Testing and Continuous Integration

Overview

1. **Testing**: The process of creating and running ancillary code that confirms your software is performing as expected.

2. **Test-driven development**: A *strategy* for incorporating testing into your code development.

3. **Continuous Integration (CI)**: A type of tool for automating parts of the testing process.

What is “testing?”

- Testing consists of writing and running code that’s **independent** of your software, whose purpose is to check that your software functions as intended.
- It is helpful in identifying and fixing issues that may arise when changing older features.
- For other developers, tests are also a source for understanding the basic functionality of each unit of code.

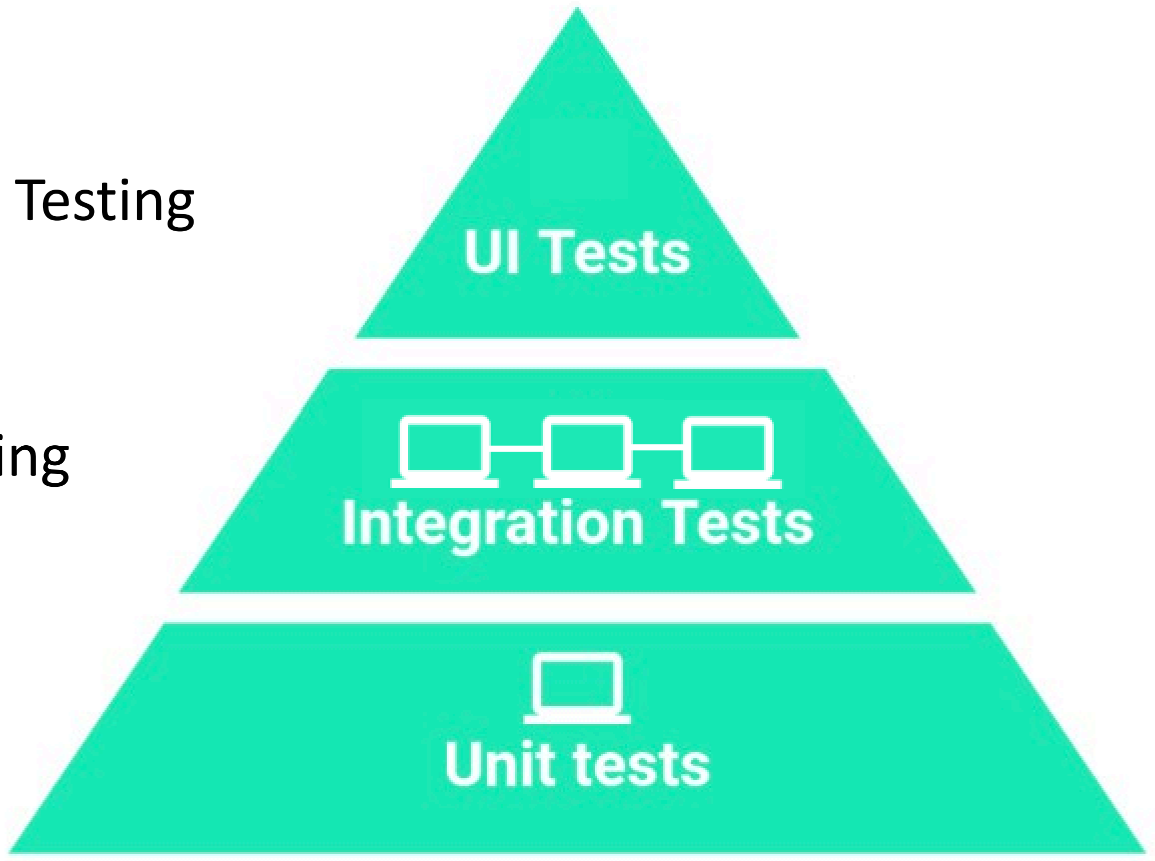
What is “testing?”

There are many levels of testing:

Highest level --- UI or Functional Testing

Intermediate level --- Integration Testing

Lowest level --- Unit Testing



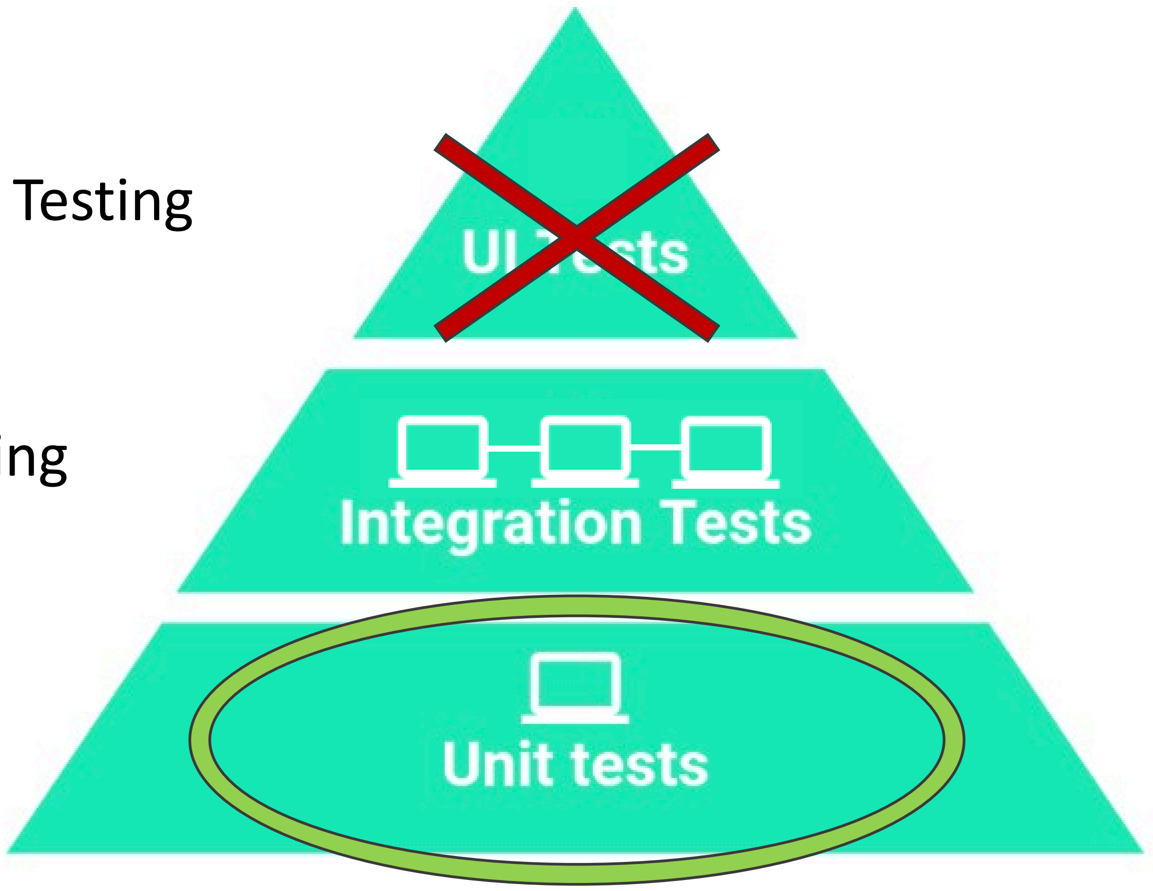
What is “testing?”

There are many levels of testing:

Highest level --- UI or Functional Testing

Intermediate level --- Integration Testing

Lowest level --- Unit Testing

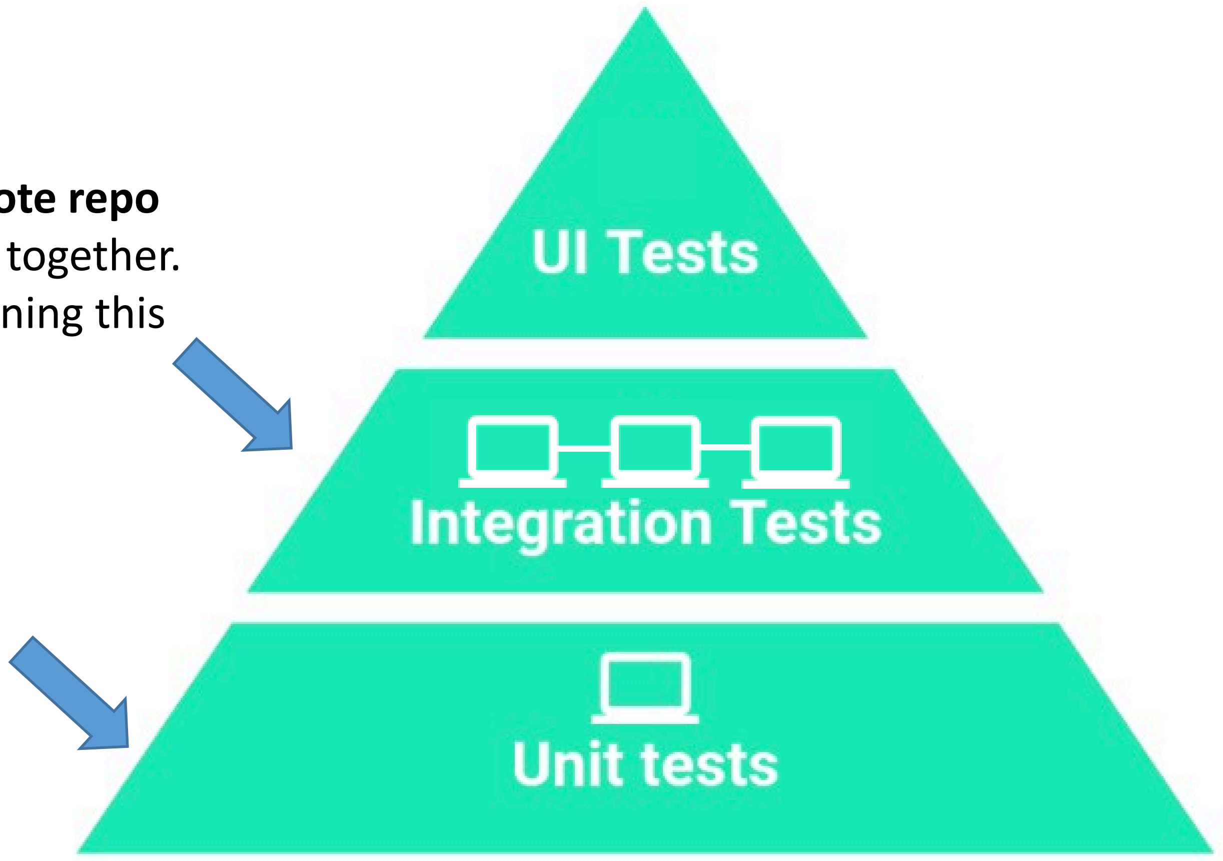


We will mostly focus on **unit testing**, but will also get some exposure to integration testing.

What is “testing?”

Integration testing is done at the level of the **remote repo** (GitHub), where all developers’ code is *integrated* together. “**Continuous Integration**” is a method for streamlining this process.

Unit testing is done at the level of the **local repo**, where an *individual* developer writes their functions (i.e. “units”).



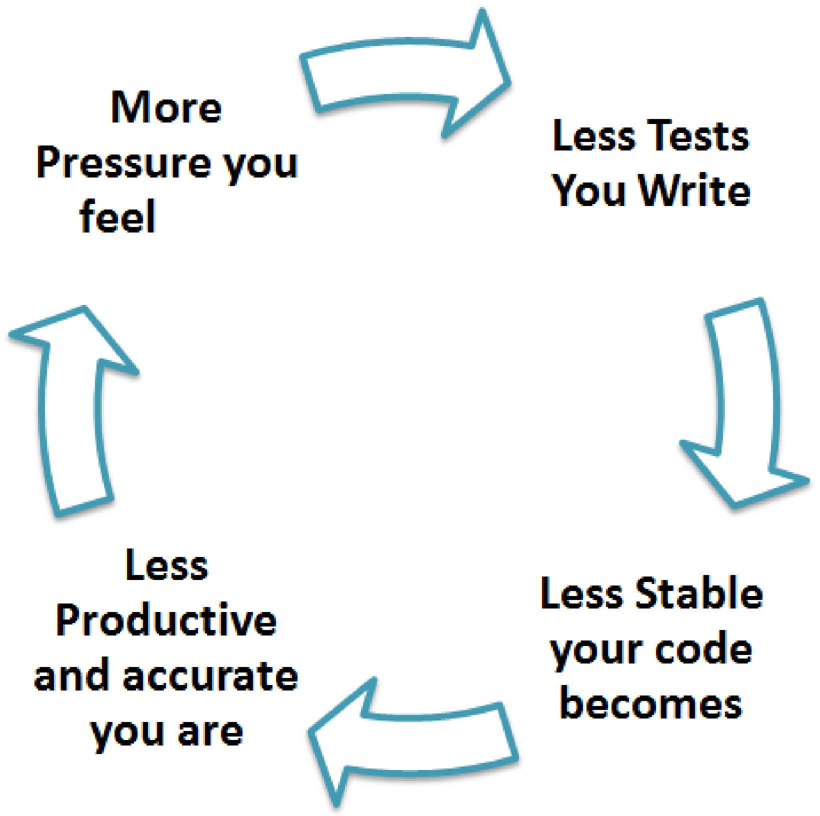
Why bother testing?

"If it hurts, do it more often." –Programing tenet

Without testing you may find yourself caught in a (common) vicious cycle during development.

The more code you write without testing, the more paths you'll have to check for errors, and thus the less inclined you are to test.

Solution: *Adopt a "test as your code" approach.*


More Pressure you feel
Less Tests You Write
Less Stable your code becomes
Less Productive and accurate you are

Unit Testing

- Unit tests are the fastest, most incremental, and most targeted of tests.
- The scope of the "unit" you are testing is **a single function**.
- Each function defined in your software should have one **test function**.
- Each *test function* should test a broad range of the function's possible behaviors (this is referred to as the test's **"coverage"**)

Types of coverage metrics:

1. **Statement coverage**: fraction of the code's executable *statements* run by the test

$$Statement\ cov = \frac{\#\ executed\ lines}{Total\ \#\ of\ lines\ of\ code}$$

2. **Conditional coverage**: fraction of the code's logical operands (AND, OR, greater than, etc.) run by the test

$$Conditional\ cov = \frac{\#\ executed\ operands}{Total\ \#\ of\ operands}$$

3. **Branch coverage**: fraction of the code's branches (paths through the code) run by the test

$$Branch\ cov = \frac{\#\ executed\ branches}{Total\ \#\ of\ braches}$$

Unit Testing

Consider the following 4-line function for a demonstration of unit tests using the three different metrics

```
def multi(a, b):
    if a > 10 and b < 20:
        a = a * b
    return a
```

Statements:

- 4 lines of executable code

Full-coverage test:

`multi(11, 5) == 55`

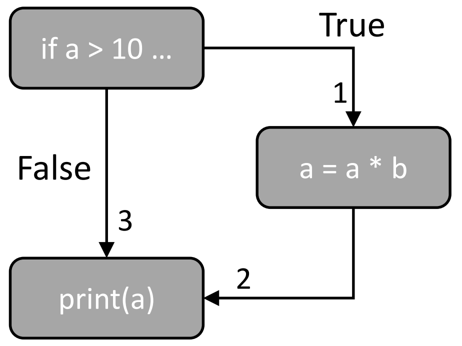
Conditionals:

- `a > 10`
- `b < 20`
- Possible combos: TT, TF, FT, FF

Full-coverage test:

```
TT: multi(11, 5) == 55
TF: multi(11, 25) == 11
FT: multi(-2, 15) == -2
FF: multi(1, 30) == 1
```

Branches:



Full-coverage test:

```
1,2: multi(11, 5) == 55
  3: multi(11, 25) == 11
```

Testing in Python

There are many testing packages for Python. These are some of the most popular.

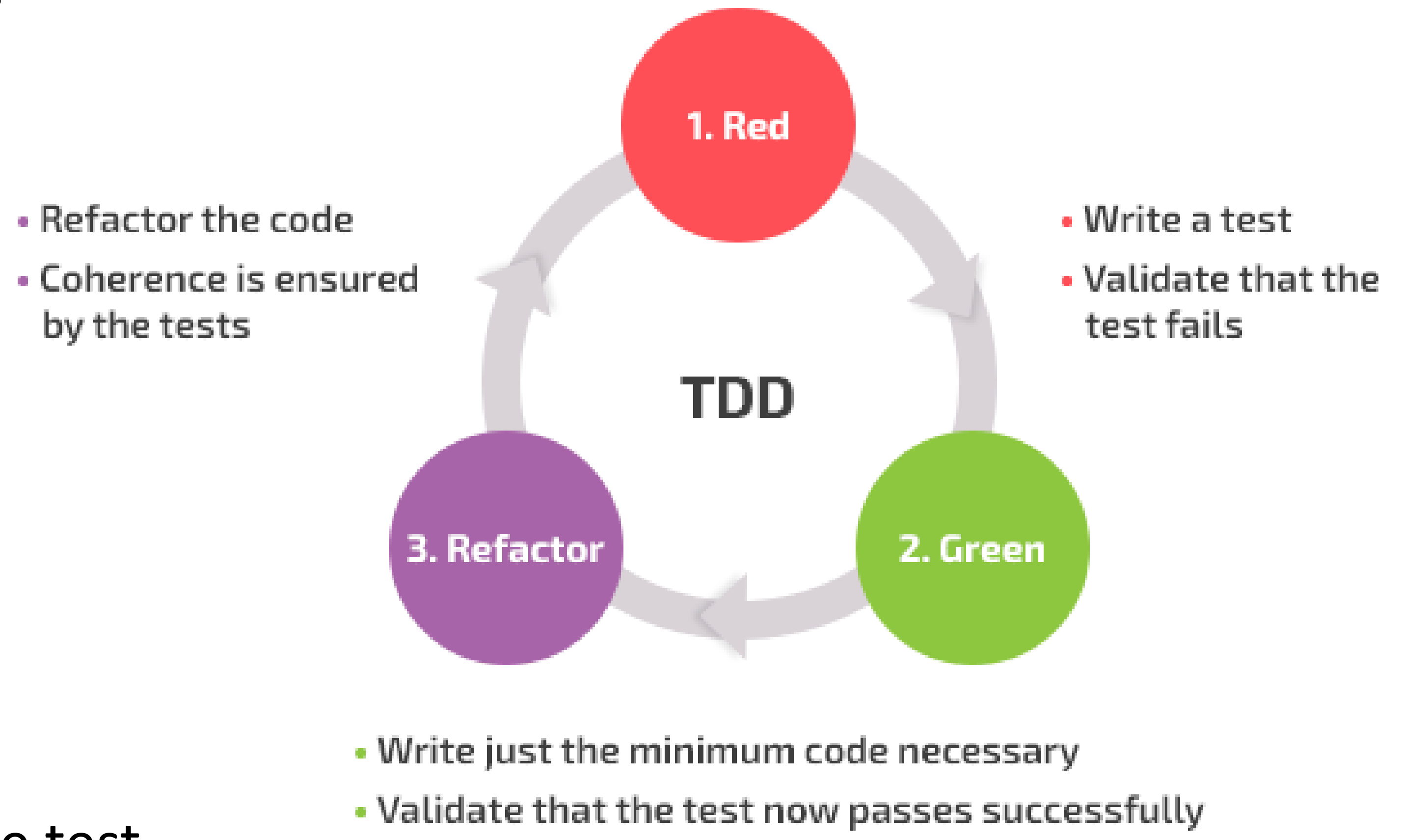
unittest

pytest

Testify

`unittest` is the built-in library for Python and is what we will use for class.

Test-Driven Development

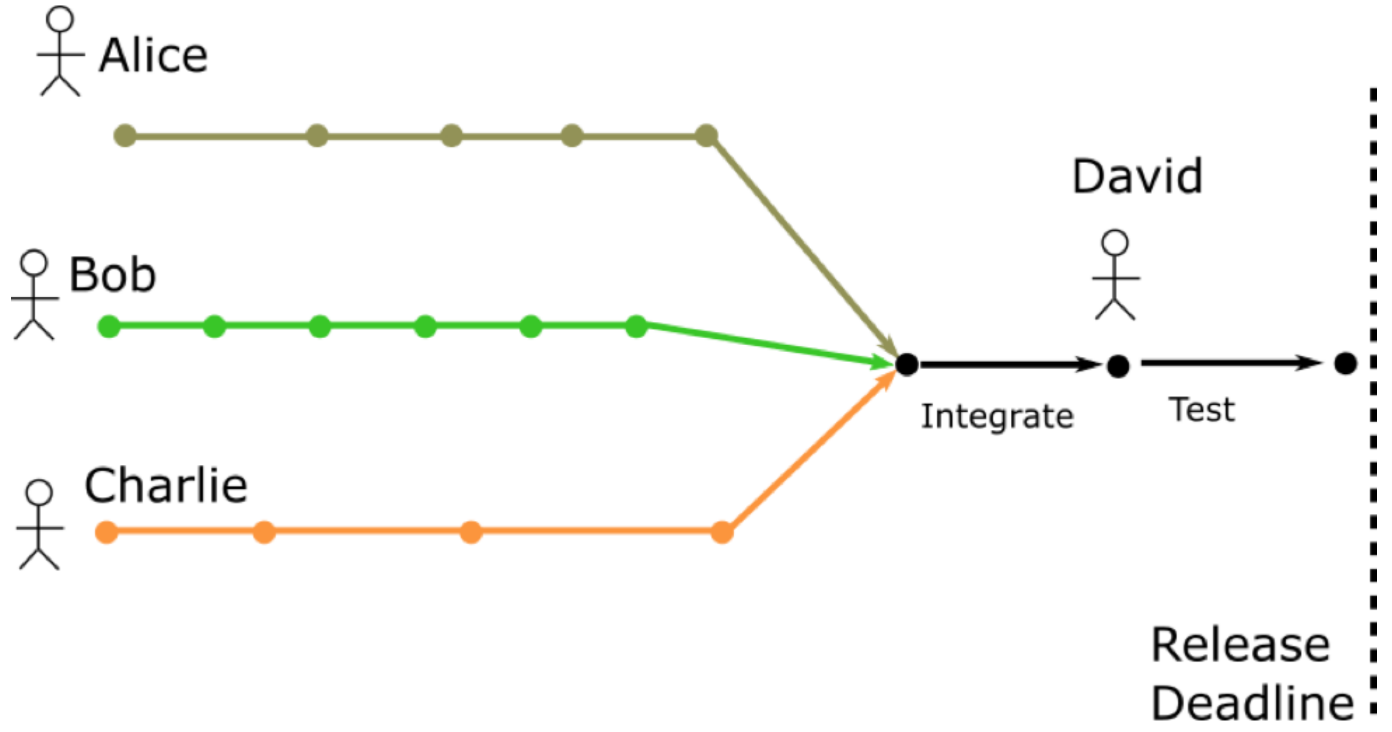

1. Red
Write a test
Validate that the test fails
TDD
2. Green
Write just the minimum code necessary
Validate that the test now passes successfully
3. Refactor
Refactor the code
Coherence is ensured by the tests

A three step process:

1. Write a failing test
2. Write the minimum code to pass the test
3. Refactor the code and repeat from #1

Traditional Development

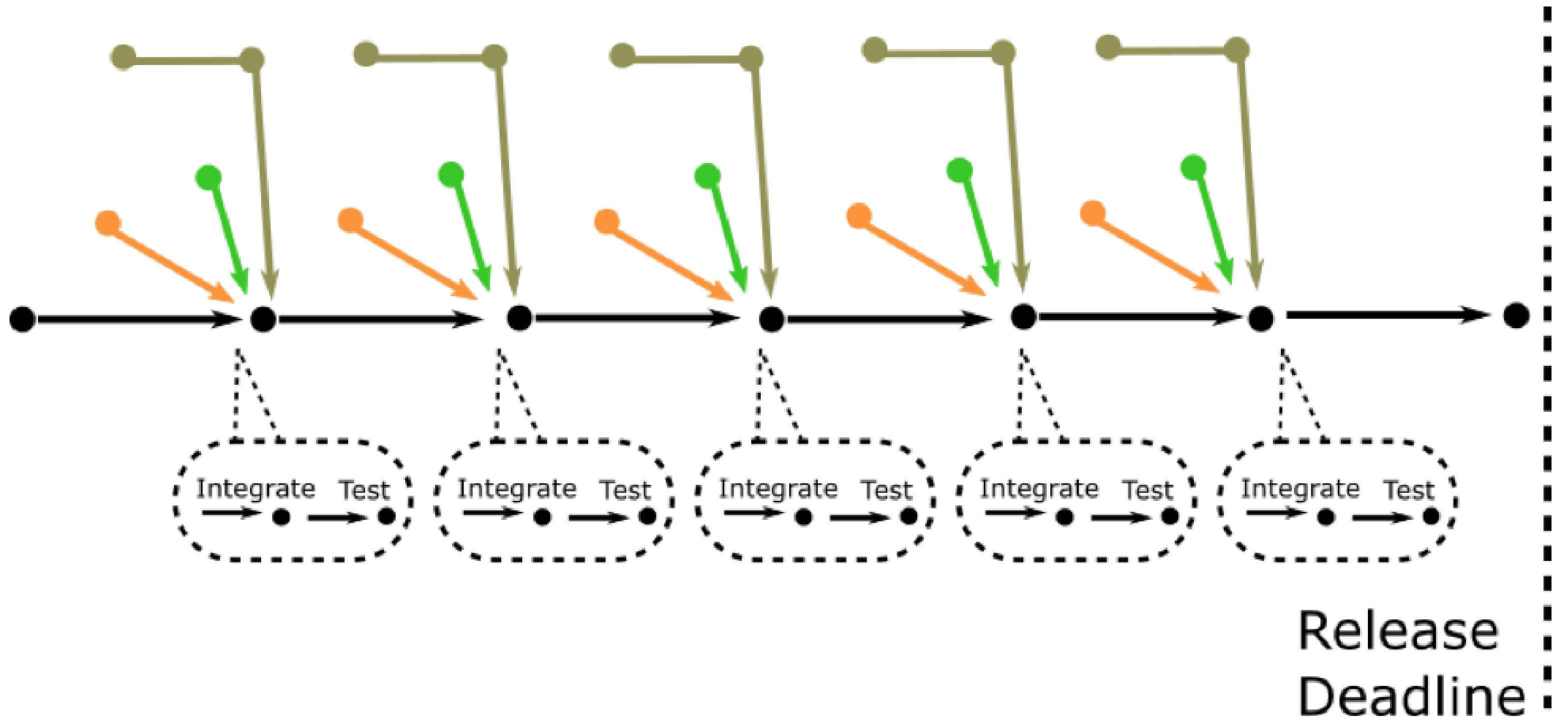
Traditional software development with a team consists of each developer working on their own feature independently and, once those features are mature, manually merging/integrating those features together into a final product, which then needs to be tested.



The problem: The longer the independent development time, the *less* features are likely to coordinate well with one another and the *more* code there will be to integrate. Basically, it results in larger amounts of incompatible code.

Continuous Integration

The solution: Frequent, regular integrations that are smaller in scope. This is called "continuous integration."



Smaller scale integrations are easier to automate and there are a number of services that can be used for this purpose.

Continuous Integration

These are some of the common services for performing CI.

They do a great deal more than just testing, including linting, packaging, compiling, containerizing, etc.

GitHub Actions

circleci

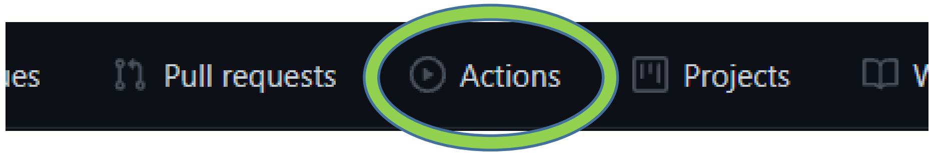
Travis CI

We will be using **GitHub Actions**, since it is well integrated into GH. And we will only be using it for testing.

Continuous Integration with

GitHub Actions

At the top of each GH repo is an "Actions" tab which contains all associated actions as well as action templates:



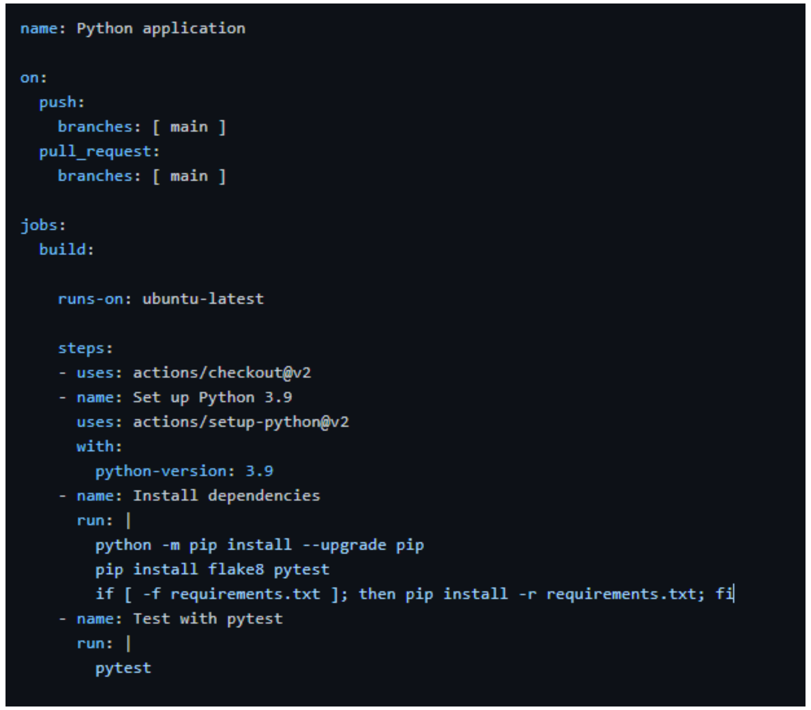
An "Action" is basically a script.

The script at right installs Python 3.9 and some python packages before running a unit test with `pytest`.

This script can be executed automatically, triggered on various GitHub commands (e.g. a push)

The script is stored in your GitHub repo in the following directory: "`.github/workflows/`"

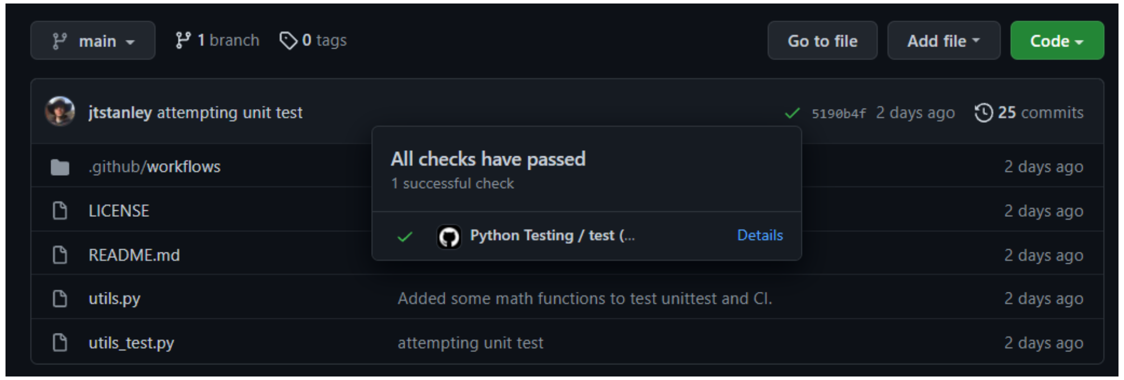
The script uses YAML file syntax. See here for an into to YAML



```yaml
name: Python application

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.9
      uses: actions/setup-python@v2
      with:
        python-version: 3.9
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Test with pytest
      run: |
        pytest
```

Continuous Integration with

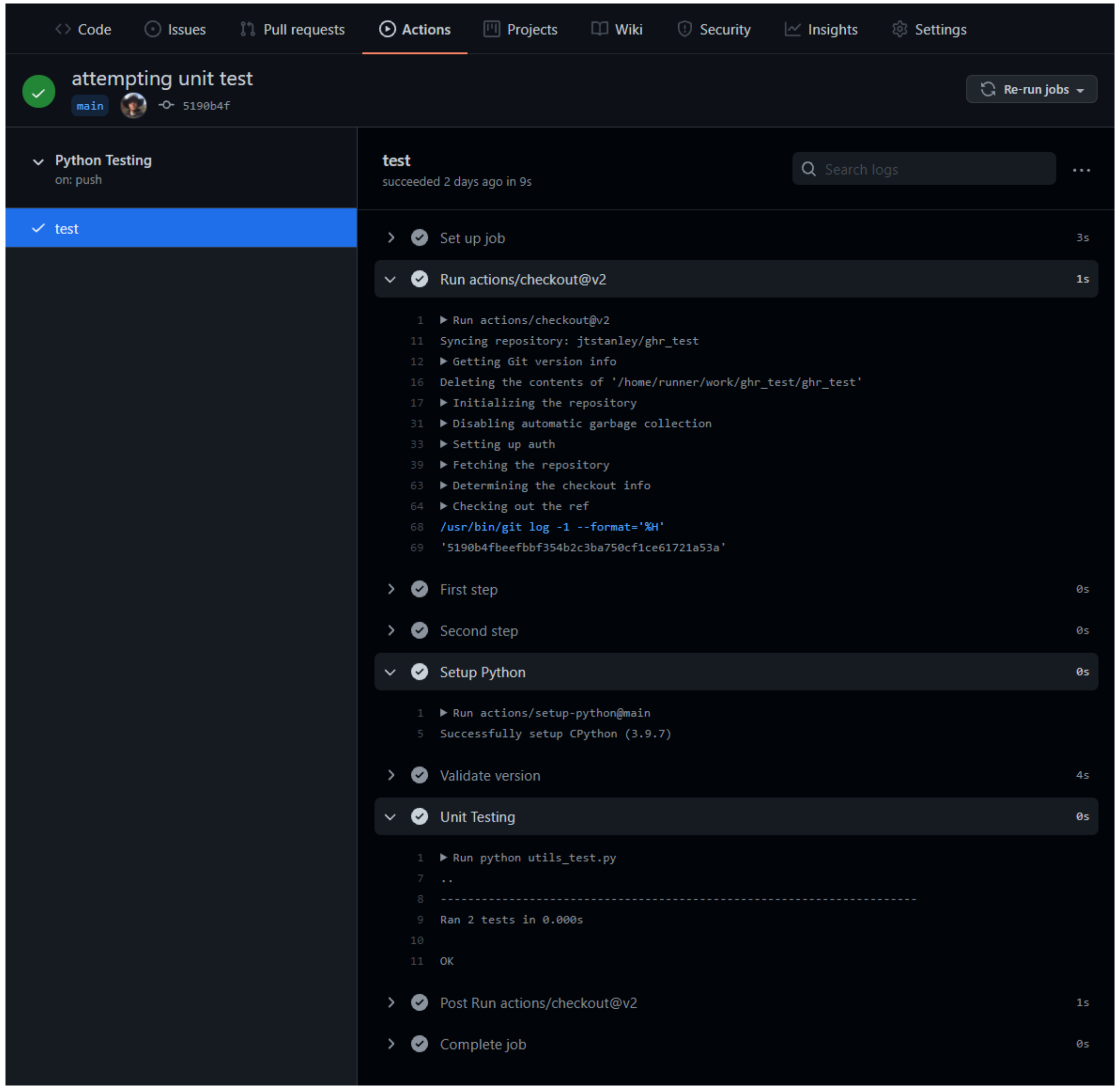
GitHub Actions


main · 1 branch · 0 tags · Go to file · Add file · Code

jtstanley attempting unit test ✓ 5190b4f 2 days ago · 25 commits

All checks have passed
1 successful check
✓ Python Testing / test (... Details

.github/workflows		2 days ago
LICENSE		2 days ago
README.md		2 days ago
utils.py	Added some math functions to test unittest and CI.	2 days ago
utils_test.py	attempting unit test	2 days ago


Code · Issues · Pull requests · Actions · Projects · Wiki · Security · Insights · Settings

attempting unit test
main · 5190b4f
Re-run jobs

Python Testing
on: push

✓ test

test
succeeded 2 days ago in 9s

Search logs

```
Set up job                                                        3s
Run actions/checkout@v2                                           1s
  ▶ Run actions/checkout@v2
  Syncing repository: jtstanley/ghr_test
  ▶ Getting Git version info
  Deleting the contents of '/home/runner/work/ghr_test/ghr_test'
  ▶ Initializing the repository
  ▶ Disabling automatic garbage collection
  ▶ Setting up auth
  ▶ Fetching the repository
  ▶ Determining the checkout info
  ▶ Checking out the ref
  /usr/bin/git log -1 --format='%H'
  '5190b4fbeefbbf354b2c3ba750cf1ce61721a53a'
First step                                                        0s
Second step                                                       0s
Setup Python                                                      0s
  ▶ Run actions/setup-python@main
  Successfully setup CPython (3.9.7)
Validate version                                                  4s
Unit Testing                                                      0s
  ▶ Run python utils_test.py
  ..
  ----------------------------------------------------------------------
  Ran 2 tests in 0.000s

  OK
Post Run actions/checkout@v2                                      1s
Complete job                                                      0s
```

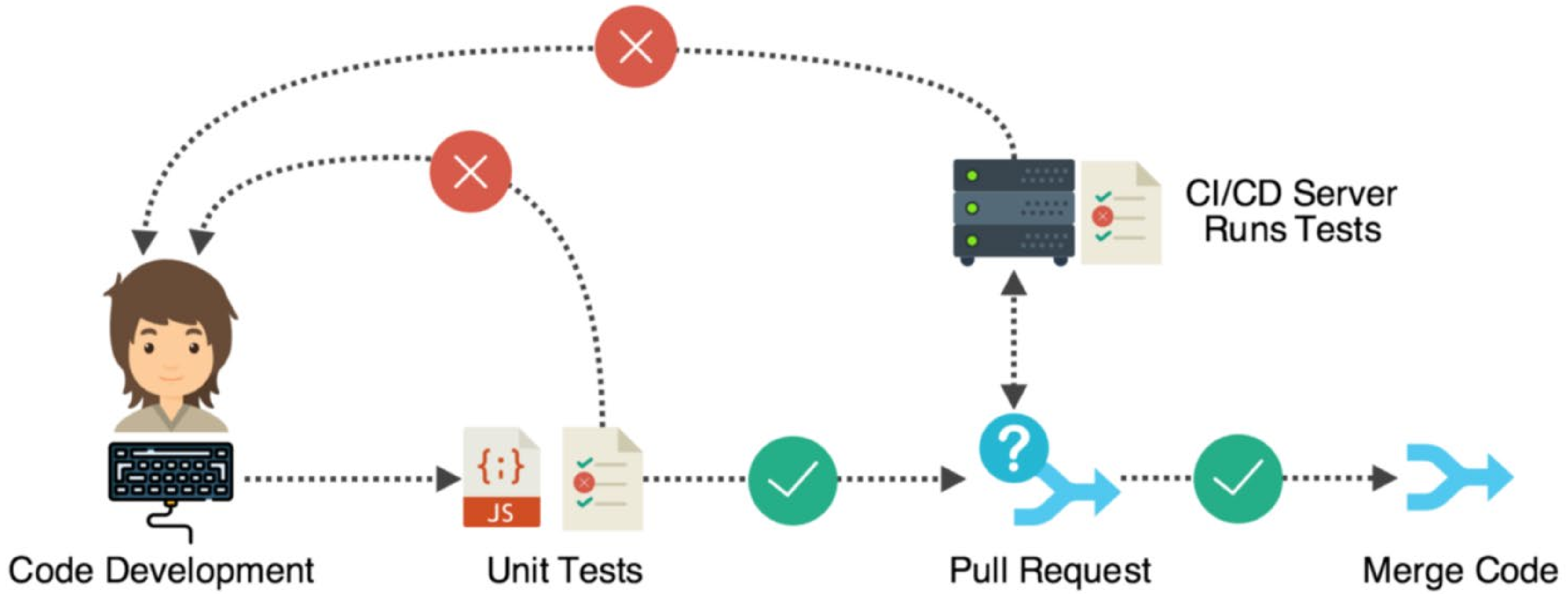
Any YAML file stored in the “workflows” directory will be monitored by GitHub.

If all Actions are executed successfully, your repo will have a green check mark.

Under the Actions tab you can see a detail summary of each Action execution and, if it failed, where it did so.

At right you can see our Python setup was successful and we ran two unit tests which passed.

The big picture: Unit testing + CI on GitHub


CI/CD Server
Runs Tests
{;}
JS
Code Development
Unit Tests
Pull Request
Merge Code