Lesson 5: Testing Software

Reference:
Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson
https://merely-useful.tech/py-rse/testing.html

Assertions

Defensive programming assumes that mistakes will happen, and guards against them.

One way to guard against them is **adding assertions,** which checks that something must be true at a certain point in the program, and halts the program if something unexpected happens.

We can add **user-defined error messages** to indicate the error.

```
frequencies = [13, 10, 2, -4, 5, 6, 25]
total = 0.0
for freq in frequencies[:5]:
    assert freq >= 0.0, 'Word frequencies must be non-negative'
    total += freq
print('total frequency of first 5 words:', total)
```

```
---------------------------------------------------------------
AssertionError                Traceback (most recent call last)
<ipython-input-19-33d87ea29ae4> in <module>()
      2 total = 0.0
      3 for freq in frequencies[:5]:
----> 4     assert freq >= 0.0, 'Word frequencies must be
            non-negative'
      5     total += freq
      6 print('total frequency of first 5 words:', total)

AssertionError: Word frequencies must be non-negative
```

Assertions (generally) fall into three categories:

- A **precondition** is something that must be true at the start of a function in order for it to work correctly.
- A **postcondition** is something that the function guarantees is true when it finishes.
- An **invariant** is something that is true for every iteration in a loop. The invariant might be a property of the data, or it might be something like, “the value of `highest` is less than or equal to the current loop index.”

Unit testing

A **unit test** checks of the correctness of a single unit of software.

- What constitutes a “unit” is subjective, but typically it means the behavior of a single function in one situation.

A unit test will typically have:

- a **fixture**, which is the thing being tested (e.g., an array of numbers);
- an **actual result**, which is what the code produces when given the fixture; and
- an **expected result** that the actual result is compared to.

Testing Frameworks

Pytest is a testing framework that simplifies the creation, organization, and execution of tests.

1. Tests are put in files whose names begin with test_.
2. Each test is a function whose name also begins with test_.
3. These functions use assert to check results.

To add more tests, we simply write more `test_` functions in this py file!

```
from collections import Counter

import countwords


def test_word_count():
    """Test the counting of words.

    The example poem is Risk, by Anaïs Nin.
    """
    risk_poem_counts = {'the': 3, 'risk': 2, 'to': 2, 'and': 1,
      'then': 1, 'day': 1, 'came': 1, 'when': 1, 'remain': 1,
      'tight': 1, 'in': 1, 'a': 1, 'bud': 1, 'was': 1,
      'more': 1, 'painful': 1, 'than': 1, 'it': 1, 'took': 1,
      'blossom': 1}
    expected_result = Counter(risk_poem_counts)
    with open('test_data/risk.txt', 'r') as reader:
        actual_result = countwords.count_words(reader)
    assert actual_result == expected_result
```

Testing Frameworks

The pytest library comes with a command-line tool called `pytest`.

When we run it with no options, it searches for all files in or fellow the working directory whose names match the pattern `test_*.py`

```
$ pytest
```

```
================== test session starts ======================
platform darwin -- Python 3.7.6, pytest-6.2.0, py-1.10.0,
pluggy-0.13.1
rootdir: /Users/amira
collected 1 item

bin/test_zipfs.py .                                    [100%]

===================== 1 passed in 0.02s =====================
```

Testing Floating-Point Values

How do we write tests when we don’t know precisely what the right answer is?

- Check if the actual value is within some **tolerance** of the expected value.
- The tolerance can be expressed as the **absolute error**, which is the absolute value of the difference between two,
- or the **relative error**, which the ratio of the absolute error to the value we’re approximating

```
==================== test session starts ======================
platform darwin -- Python 3.7.6, pytest-6.2.1, py-1.10.0,
pluggy-0.13.1
rootdir: /Users/amira
collected 2 items

bin/test_zipfs.py F.                                     [100%]

========================== FAILURES ===========================
_____________________________ test_alpha ______________________________

  def test_alpha():
      """Test the calculation of the alpha parameter.

      The test word counts satisfy the relationship,
        r = cf**(-1/alpha), where
        r is the rank,
        f the word count, and
        c is a constant of proportionality.

      To generate test word counts for an expected alpha of
        1.0, a maximum word frequency of 600 is used
        (i.e. c = 600 and r ranges from 1 to 600)
      """
      max_freq = 600
      counts = np.floor(max_freq / np.arange(1, max_freq + 1))
      actual_alpha = plotcounts.get_power_law_params(counts)
      expected_alpha = 1.0
>     assert actual_alpha == expected_alpha
E     assert 0.9951524579316625 == 1.0

bin/test_zipfs.py:26: AssertionError
================== short test summary info ==================
FAILED bin/test_zipfs.py::test_alpha - assert 0.99515246 == 1.0
================= 1 failed, 1 passed in 3.98s =================
```

This might actually be close enough to 1 for the purposes of the test.

```
from collections import Counter

import pytest
import numpy as np

import plotcounts
import countwords


def test_alpha():
    """Test the calculation of the alpha parameter.

    The test word counts satisfy the relationship,
      r = cf**(-1/alpha), where
      r is the rank,
      f the word count, and
      c is a constant of proportionality.

    To generate test word counts for an expected alpha of
      1.0, a maximum word frequency of 600 is used
      (i.e. c = 600 and r ranges from 1 to 600)
    """
    max_freq = 600
    counts = np.floor(max_freq / np.arange(1, max_freq + 1))
    actual_alpha = plotcounts.get_power_law_params(counts)
    expected_alpha = pytest.approx(1.0, abs=0.01)
    assert actual_alpha == expected_alpha


def test_word_count():
    #...as before...
```

… so we can set an appropriate tolerance.

```
$ pytest
```

```
==================== test session starts ======================
platform darwin -- Python 3.7.6, pytest-6.2.0, py-1.10.0,
pluggy-0.13.1
rootdir: /Users/amira
collected 2 items

bin/test_zipfs.py ..                                     [100%]

===================== 2 passed in 0.69s =======================
```

Integration Testing

Integration testing is a test that checks whether the parts of a system work **properly when put together.**

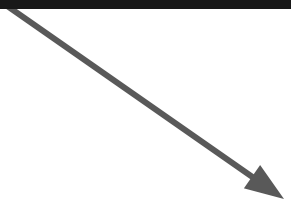
Integration tests are structured the same way as unit tests:

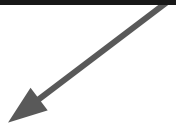
- **a fixture** is used to produce an **actual result** that is compared against the **expected result**.
- However, **creating the fixture** and running the code can be considerably **more complicated**

Example: Testing functions used to read in a txt file and count the 5 words

```python
def read_text(file_path):
    with open(file_path, 'r') as file:
        return file.read()
```

```python
def count_words(text):
    words = text.split()
    return len(words)
```





```python
import pytest

def test_text_processing_integration():
    text = read_text('sample.txt')
    count = count_words(text)
    assert count == 5
```

Regression Testing

When we don’t know the answer, we can use regression tests to **compare today’s answer with a previous one**.

This **doesn’t guarantee that the answer is right**—if the original answer is wrong, we could carry that mistake forward indefinitely—but it does **draw attention to any changes** (or “regressions”).

```python
def calculate_discount(original_price, discount_percentage):
    return original_price - (original_price * discount_percentage / 100)

def test_calculate_discount_regression():
    expected_price = 80
    actual_price = calculate_discount(100, 20)
    assert actual_price == expected_price
```

Test Coverage

How much of our code do the tests we've written check? We can use the `coverage` library to get the **percentage of lines of code that have been tested**.

```
$ coverage run -m pytest
```

```
===================== test session starts =====================
platform darwin -- Python 3.7.6, pytest-6.2.0, py-1.10.0,
pluggy-0.13.1
rootdir: /Users/amira
collected 4 items

bin/test_zipfs.py ....                                  [100%]

====================== 4 passed in 0.72s ======================
```

Test Coverage

```
$ coverage report -m
```

```
bin/countwords.py      20      7    65%   25-26, 30-38
bin/plotcounts.py      58     37    36%   48-55, 75-77, 82-83,
                                          88-118, 122-140
bin/test_zipfs.py      31      0   100%
bin/utilities.py        8      5    38%   18-22
------------------------------------------------------
TOTAL                 117     49    58%
```

Test Coverage

If we run `coverage html` at the command line and open `htmlcov/index.html`, we can click on the name of our scripts and get colorized line-by-line display.

Continuous Integration

- runs tests automatically whenever a change is made
  - can also be set up to alert programmers how their changes may affect others systems (i.e., warning a Windows user that a change would break things for a Mac user)
- Travis CI:
  - popular tool that integrates well with Github
  - every time a change is committed to a repo, Travis CI:
    - creates a fresh environment
    - makes a fresh clone of the repo
    - runs whatever commands the project's managers have set up

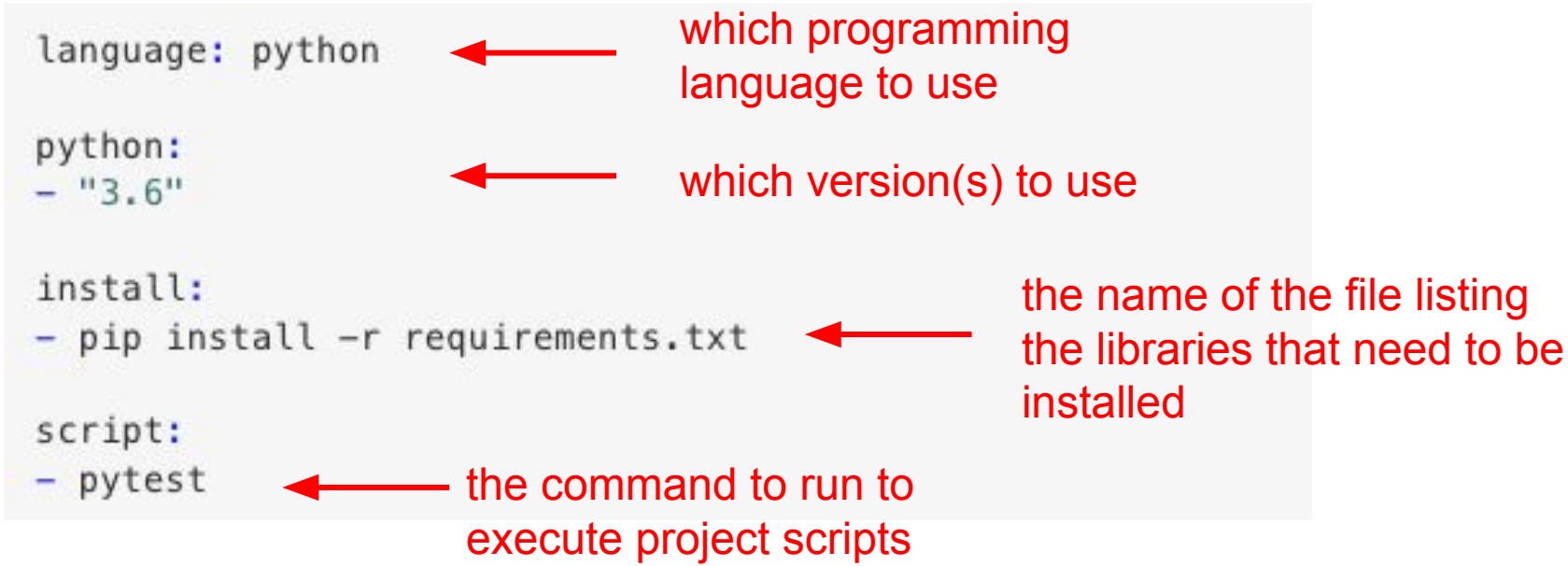
Continuous Integration: Travis CI

To prep for Travis CI, first add a file called `.travis.yml` to the root directory of the repo

```
language: python

python:
- "3.6"

install:
- pip install -r requirements.txt

script:
- pytest
```

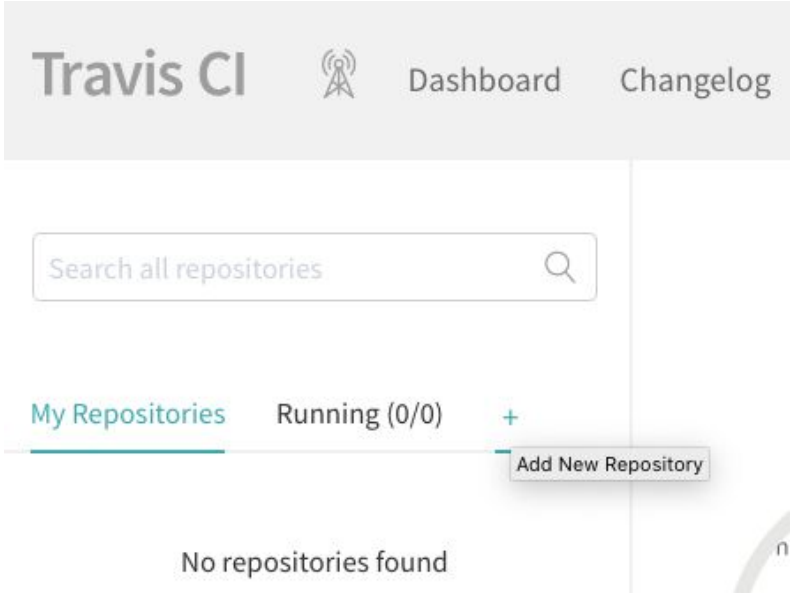
Continuous Integration: Travis CI

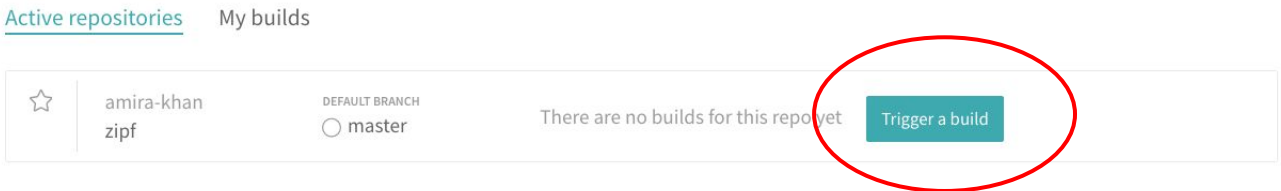
To prep for Travis CI, first add a file called `.travis.yml` to the root directory of the repo



Continuous Integration: Travis CI

1. Create an account on https://www.travis-ci.com/ (if we don't already have one).
2. Link our Travis CI account to our GitHub account (if we haven't done so already).
3. Tell Travis CI to watch the repository that contains our project.





When to test?
Option 1: Test Driven Development

Rather than writing code and then writing tests, we write tests first and then write just enough code to make them pass

Advocates claim that it leads to better code because:

1. Writing tests **clarifies** what the code is actually supposed to do.
2. It eliminates **confirmation bias**.
    a. If someone has just written a function, they are predisposed to want it to be right, so they will bias their tests towards proving that it is correct instead of trying to uncover errors.
3. Writing tests first ensures that they **actually get written**.

When to test?
Option 2: Checking-Driven Development (recommended)

- Writing just a few lines of code and testing it before moving on rather than writing several pages of code and then spending hours on testing
- For example: every time we add a step to our pipeline and look at its output, we also add a check of some kind to the pipeline to ensure that what we are checking remains true if it were run on other data or if the pipeline evolves