Setting Up Your First GitHub Actions Workflow with Python

Complete Project Structure

Let me show you exactly how the project should be organized:

```
github-actions-demo/
├── .github/
│   └── workflows/
│       └── ci.yml
├── src/
│   ├── __init__.py
│   └── calculator.py
├── tests/
│   ├── __init__.py
│   └── test_calculator.py
├── .gitignore
├── requirements.txt
└── README.md
```

Step A: Setup Commands

```
# Create project directory
mkdir github-actions-demo
cd github-actions-demo

# Create directory structure
mkdir src
mkdir tests
mkdir .github
mkdir .github/workflows

# Create empty __init__.py files (makes directories Python packages)
touch src/__init__.py
touch tests/__init__.py

# Create our main files
touch src/calculator.py
touch tests/test_calculator.py
touch .github/workflows/ci.yml
touch requirements.txt
touch .gitignore
touch README.md
```

Step B: Create the Python Calculator

```python
# src/calculator.py
"""
A simple calculator class for demonstrating CI/CD with GitHub Actions
"""

class Calculator:
    """Basic calculator with four operations"""

    def add(self, a, b):
        """Add two numbers"""
        return a + b

    def subtract(self, a, b):
        """Subtract b from a"""
        return a - b

    def multiply(self, a, b):
        """Multiply two numbers"""
        return a * b

    def divide(self, a, b):
        """Divide a by b"""
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b

    def power(self, a, b):
        """Raise a to the power of b"""
        return a ** b
```

Step C: Create Python Tests

```python
"""

import unittest
import sys
import os

# Add the src directory to the path so we can import our calculator
sys.path.insert(0, os.path.join(os.path.dirname(__file__), '..', 'src'))

from calculator import Calculator


class TestCalculator(unittest.TestCase):
    """Test cases for Calculator class"""

    def setUp(self):
        """Set up test fixtures before each test method"""
        self.calc = Calculator()

    def test_add_positive_numbers(self):
        """Test adding two positive numbers"""
        result = self.calc.add(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        """Test adding negative numbers"""
        result = self.calc.add(-1, -1)
        self.assertEqual(result, -2)

    def test_add_zero(self):
        """Test adding zero"""
        result = self.calc.add(5, 0)
        self.assertEqual(result, 5)

    def test_subtract_numbers(self):
        """Test subtracting two numbers"""
        result = self.calc.subtract(5, 3)
        self.assertEqual(result, 2)
```

```python
    def test_subtract_negative_result(self):
        """Test subtraction with negative result"""
        result = self.calc.subtract(3, 5)
        self.assertEqual(result, -2)

    def test_multiply_numbers(self):
        """Test multiplying two numbers"""
        result = self.calc.multiply(4, 3)
        self.assertEqual(result, 12)

    def test_multiply_by_zero(self):
        """Test multiplication by zero"""
        result = self.calc.multiply(5, 0)
        self.assertEqual(result, 0)

    def test_divide_numbers(self):
        """Test dividing two numbers"""
        result = self.calc.divide(10, 2)
        self.assertEqual(result, 5)

    def test_divide_by_zero_raises_error(self):
        """Test that dividing by zero raises ValueError"""
        with self.assertRaises(ValueError):
            self.calc.divide(10, 0)

    def test_power_operation(self):
        """Test power operation"""
        result = self.calc.power(2, 3)
        self.assertEqual(result, 8)

    def test_power_zero_exponent(self):
        """Test power with zero exponent"""
        result = self.calc.power(5, 0)
        self.assertEqual(result, 1)


if __name__ == '__main__':
    unittest.main()
```

Step D: Create Requirements File

```
# requirements.txt
# No external dependencies needed for this basic demo
# Python's built-in unittest module is sufficient

# For more advanced projects, you might include:
# pytest≥7.0.0
# flake8≥5.0.0
# black≥22.0.0
# coverage≥6.0.0
```

Step E: Create .gitignore

```
# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject
```

Step F: Create GitHub Actions Workflow

```yaml
# .github/workflows/ci.yml
name: CI Pipeline

# When should this workflow run?
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

# What should happen when it runs?
jobs:
  test:
    name: Run Tests
    runs-on: ubuntu-latest

    steps:
    # Step 1: Get the code from the repository
    - name: Checkout code
      uses: actions/checkout@v4

    # Step 2: Set up Python environment
    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.9'
```

```yaml
    # Step 3: Install dependencies (if any)
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi

    # Step 4: Run tests
    - name: Run tests
      run: |
        python -m unittest discover tests -v

    # Step 5: Run a simple syntax check
    - name: Check Python syntax
      run: |
        python -m py_compile src/*.py tests/*.py
```

Step G: Create README

```
# GitHub Actions Demo - Python Calculator

A simple Python calculator project to demonstrate CI/CD with GitHub Actions.

## Project Structure

```
github-actions-demo/
├── .github/
│   └── workflows/
│   └── ci.yml   # GitHub Actions workflow
├── src/
│   ├── __init__.py
│   └── calculator.py   # Main calculator class
├── tests/
│   ├── __init__.py
│   └── test_calculator.py # Test suite
├── .gitignore
├── requirements.txt
└── README.md
```
```

Features

- Basic calculator with add, subtract, multiply, divide, and power operations
- Comprehensive test suite using Python's built-in unittest module
- Automated CI/CD pipeline with GitHub Actions
- Runs tests automatically on every push and pull request

Running Tests Locally

```bash
# Run all tests
python -m unittest discover tests -v

# Run specific test file
python -m unittest tests.test_calculator -v

# Run from the tests directory
cd tests
python test_calculator.py
```

CI/CD Pipeline

The GitHub Actions workflow (`.github/workflows/ci.yml`) automatically:

1. **Triggers** on pushes to `main` or `develop` branches and pull requests
2. **Sets up** a Python 3.9 environment
3. **Installs** dependencies from `requirements.txt`
4. **Runs** all tests with verbose output
5. **Checks** Python syntax for all files

Learning Objectives

This project demonstrates:
- Setting up a basic GitHub Actions workflow
- Running automated tests in CI/CD
- Project structure best practices
- Git workflow with automated testing

During this Demo:

1. **The workflow triggers automatically** - No manual intervention needed
2. **Each step is logged** - Students can see exactly what's happening
3. **Green checkmarks vs red X's** - Visual feedback on success/failure
4. **The workflow runs in a clean environment** - Fresh Ubuntu VM every time

Key Learning Points:

- ✅ The workflow runs automatically on every push
- ✅ Failed tests prevent the workflow from succeeding
- ✅ You get immediate feedback about what's broken
- ✅ The broken code never reaches production