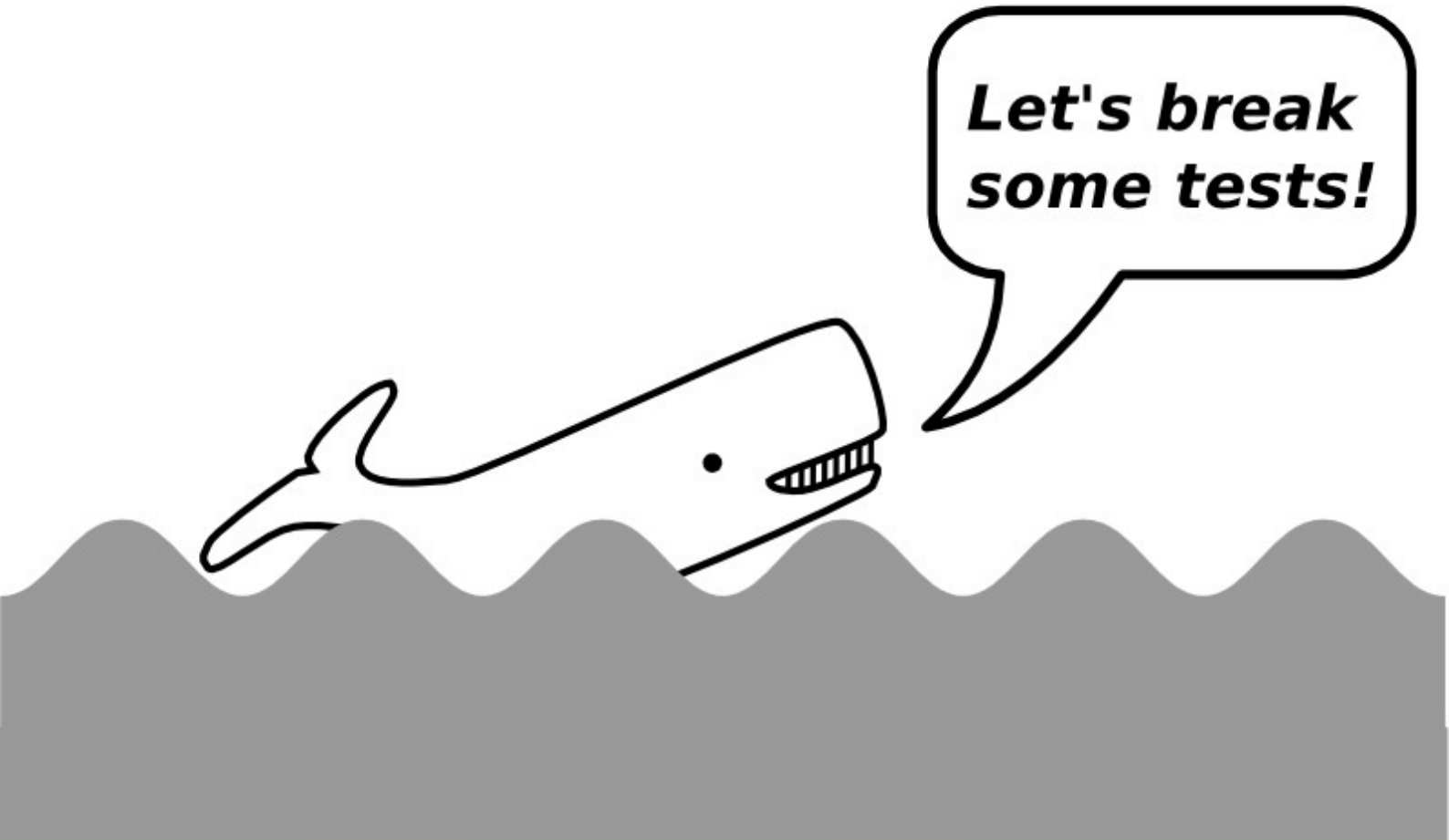
ACADEMIS

Python Testing Tutorial



featuring Melville's Moby Dick

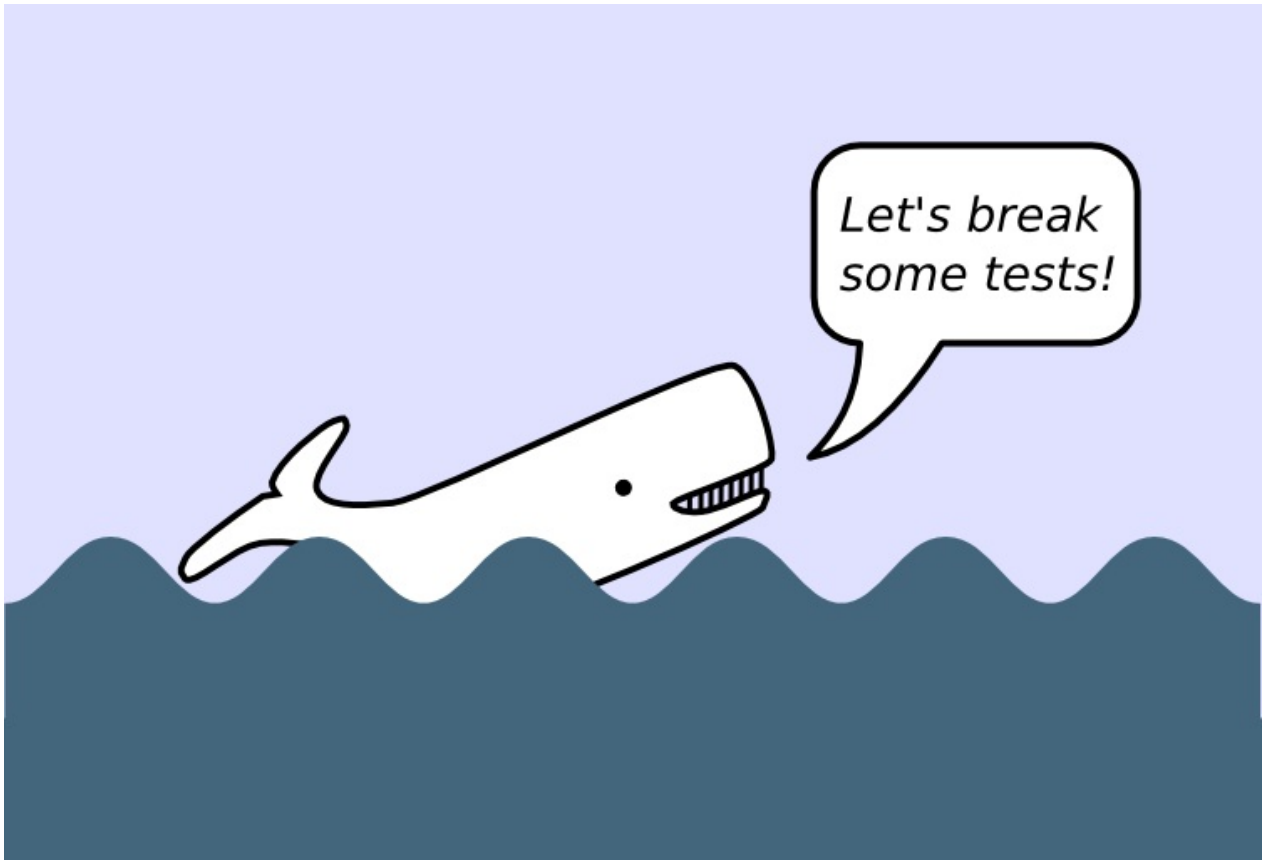
Table of Contents

Python Testing Tutorial

Overview

This tutorial helps you to learn automated testing in Python 3 using the `py.test` framework.

Some material also covers the `unittest` and `nose` frameworks, and most should work on Python 2 as well. Instructions for trainers are included.



Latest version of this book

- Complete material (book + code examples): https://github.com/krother/python_testing_tutorial.
- PDF and EPUB versions: https://www.gitbook.com/book/krother/python-testing-tutorial

Copyright

Feedback and comments are welcome at: krother@academis.eu

Contributors

Kristian Rother, Magdalena Rother, Daniel Szoska

Quotes

"Call me Ishmael"
Herman Melville, Moby Dick 1851

"UNTESTED == BROKEN"
Schlomo Shapiro, EuroPython 2014

"Code without tests is broken by design”
Jacob Kaplan-Moss

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
Brian Kernighan, "The Elements of Programming Style", 2nd edition, chapter 2

"Pay attention to zeros. If there is a zero, someone will divide by it."
Cem Kaner

"If you don’t care about quality, you can't meet any other requirement”
Gerald M. Weinberg

"Testing shows the presence, not the absence of bugs."
Edsger W. Dijkstra

"... we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."
Bill Gates (Information Week, May 2002)

Warming Up

How many words are in the following sentence?

```
The program works perfectly?
```

You will probably agree, that the sentence contains **four words**.

How many words are in the next sentence?

```
That #§&%$* program still doesn't work!\nI already
de-bugged it 3 times, and still numpy.array
keeps raising AttributeErrors. What should I do?
```

You may find the answer to this question less obvious. It depends on how precisely the special characters are interpreted.

What is automated testing good for?

Writing automated tests for your software helps you to:

- get clear on what you want the program to do.
- identify gaps in the requirements.
- prove the presence of bugs (**not their absence!**).
- help you during refactoring.

Unit Tests

Exercise 1: Test a Python function

The function **main()** in the module **word_counter.py** calculates the number of words in a text body.

For instance, the following sentence contains **three** words:

```
Call me Ishmael
```

Your task is to prove that the **main()** function calculates the number of words in the sentence correctly with **three**.

Run the example test in **test_unit_test.py**.

Exercise 2: Test proves if code is broken

The test in the module **test_failing_code.py** fails, because there is a bug in the function **word_counter.average_word_length()**. In the sentence

```
Call me Ishmael
```

The words are **four, two,** and **seven** characters long. This gives an average of:

```
>>> (4 + 2 + 7) / 3.0
4.333333333333333
```

Fix the code in `test_broken_code.py`, so that the test passes.

Exercise 3: Code proves if tests are broken

The test in the module **test_failing_test.py** fails, because there is a bug in the test file.

Your task is to fix the test, so that the test passes. Use the example in **test_broken_test.py**.

Exercise 4: Test border cases

High quality tests cover many different situations. The most common situations for the program **word_counter.py** include:

test case	description	example input	expected output
empty	input is valid, but empty	""	0
minimal	smallest reasonable input	"whale"	1
typical	representative input	"whale eats captain"	3
invalid	input is supposed to fail	777	*Exception raised*
maximum	largest reasonable input	*Melville's entire book*	*more than 200000*
sanity	program recycles its own output	*TextBody A created from another TextBody B*	*A equals B*
nasty	difficult example	"That #~&%* program still doesn't work!"	6

Your task is to make all tests in **test_border_cases.py** pass.

Testing Command-Line Programs

Exercise 1: Test a command-line application

The program **word_counter.py** can be used from the command line to calculate the most frequent words with:

```
python word_counter.py mobydick_summary.txt
```

Command-line applications need to be tested as well. You find tests in **test_commandline.py**.

Your task is to make sure the command-line tests pass.

Exercise 2: Test command-line options

The program **word_counter.py** calculates most frequent words in a test file. It can be used from the command line to calculate the top five words:

```
python word_counter.py moby_dick_summary.txt 5
```

Your task is to develop a new test for the program.

Exercise 3: User Acceptance

The ultimate test for any software is whether your users are able to do what they need to get done.

Your task is to *manually* use the program **word_counter.py** to find out whether Melville used *'whale'* or *'captain'* more frequently in the full text of the book *"Moby Dick"*.

The User Acceptance test cannot be replaced by a machine.

Mock Objects

Exercise 1: Using a Mock Object

The function **word_report.get_top_words()** requires an instance of the class **TextBody**. You need to test the function, excluding the possibility that the **TextBody** class is buggy. To do so, you need to replace the class by a **Mock Object**, a simple placeholder.

Your task is to write a test for the function **word_counter.get_top_words()** that does not use the class **TextBody**.

Test Data

Exercise 1: A module with test data

Create a new module **test_data.py** with a string variable that contains a sentence with lots of special characters:

```
"That #§&%$* program still doesn't work!\nI already de-bugged it 3 times, and still nu
mpy.array keeps raising AttributeErrors. What should I do?"
```

Your task is to write a test for the module **word_count.py** using the string imported from the **test_data** module.

Exercise 2: Preparing tests with fixtures

Sometimes multiple tests need similar preparations. For instance, the tests in **test_word_report.py** require loading the contents of the text file **mobydick_summary.txt**.

Your task is to make sure the code for loading the text file appears only once.

Exercise 3: Sets of example data

You have a list of pairs (data sample, expected result) for the program **count_words.py** that apply to the text **mobydick_summary.txt**:

word	count
months	1
whale	5
captain	4
white	2
harpoon	1
Ahab	1

Your task is to create six tests from these samples. Figure out how more pairs can be added easily. In particular, *don't* copy-paste a new test function for each data sample.

Exercise 4: Write a test with sample data

The module **word_report.py** contains a function to calculate the most frequent words in a text body. It should produce the following top five results for the book in **mobydick.txt**:

position	word
1.	of
2.	the
3.	is
4.	sea
5.	ship

Your task is to write tests for these five positions.

Exercise 5: Import test data in multiple test packages

In a big software project, your tests are distributed to two packages. Both **test_first.py** and **test_second.py** require the variable **MOBYDICK_SUMMARY** from the module **test data.py**. The package structure is like this:

```
testss/
    test_a/
        __init__.py
        test_first.py
    test_b/
        __init__.py
        test_second.py
    __init__.py
    test_data.py
    test_all.py
```

Your task is to make sure that the variable **MOBYDICK_SUMMARY** is correctly imported to both test modules, so that the tests pass for all of:

```
tests/test_a/test_first.py
tests/test_b/test_second.py
tests/test_all.py
```

Test Suites

Exercise 1: Test selection

Your task is to run only the function **test_word_counter.test_simple** from the test suite in **tests/**.

Exercise 2: Test collection

Run all tests for the `mobydick` package in the directory **tests/** with one command. Make the tests pass.

Exercise 3: Integrate a test suite in a Python package

Make it possible to run all tests for the `mobydick` package by typing:

```
python setup.py test
```

Test Coverage

Exercise 1: Calculate Test Coverage

Your task is to calculate the percentage of code covered by automatic tests for the modules **word_counter.py** and **word_report.py**.

Exercise 2: Identify uncovered lines

Your task is to find out which lines of **word_counter.py** are not covered by tests.

Exercise 3: Increase test coverage

Your task is to bring test coverage of **word_counter.py** to 100%.

Testing New Features

Exercise 1: Add new feature: special characters

Add a new feature to the **word_counter.py** program. The program should remove special characters from the text before counting words.

Your task is to prove that the new feature is working.

Exercise 2: Add new feature: ignore case

Add a new feature to the **word_counter.py** program. The program should ignore the case of words, e.g. *'captain'* and *'Captain'* should be counted as the same word.

Your task is to prove that the new feature is working.

Exercise 3: Add new feature: word separators

The program **word_counter.py** does separate words at spaces, but not tabulators. You need to change that.

The following sentence should also contain **four** words:

```
The\tprogram\tworks\tperfectly.
```

Your task is to add a test for this new situation and make it work.

Recap

Match the test strategies with the according descriptions.

test strategy	description
Unit Test	files and examples that help with testing
Acceptance Test	collection of tests for a software package
Mock	relative amount of code tested
Fixture	tests a single module, class or function
Test suite	prepare tests and clean up afterwards
Test data	replaces a complex object to make testing simpler
Test coverage	tests functionality from the users point of view

Test Frameworks in Python

Introduction to the unittest Framework in Python

unittest is a Python framework for writing Unit Tests, Integration Tests, and Acceptance Tests. It mainly provides a class **TestCase** and a **main()** method.

unittest is typically imported with:

```
from unittest import TestCase, main
```

Writing a test class

Test classes should extend TestCase, and contain at least one method starting with test_ . Test methods contain assertions.

TestCase offers many assertion methods (assertEqual, assertAlmostEqual, assertTrue etc.).

```
class AdditionTests(TestCase):

    def test_add(self):
        self.assertEqual(add(3, 4), 7)
```

Running the tests

The **unittest.main** method will look for all classes derived from TestCase that have been imported. It runs all tests inside them and reports.

Typically, you will find main() called in a separate code block:

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

You can run Python test files with unittest without calling main()

```
python -m unittest test_file
```

Note: The name of the test module is spelled without .py

Testing command-line scripts

To test a command-line script call it using a shell command and redirect the output for further evaluation. The simplest way is to use **os.system**:

```
import os
os.system('python myprog.py > out.txt')
```

Discovering tests

```
python -m unittest discover
```

Test data and fixtures

The methods setUp() and tearDown() can be used to prepare testing and clean up afterwards.

Importing test data in multiple packages

When you have many tests distributed to sub-packages, you may want to share test data among them. There are two ways to do so:

Either set the PYTHONPATH variable to the directory with your tests.

Alternatively, patch **sys.path** in a local module test_data.py in each of the sub-packages, so that they import ../test_data.*

Testing Python Applications with nose

"nose extends unittest to make testing easier"

Getting started

Requirements

- Python 2.7

Installing nose

```
sudo easy_install nose
```

Documentation

https://nose.readthedocs.org/en/latest/
https://nose.readthedocs.org/en/latest/writing_tests.html

Assertions in nose

Same syntax as in unittest

The TestCase class works in the same way as with the **unittest** module. You can run your existing tests with nose.

Test functions without classes

You can write tests without subclassing **TestCase**. You can write tests as simple functions:

```
from nose.tools import assert_equal

def test_example():
    assert_equal(1 + 1, 2)
```

List available assert functions

```
import nose.tools
print dir(nose.tools)
```

Running tests

Running a single test module

```
nosetests only_test_this.py
```

In contrast to **unittest**, it is no longer neccessary to include a **main** block in your test file.

Selecting which tests to run

```
nosetests test.module
nosetests another.test:TestCase.test_method
nosetests a.test:TestCase
nosetests /path/to/test/file.py:test_function
```

Test Detection

Running auto-detected tests with nose

```
nosetests
nosetests -v
nosetests --with-doctest
```

Running nose from Python

For instance as part of a setup script

```
import nose
nose.main()
```

Which tests does nose detect automatically?

All tests identified by nose have **'test'** or **'Test'** at a word boundary or following a - or _) and lives in a module that also matches that expression will be run as a test.

The test finder examines Python files and directories that match this pattern. Packages in the current directory are always examined.

Writing a nose configuration file

Create a **.noserc** file in your home directory containing:

```
[nosetests]
verbosity=3
with-doctest=1
```

Calculating test coverage

```
sudo easy_install coverage
nosetests --with-coverage
nosetests --with-coverage --cover-html
cd cover
firefox index.html
```

Instructions for Trainers

Overview

This toolkit helps you to prepare training courses on automated testing in Python. It allows you to create courses with interchangeable

- testing frameworks
- background of participants
- course duration

Our aim is to save you preparation time while leaving room for your own ideas. Most of all, we hope you have fun in your next course.

How to run a course using this toolkit

1. Introduce the Moby Dick Theme to your trainees
2. Copy the code in *code/mobydick* and *code/test_your_framework*.
3. Set the PYTHONPATH environment variable, so that you can do

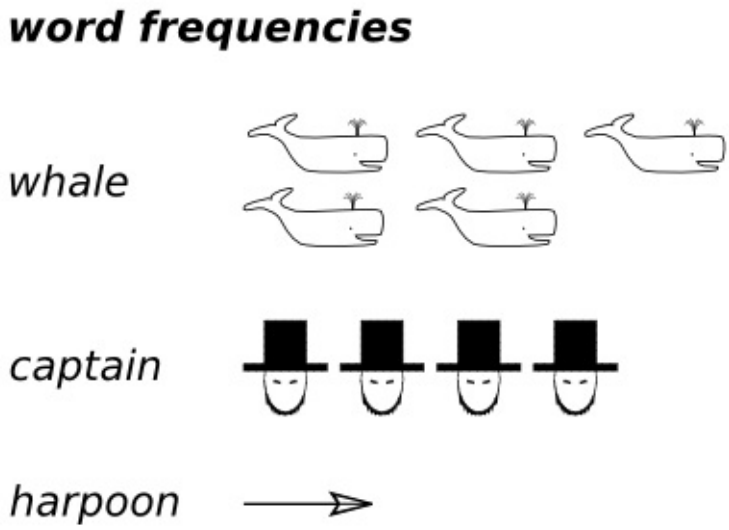
   import mobydick

4. Share the chapter "Challenges" with your trainees.
5. Share the chapter "Reference" on your test framework with your trainees.
6. Start coding!

Counting Words in Moby Dick

Moby Dick: Plot synopsis

Captain Ahab was vicious because Moby Dick, the white whale, had bitten off his leg. So the captain set sail for a hunt. For months he was searching the sea for the white whale. The captain finally attacked the whale with a harpoon. Unimpressed, the whale devoured captain, crew and ship. The whale won.



Video

Moby Dick short synopsis on Youtube

Course Objective

Herman Melville's book *"Moby Dick"* describes the epic fight between the captain of a whaling ship and a whale. In the book, the whale wins by eating most of the other characters.

But does he also win by being mentioned more often?

In this course, you have a program that analyzes the text of Melville's book.

You will test whether the program work correctly?

Why was this example selected?

Three main reasons:

- The implementation is simple enough for beginners.
- Counting words easily yields different results (because of upper/lower case, special characters etc). Therefore the program needs to be thoroughly tested.
- You can easily change the theme to another book from Project Gutenberg.

Lesson Plan for a 45' tutorial

Target audience

Programmers who have already written programs on their own but would like to learn about automated software testing.

Learning Objective

During the tutorial participants will implement automatic test functions that pass for the Moby Dick example. using the unittest module within 20'.

Lesson Plan

module	topic	time
warm-up	hello	1'
warm-up	question: How do you know that your code works?	4'
motivation	explain the benefit: You will be able to check in a few seconds that your program works.	1’
new content	overview of the code example	1’
new content	run the code example; collective analysis	15'
application	write code using the task description	20'
wrap-up	discuss pros and cons of testing	15'
wrap-up	point to materials	2'
wrap-up	goodbye	1'

Lesson plan for a 180' tutorial

I used a very similar lesson plan to conduct a training at EuroPython 2014. The audience consisted of about 60 Python programmers, including beginners and seasoned developers.

module	topic	time
warm-up	introduce the Moby Dick theme	5'
warm-up	icebreaker activity	5'
warm-up	announce training objectives	5'
part 1	**Writing automatic tests in Python**	45’
warm-up	methods in the unittest module	5’
new content	presentation: Unit Tests, Integration Tests, and Acceptance Tests	15’
application	challenges 1.1 - 1.5	20’
wrap-up	Q & A	5’
part 2	**Integration and Acceptance Tests (45')**	
warm-up	quiz on test strategies	10'
new content	presentation on Test-Driven-Development	10’
application	challenges 2.1 - 3.3	20'
wrap-up	Q & A	5’
break		10’
part 3	**Tests data and test suites (45')**	
warm-up	multiple choice questions	10'
new content	presentation on test suites	10’
application	exercises 4, 5, 6	20'
wrap-up	Q & A	5’
summary	**Benefits of testing (25')**	
transfer	group discussion on benefits of testing	20’
finishing	summary	4’
finishing	goodbye	1'