 14 day free trial account on www.elastic.co

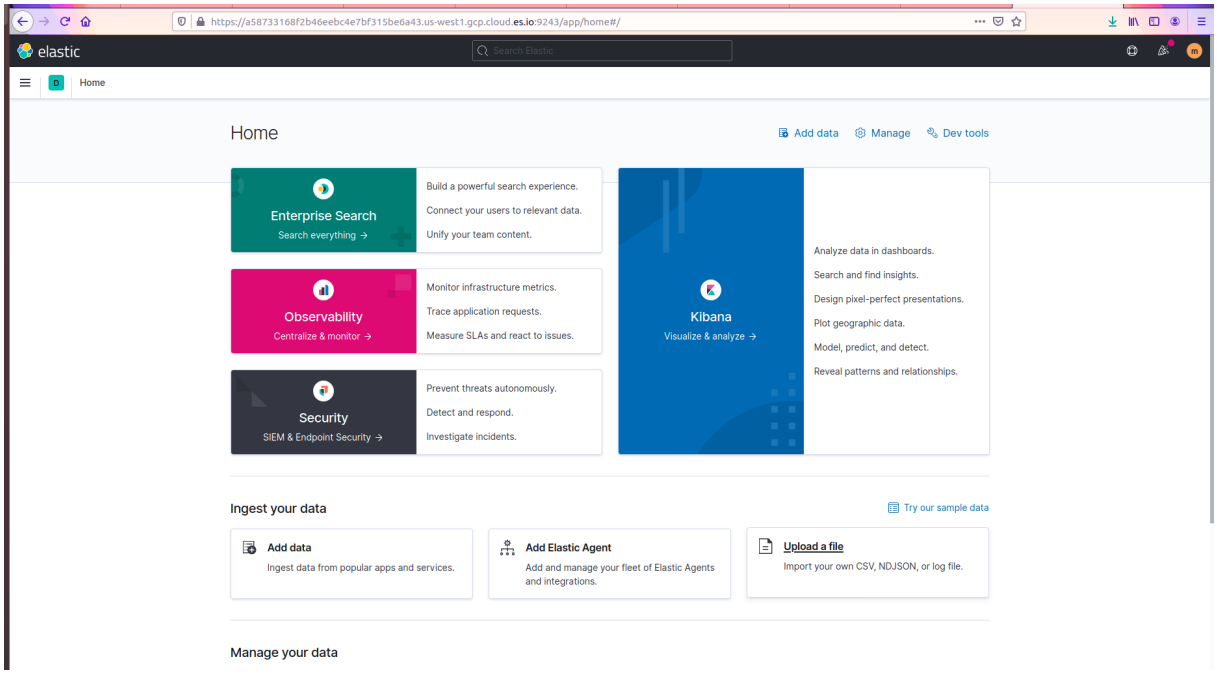
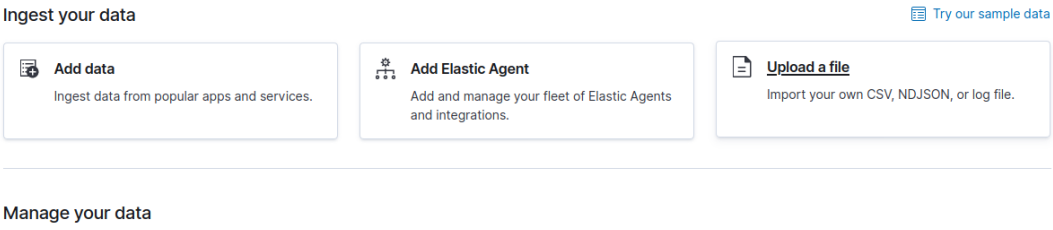
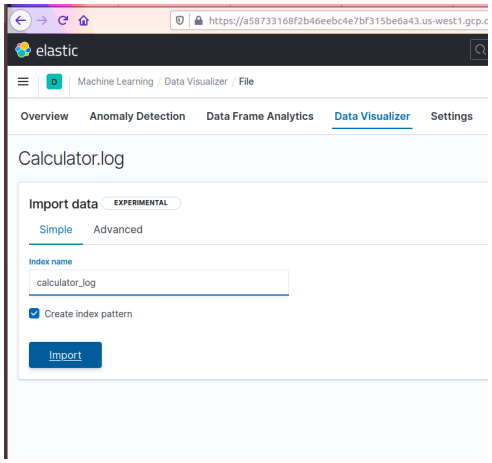


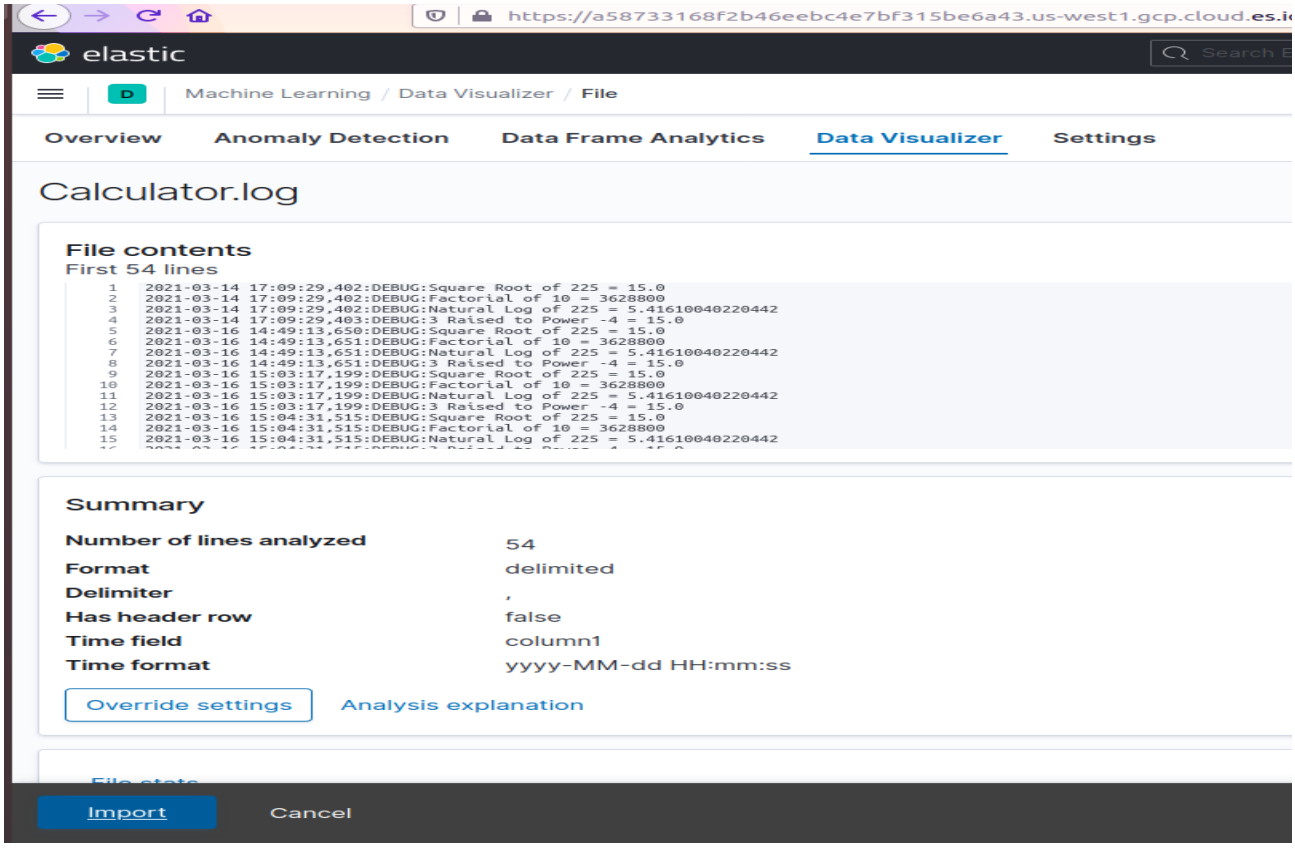
Fig: ELK Dashboard

Next, we upload the log file generated by our calculator python application.

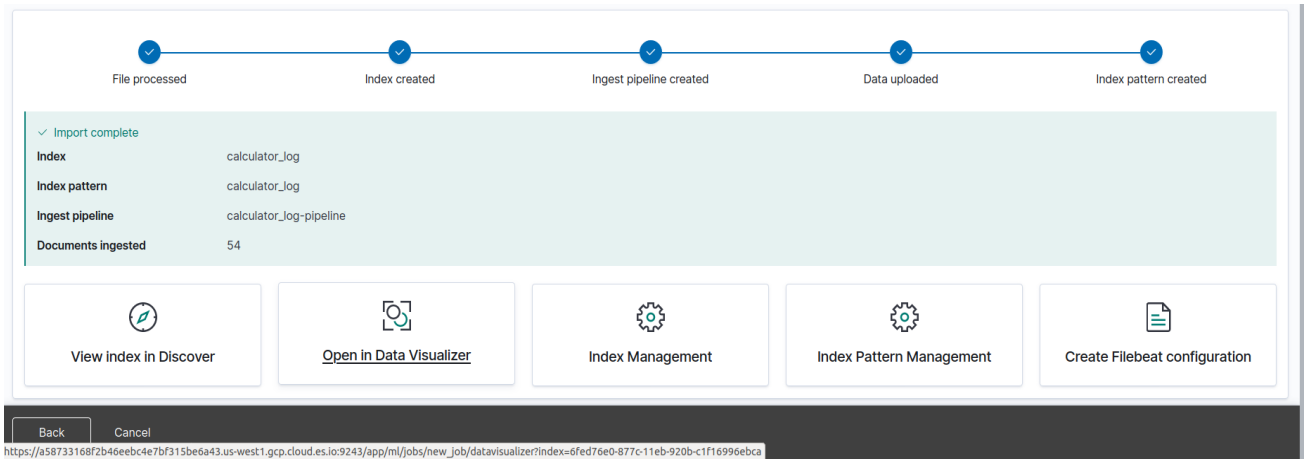


Click on Import and add an index name.





If the visualization build is successful, we will see something like this



Let’s open our log file in data visualizer.

Note that we can change have multiple indexes and we can refine the graph by the time from which we want to see the logs.

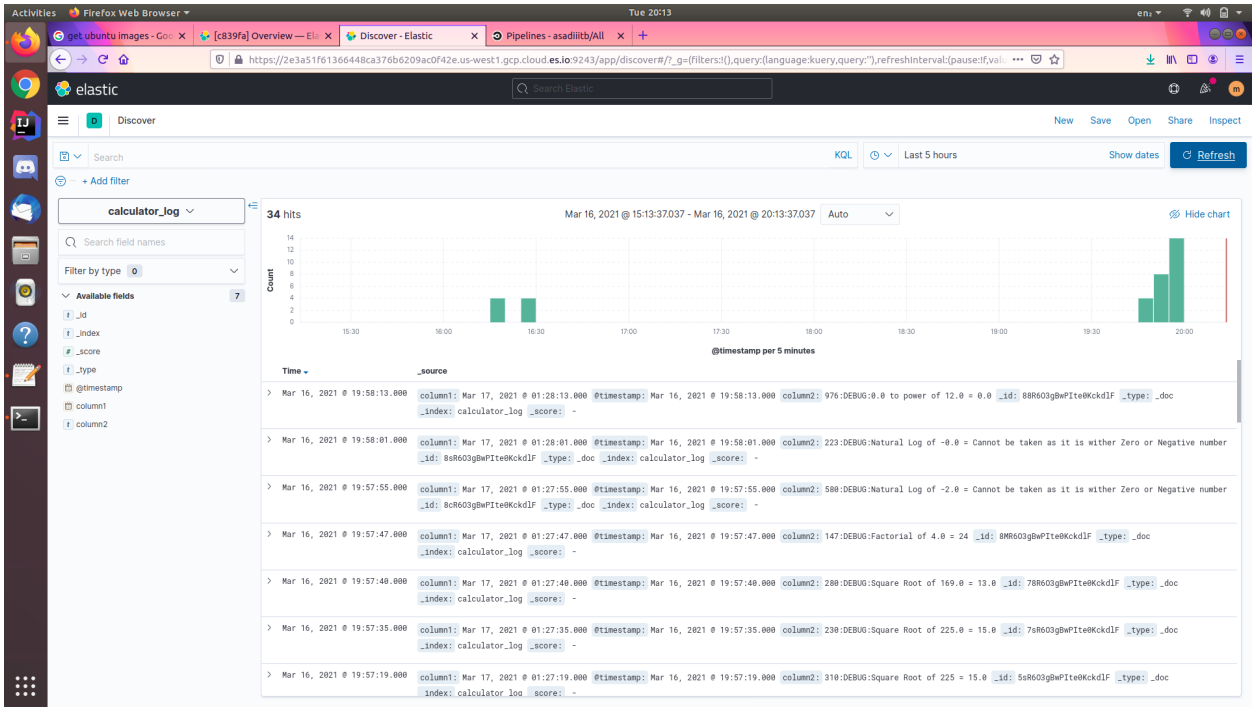


Fig: Visualizing logs of last 5 hours

We can refine this for certain other duration like below.

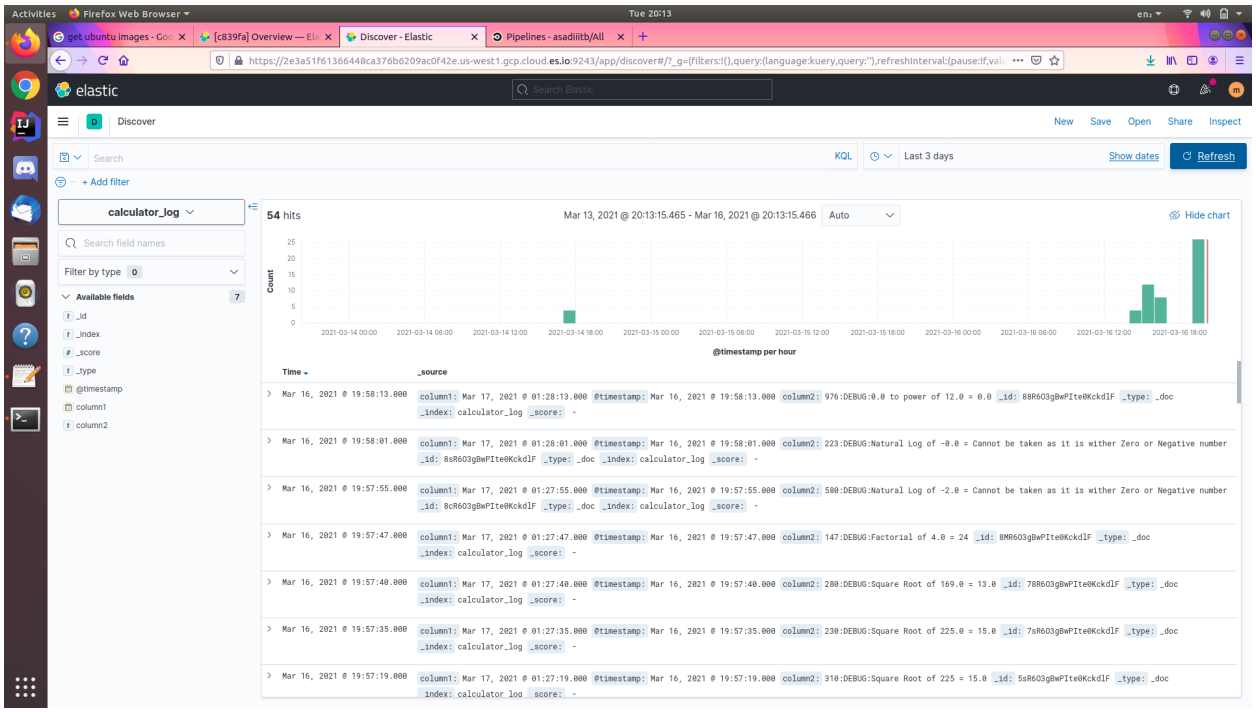


Fig: Visualizing logs of last 3 days

Future Scope:

- We can use a convert this terminal based calculator to a GUI based Calculator, Glimse of which can be seen below. You can view its source code at https://github.com/asadiiitb/ScientificCalculator. Due to Display Environment error we were unable to connect it to CircleCI and Docker Image.

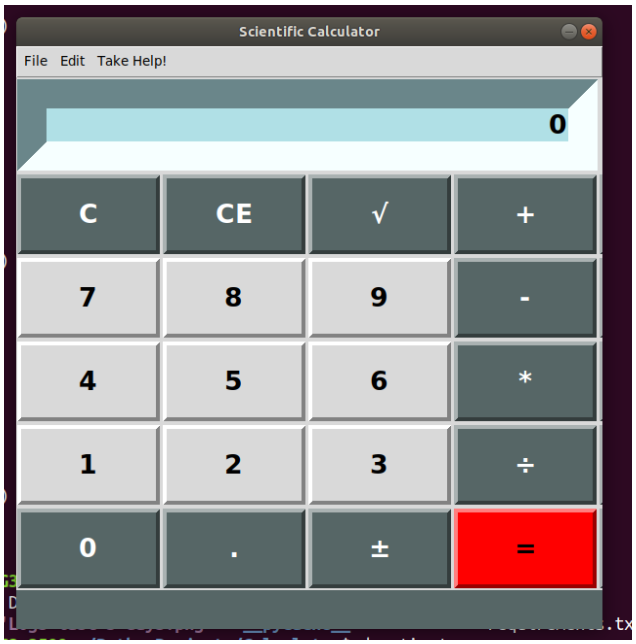


Fig : Scientific Calculator in Standard Mode

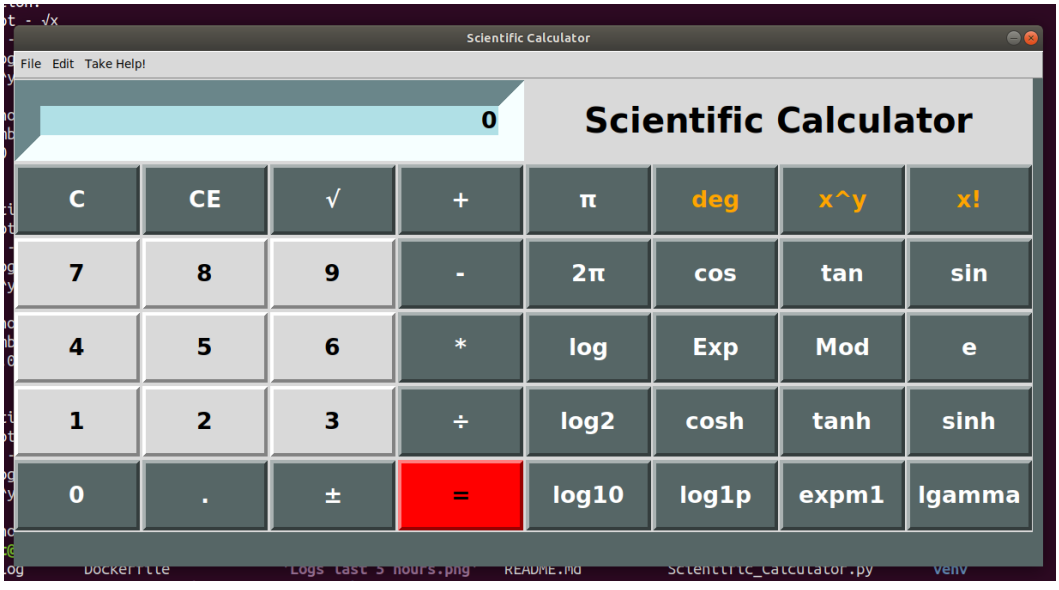


Fig : Calculator in Scientific Mode

References

1. https://circleci.com/blog/using-circleci-workflows-to-replicate-docker-hub-automated-builds/
2. https://circleci.com/blog/how-to-build-a-docker-image-on-circleci-2-0/
3. https://realpython.com/python-logging/
4. https://circleci.com/blog/build-cicd-piplines-using-docker/
5. https://realpython.com/docker-in-action-fitter-happier-more-productive/
6. https://realpython.com/python-continuous-integration/

GitHub : https://github.com/asadiiitb/Dev-Ops-MiniProject
Docker Hub : https://hub.docker.com/repository/docker/imasadansari/devops_calculator