Testing Python

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

14 Oct 2019

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

What is Testing?

NO:

Run a program on some nice examples.

What is Testing?

NO:

Run a program on some nice examples.

YES:

Run a program with the intent of finding an error.

- identify corner cases
- devise tricky examples
- exercise the program logic

What is Testing?

NO:

Run a program on some nice examples.

YES:

Run a program with the intent of finding an error.

- identify corner cases
- devise tricky examples
- exercise the program logic

Caveat (Edsger W. Dijkstra, 1970, EWD249)

Program testing can be used to show the presence of bugs, but never to show their absence!

Why Test?

- Increase Confidence
- Early and quick feedback on changes
  Up to 20% of bugfixes introduce new bugs. Beware!
- Debugging aid
- TDD (test driven design)
  Specification by way of test cases
  Implementation proceeds along the test cases

The Downside

- Tests are also code and can be buggy
- Tests take time and effort to write and maintain
- Tests can be brittle can give different results on different runs
- Tests can give a false sense of security (remember Dijkstra!)

Different Kinds of Tests
Select examples

Unit Test

- Tests a unit of code in isolation.
- A unit can be a single function or method, an entire class, or an entire module.
- Lightweight and fast.

Different Kinds of Tests
Select examples

Unit Test

- Tests a unit of code in isolation.
- A unit can be a single function or method, an entire class, or an entire module.
- Lightweight and fast.

Integration Test

- Tests the interplay of several units.
- Stress on checking compatibility of interfaces.

Different Kinds of Tests
Select examples

Unit Test

- Tests a unit of code in isolation.
- A unit can be a single function or method, an entire class, or an entire module.
- Lightweight and fast.

Integration Test

- Tests the interplay of several units.
- Stress on checking compatibility of interfaces.

System Test

- Test of applications on the system level.
- Heavyweight.

Automatic Tests

Why automatize?

- Tests are code
- Tests can be parameterized and run with several instances
- Tests can run in the background (in the cloud, over night, . . .)
- Regression tests:
  - Run tests after each change
  - Newly introduced bugs can be caught early

Unit Testing

- Well-understood methodology
- Supports TDD
- Tool support
- Easily automatized

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Let's Test

Task

1. The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.
2. Write meaningful tests for this function.

Let's Test

Task

1. The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.
2. Write meaningful tests for this function.

How to Approach Testing

```
def list_filter (x, xs):
    ######
```

- Suppose your worst enemy implemented this function.
- How would you test it?

Meaningful Tests

The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.

Meaningful Tests

The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.

1. empty list (boundary case):
   `list_filter (4, []) == []`

Meaningful Tests

The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.

1. empty list (boundary case):
   ```
   list_filter (4, []) == []
   ```
2. sharpness of the test (mixup of the relation):
   ```
   list_filter (4, [4]) == [4]
   list_filter (4, [3]) == [3]
   list_filter (4, [5]) == []
   ```

Meaningful Tests

The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.

1. empty list (boundary case):
   ```
   list_filter (4, []) == []
   ```
2. sharpness of the test (mixup of the relation):
   ```
   list_filter (4, [4]) == [4]
   list_filter (4, [3]) == [3]
   list_filter (4, [5]) == []
   ```
3. uniformity (problem with the iteration):
   ```
   list_filter (4, [1,3,5]) == [1,3]
   list_filter (4, [1,5,4]) == [1,4]
   ```

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Writing Tests with pytest

- `pytest` (http://pytest.org/en/latest/) is a Python tool for testing
- We start with the simplest way of using it.

Writing Tests with pytest

A pytest Test

- Each function whose name starts with `test_` is a test function.
- Each test function should contain an `assert` statement that corresponds to a valid property of the subject.
- Test functions can be included at the end of the source file.
- Running the source file with `pytest` executes all test functions.

Example Tests

```
def test_empty():
    assert list_filter (4, []) == []

def test_sharp1():
    assert list_filter (4, [4]) == [4]

def test_sharp2():
    assert list_filter (4, [3]) == [3]

def test_sharp3():
    assert list_filter (4, [5]) == []

def test_uniform1():
    assert list_filter (4, [1,3,5]) == [1,3]

def test_uniform2():
    assert list_filter (4, [1,5,4]) == [1,4]
```

Running the Tests

On a buggy implementation in file `list_filter.py`:

```python
def list_filter (x, xs):
    return [ y for y in xs if y < x ]
```

To run the tests:

```
src$ pytest list_filter.py
```

Output of pytest list_filter.py

```
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
rootdir: /Users/thiemann/svn/proglang-talks/20191014-python-testing-for-physics/src/list_filter, inifile:
collected 6 items

list_filter.py .F...F                                                    [100%]

=================================== FAILURES ===================================
_________________________________ test_sharp1 __________________________________

    def test_sharp1():
>       assert list_filter (4, [4]) == [4]
E       assert [] == [4]
E         Right contains more items, first extra item: 4
E         Use -v to get the full diff

list_filter.py:12: AssertionError
________________________________ test_uniform2 _________________________________

    def test_uniform2():
>       assert list_filter (4, [1,5,4]) == [1,4]
E       assert [1] == [1, 4]
E         Right contains more items, first extra item: 4
E         Use -v to get the full diff

list_filter.py:24: AssertionError
====================== 2 failed, 4 passed in 0.08 seconds ======================
```

More Verbose Output of `pytest -v list_filter.py`

```
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.0.1, py-1.7.0, pluggy-0.8.0 -- /usr/local/opt/python/bin/python3.7
cachedir: .pytest_cache
rootdir: /Users/thiemann/svn/proglang-talks/20191014-python-testing-for-physics/src/list_filter, inifile:
collecting ... collected 6 items

list_filter.py::test_empty PASSED                                        [ 16%]
list_filter.py::test_sharp1 FAILED                                       [ 33%]
list_filter.py::test_sharp2 PASSED                                       [ 50%]
list_filter.py::test_sharp3 PASSED                                       [ 66%]
list_filter.py::test_uniform1 PASSED                                     [ 83%]
list_filter.py::test_uniform2 FAILED                                     [100%]

=================================== FAILURES ===================================
_________________________________ test_sharp1 __________________________________

    def test_sharp1():
>       assert list_filter (4, [4]) == [4]
E       assert [] == [4]
E         Right contains more items, first extra item: 4
E         Full diff:
E         - []
E         + [4]
E         ?  +

list_filter.py:12: AssertionError
________________________________ test_uniform2 _________________________________

    def test_uniform2():
>       assert list_filter (4, [1,5,4]) == [1,4]
E       assert [1] == [1, 4]
E         Right contains more items, first extra item: 4
E         Full diff:
E         - [1]
E         + [1, 4]
```

Usability

Advice

- Each test function should contain **one** `assert` to test **one** property!

⇒ Testing stops at the first failing `assert` in a function, the remaining asserts are ignored!

Which Errors are Detected?

```
def list_filter (x, xs):
    return [ y for y in xs if y < x ]
```

```
def list_filter (x, xs):
    return [ x for y in xs if y <= x ]
```

```
def list_filter (x, xs):
    r = []
    for y in xs:
        if y <= x: r = [y] + r
    return r
```

```
def list_filter (x, xs):
    r = []
    for i in range(1, len(xs)):
        if xs[i] <= x: r = r + [xs[i]]
    return r
```

Aside on the Specification

The function `list_filter` has two parameters, an integer `x` and a list of integers `xs`, and returns the list of all elements of `xs` which are less than or equal to `x`.

What does it actually fix?

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Testing with Numbers

Task

1. The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.
2. We represent a 2D point by a tuple.
3. Write meaningful tests for this function.

Testing Rotation

The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.

Testing Rotation

The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.

1. rotating the origin by any angle should not matter:
   `vector_rotate ((0,0), 42) == (0, 0)`

Testing Rotation

The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.

1. rotating the origin by any angle should not matter:
   `vector_rotate ((0,0), 42) == (0, 0)`
2. rotating any vector by 0 degrees should leave the vector unchanged:
   `vector_rotate ((10,10), 0) == (10,10)`

Testing Rotation

The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.

1. rotating the origin by any angle should not matter:
   `vector_rotate ((0,0), 42) == (0, 0)`
2. rotating any vector by 0 degrees should leave the vector unchanged:
   `vector_rotate ((10,10), 0) == (10,10)`
3. rotating the unit vector (1,0) by 90 (180, 270) degrees should yield the unit vector (0,1) (resp (-1,0), (0,-1)):
   `assert vector_rotate ((1,0), 90) == (0,1)`

Testing Rotation

The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.

1. rotating the origin by any angle should not matter:
   `vector_rotate ((0,0), 42) == (0, 0)`
2. rotating any vector by 0 degrees should leave the vector unchanged:
   `vector_rotate ((10,10), 0) == (10,10)`
3. rotating the unit vector (1,0) by 90 (180, 270) degrees should yield the unit vector (0,1) (resp (-1,0), (0,-1)):
   `assert vector_rotate ((1,0), 90) == (0,1)`
4. rotating any vector by any angle should leave the length of the vector unchanged

Testing Rotation

The function `vector_rotate` has two parameters, a 2D point `p` and an angle `a` in degrees, and returns a 2D point rotated by `a` degrees around the origin.

1. rotating the origin by any angle should not matter:
   `vector_rotate ((0,0), 42) == (0, 0)`
2. rotating any vector by 0 degrees should leave the vector unchanged:
   `vector_rotate ((10,10), 0) == (10,10)`
3. rotating the unit vector (1,0) by 90 (180, 270) degrees should yield the unit vector (0,1) (resp (-1,0), (0,-1)):
   `assert vector_rotate ((1,0), 90) == (0,1)`
4. rotating any vector by any angle should leave the length of the vector unchanged
5. if `vector_rotate (v, a) == w`, then
   `cos (a) == (v * w) / (v * v)` where `*` stands for the dot product

Applying the Tests to a Correct Implementation . . .

```
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.0.1, py-1.7.0, pluggy-0.8.0 -- /usr/local/opt/python/bin/python3.7
cachedir: .pytest_cache
rootdir: /Users/thiemann/svn/proglang-talks/20191014-python-testing-for-physics/src/tuple_rotate, inifile:
collecting ... collected 5 items

tuple_rotate.py::test_rot_origin PASSED                                  [ 20%]
tuple_rotate.py::test_rot0 PASSED                                        [ 40%]
tuple_rotate.py::test_rot90 FAILED                                       [ 60%]
tuple_rotate.py::test_length PASSED                                      [ 80%]
tuple_rotate.py::test_angle PASSED                                       [100%]

=================================== FAILURES ===================================
__________________________________ test_rot90 __________________________________

    def test_rot90():
>       assert vector_rotate ((1,0), 90) == (0,1)
E       assert (6.123233995736766e-17, 1.0) == (0, 1)
E         At index 0 diff: 6.123233995736766e-17 != 0
E         Full diff:
E         - (6.123233995736766e-17, 1.0)
E         + (0, 1)

tuple_rotate.py:26: AssertionError
====================== 1 failed, 4 passed in 0.07 seconds ======================
```

Floating Point Strikes again

Golden Rule

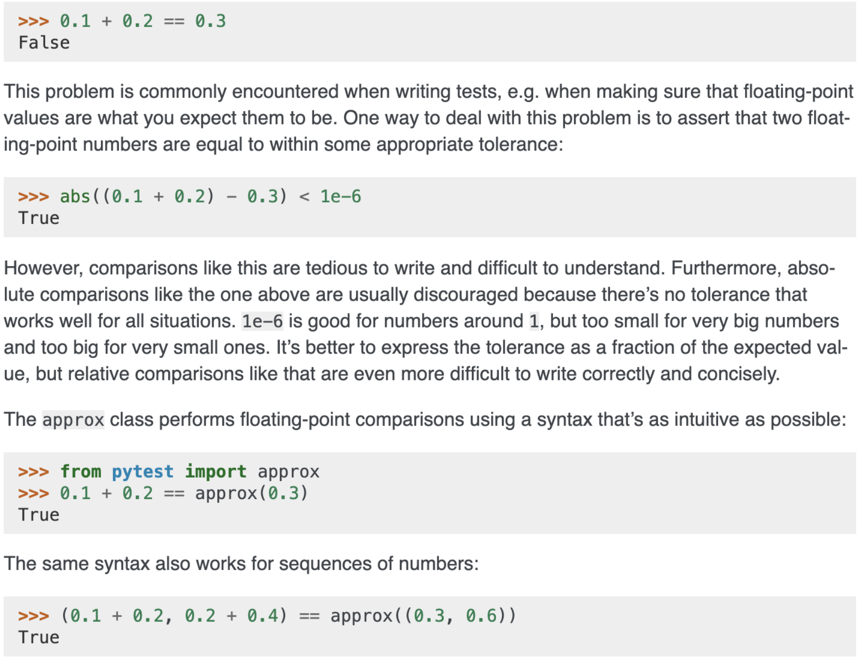
- **Never, never, never** compare floating point numbers for equality!
- See https://docs.python.org/3/tutorial/floatingpoint.html for the reason

Comparing Floating Point in pytest

- Use `pytest.approx`
- This function applies to numbers, sequences, dictionaries, numpy, etc
- It modifies the comparision to make it approximate

From the pytest documentation

http://pytest.org/en/latest/reference.html#pytest-approx



```
>>> 0.1 + 0.2 == 0.3
False
```

This problem is commonly encountered when writing tests, e.g. when making sure that floating-point values are what you expect them to be. One way to deal with this problem is to assert that two floating-point numbers are equal to within some appropriate tolerance:

```
>>> abs((0.1 + 0.2) - 0.3) < 1e-6
True
```

However, comparisons like this are tedious to write and difficult to understand. Furthermore, absolute comparisons like the one above are usually discouraged because there's no tolerance that works well for all situations. `1e-6` is good for numbers around `1`, but too small for very big numbers and too big for very small ones. It's better to express the tolerance as a fraction of the expected value, but relative comparisons like that are even more difficult to write correctly and concisely.

The `approx` class performs floating-point comparisons using a syntax that's as intuitive as possible:

```
>>> from pytest import approx
>>> 0.1 + 0.2 == approx(0.3)
True
```

The same syntax also works for sequences of numbers:

```
>>> (0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 0.6))
True
```

Solution

- ...
- rotating the unit vector (1,0) by 90 (180, 270) degrees should yield the unit vector (0,1) (resp (-1,0), (0,-1)):
  `assert vector_rotate ((1,0), 90) == approx((0,1))`
- [actually, this modification should be applied to **all** tests for this function]

Remark

- Most of the useful tests for `vector_rotate` are **property tests**
- Their formulation includes wording like "any vector" or "any angle" or "for all positive numbers".
- They are most effective if tested for many inputs rather than just one.

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Exercise

Task

Write meaningful tests for the following functions:

1. The function `leap_year` has one parameter, an integer `y` representing a year in the Gregorian calendar, and returns whether `y` is a leap year or not.
   The Gregorian calendar is defined for years `y` greater than 1582 and considers `y` a leap year iff
   - `y` is divisible by 4; and
   - if `y` is divisible by 100, then `y` must be divisible by 400.
2. The function `intersect` has four 2D-points as parameters, representing two lines in two-point-form, and returns the intersection point of those lines, if it exists uniquely, and `None` otherwise.

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

The first subsection is based on the book
Python Continuous Integration and Delivery: A Concise Guide with Examples. Moritz Lenz. Apress 2019.

Continuous Testing

- Testing works best if it is automated
- Good practice: run tests locally before checking in
- But testing a system
  - can be influenced by the local configuration
  - can be time consuming (size, different versions)
  - can be influenced by other developers' changes

⇒ **Continuous Testing**

- Part of the Continuous Integration / Continuous Delivery (CI/CD) tale

⇒ Tests run regularly and/or at each commit to the source repository

- . . . in a controlled environment, on a dedicated machine

The Dedicated Test Machine

On-Premise

- Roll your own CI-server on a machine controlled by your institution
- Preferred for closed source projects

The Dedicated Test Machine

On-Premise

- Roll your own CI-server on a machine controlled by your institution
- Preferred for closed source projects

Software as a Service (SaaS)

- CI-server maintained by the vendor

The Dedicated Test Machine

On-Premise

- Roll your own CI-server on a machine controlled by your institution
- Preferred for closed source projects

Software as a Service (SaaS)

- CI-server maintained by the vendor

In both cases. . .

- need to run potentially faulty software in a controlled way
- several simultaneous runs must be supported

⇒ some isolation mechanism should be used

- industry standard: container-based approach (e.g., **docker**)

Intermezzo: What is Docker?

- Docker is a **container technology**
- A container provides **virtualization** at the operating system level
- Virtualization means that multiple applications can run in isolation on the same machine
- A containerized application is provided as an **image** that contains everything it needs to run starting from the operating system and all customizations
- Containers can
  - share resources with the host and with one another
  - network among themselves and with the outside world
- (Docker is supported by a company called Docker Inc, but there is an open-source version of the software)

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Setting up an On-Premise CI-Server

- Jenkins (https://jenkins.io/) is a popular open-source CI-Server
- Jenkins is a Java application, which can be difficult to install
- But there is a prebuilt docker image for Jenkins that we can customize for our needs of testing Python programs
- This image is available from a central registry, the **docker cloud**, and can be summoned by its name `jenkins/jenkins:lts`
- Hence, our strategy
  - Customize the `jenkins/jenkins:lts` image
  - Run Jenkins in a docker container on a server of our choice
- To build a new docker image, we need to write a recipe, the `Dockerfile`
- It should be created in an otherwise empty directory

The Dockerfile

```
1 FROM jenkins/jenkins:lts
2 USER root
3 RUN apt-get update \
4     && apt-get install -y python3-pip python3 \
5     && rm -rf /var/lib/apt/lists/* \
6     && pip3 install -U pytest tox
7 USER jenkins
```

1. Specify the base image (which itself builds on a debian image)
2. Switch user to enable installing software
3. Update the package repository
4. Install Python3
5. Cleanup
6. Install `pytest` and `tox` (which can run tests in different configurations)
7. Switch back to non-privileged user

Building the Image

- In the directory with the Dockerfile run

```
jenkins-image$ docker build -t jenkins-python .
```

- It can take a while to construct this image; instead we will use a prebuilt image `pthie/testing:jenk1`

Starting the CI-Server

```
$ docker run --rm -p 8080:8080 -p 50000:50000 \
     -v jenkins_home:/var/jenkins_home pthie/testing:jenk1
```

Obtains the requested image and starts it

- `--rm` remove the container on termination
- `-p 8080:8080` Jenkins is configured to listen on port 8080 in the container; this connects the container port to the same port on the host machine
- `-p 50000:50000` (for attaching slave servers)
- `-v jenkins_home:/var/jenkins_home` attaches a volume (host directory) to the container for persistent state
- `pthie/testing:jenk1` name of the image to run

Configuring the CI-Server

- Running the container yields a lot of output
- The important part is this:

```
Jenkins initial setup is required. An admin user has been create
Please use the following password to proceed to installation:

66e82ef484a04725bd0eea067e75e778
```

- Point your browser to `http://127.0.0.1:8080/` to access the Jenkins configuration (you will be asked to the above password)
- (Standard packages are more than sufficient)
- Create a user and login

Getting Jenkins in English

- Jenkins UI uses the browser's default language
- To change that to English
  - "Manage Jenkins" → "Manage Plugins" → ['Available' tab]
  - Check "Locale Plugin" checkbox and "Install without restart" button.
  - "Manage Jenkins" → "Configure System" → "Locale".
  - Enter LOCALE code for English: `en_US`
  - Check "Ignore browser preference and force this language to all users".

Source: https://superuser.com/questions/879392/how-force-jenkins-to-show-ui-always-in-english/882823

Creating a Test Project

- Starting page → "New Item"
- Give it some name, e.g. `python-webcount`
- Select "Free Style Software Project" → "OK"
- "Source code management" → Git → repository URL
  for example: https://github.com/python-ci-cd/python-webcount
  but it's better to clone the repository and work on your own copy
- "Build Trigger" → "Poll SCM"
- Enter `H/5 * * * *` as schedule (check every five minutes)
- "Build" → select "Execute Shell" and enter

```
cd $WORKSPACE
TOXENV=py35 python3 -c 'import tox; tox.cmdline()'
```

- Save the page: everything is up an running!

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Setting up Testing via Circle-CI

- Circle-CI provides CI infrastructure which can be linked to (e.g.) GitHub
- Using it with Python is straightforward:
  - Register with Circle-CI (easiest with your GitHub account)
  - Select a repository to add from the menu
  - Follow the instructions: in the repository add a `.circleci` Directory with a file `config.yml`
  - This file is essentially the "official" Python CircleCI project template
  - Up to a single modification to install `pytest` (next slide)

Config.yml: install dependencies

The last line needs to be added to the "install dependencies" step

```
- run:
    name: install dependencies
    command: |
      python3 -m venv venv
      . venv/bin/activate
      pip install -r requirements.txt
      pip install -U pytest
```

Full file may be found in https://github.com/peterthiemann/python-webcount

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Pytest Features

- Testing the bad case: exceptions
- Depending on external libraries, databases, or the internet
- More on structuring test suites

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Testing the Bad Case: Binary Search

```
def search(item, seq):
    """binary search"""
    left = 0
    right = len(seq)
    while left < right:
        middle = (left + right) // 2
        middle_element = seq[middle]
        if middle_element == item:
            return middle
        elif middle_element < item:
            left = middle + 1
        else:
            right = middle
    raise ValueError("Value␣not␣in␣sequence")
```

- It's common in Python to raise an exception to indicate a failure

pytest.raises: Check that the exception is raised

```
def test_empty ():
    r = search (42, [])
    assert r == 0
```

- Running this test raises an exception, which is reported as a test failure!
- To amend this problem, `pytest` provides a context manager `pytest.raises`, which catches the expected exception `ValueError`:

```
def test_empty ():
    import pytest
    with pytest.raises (ValueError):
        r = search (42, [])
    with pytest.raises (ValueError):
        r = search (0, [1,3,5])
    with pytest.raises (ValueError):
        r = search (4, [1,3,5])
    with pytest.raises (ValueError):
        r = search (60, [1,3,5])
```

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Depending on the Internet
Design for Testability

The Credo of Unit Testing

Unit tests should be efficient, predictable, and reproducible.

Depending on the Internet

Design for Testability

The Credo of Unit Testing

Unit tests should be efficient, predictable, and reproducible.

Design for Testability: Isolation

Tests should be isolated from external resources (internet, databases, etc) because their use may

- cause unpredictable outputs;
- have unwanted side effects (on the resource);
- degrade performance;
- require credentials, which are tricky to manage.

Example with Dependency

```
import requests
def most_common_word_in_web_page(words, url):
    """
    finds the most common word from a list of words in a web page,
    """
    response = requests.get(url)
    text = response.text
    word_frequency = {w: text.count(w) for w in words}
    return sorted(words, key=word_frequency.get)[-1]

if __name__ == '__main__':
    most_common = most_common_word_in_web_page(
            ['python', 'Python', 'programming'],
            'https://python.org/',
            )
    print(most_common)
```

How to Test This Example?

- At the time of writing, this program prints `Python`, but who knows what happens tomorrow?
- A testing environment (in particular on a CI-Server) may not support network connections.
- There are several approaches to testing such examples
  1. Modularity: separate program logic from resource access
     Advantage: always a good idea
     Disadvantage: the actual resource access is never tested
  2. Abstraction and mocking: abstract over the resource and supply a fake resource during testing
     Advantage: can test entire code
     Disadvantage: mocking must accurately mimic the resource's behavior
  3. Patching: overwrite functionality of the resource during testing

Example: Modularity

```
import requests
def most_common_word_in_web_page(words, url):
    response = requests.get(url)
    return most_common_words (words, response.text)

def most_common_words (words, text):
    word_frequency = {w: text.count(w) for w in words}
    return sorted(words, key=word_frequency.get)[-1]

if __name__ == '__main__':
    most_common = most_common_word_in_web_page(
         ['python', 'Python', 'programming'],
         'https://python.org/',
         )
    print(most_common)
```

- Standard unit testing applicable to `most_common_words`

Example: Abstraction and Mocking

- Abstract over the requests module

```python
def most_common_word_in_web_page(words, url, useragent=requests):
    response = useragent.get(url)
    return most_common_words (words, response.text)
```

- For `useragent`, we can supply any object that has a `get` method that returns an object with a `text` field.

```python
def test_with_dummy_classes():
    class TestResponse():
        text = 'aa␣bbb␣c'
    class TestUserAgent():
        def get(self, url):
            return TestResponse()
    result = most_common_word_in_web_page(
        ['a', 'b', 'c'],
        'https://python.org/',
        useragent=TestUserAgent()
    )
    assert result == 'b'
```

Example: Abstraction and Mocking (continued)

- Writing dummy objects can become tedious
- Fortunately, they can be replaced by configurable **mock objects**

```
def test_with_mock_objects():
    from unittest.mock import Mock
    mock_requests = Mock()
    mock_requests.get.return_value.text = 'aa␣bbb␣c'
    result = most_common_word_in_web_page(
        ['a', 'b', 'c'],
        'https://python.org/',
        useragent=mock_requests
    )
    assert result == 'b'
    assert mock_requests.get.call_count == 1
    assert mock_requests.get.call_args[0][0] == 'https://python.org
```

Example: Abstraction and Mocking (continued)

- Mock objects appear quite magical
- `mock_requests.get` creates a new mock object in `mock_requests`'s `get` property
- `mock_requests.get.return_value` implies that this mock is a function that returns another mock object
- `mock_requests.get.return_value.text` ... which in turn has a `text` property
- `mock_requests.get.return_value.text = '...'` ... which is set to a string

A Simple Example with Mocks

```python
from unittest.mock import Mock

def test_mock():
    mock = Mock()
    mock.x = 3
    mock.y = 4
    mock.distance.return_value = 5
    assert mock.x * mock.x + mock.y * mock.y == \
        mock.distance() * mock.distance()
    assert mock.distance.call_count == 2
    mock.distance.assert_called_with()
```

- define two properties `x` and `y`
- define a method `distance`
- check functionality
- check that `distance` is called twice
- check it's called with the right arguments (no arguments)

Example: Patching

Overwrite the functionality of the resource during testing

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End

Separating Tests from User Code

- Tests and application code should live in separate files
- Typical setup
  - application code in a module
  - test code in another module in a different directory
- Customarily, test code lives in a subdirectory called `tests`

Structure of a Python application

- Project name: sample
- Exposes module (package): sample

```
README.md
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.md
conftest.py
tests/test_basic.py
tests/test_advanced.py
```

See https://github.com/kennethreitz/samplemod

Structure of a Python Application

Structure of a Python package

- A package is a module consisting of several source files in a directory
- It should contain a special file `__init__.py`
- Typically this file imports the exposed names from the other files in the directory

Structure of a Python Application

Structure of a Python package

- A package is a module consisting of several source files in a directory
- It should contain a special file `__init__.py`
- Typically this file imports the exposed names from the other files in the directory

Testing a Python Application

- `pytest` is invoked in the root directory
- Recursively looks for file names beginning with `test_` and executes them
- Each test file imports application modules relative to the project root
- `conftest.py` (empty file in the project root) indicates the project root directory to `pytest`

Plan

1 Testing Python
- Let's Test
- pytest
- Testing with Numbers
- Exercise

2 Continuous Testing
- On-Premise Server
- Software as a Service

3 More Testing Secrets
- Testing the Bad Case
- Depending on External Resources
- Structuring Test Suites

4 The End