Practical testing with pytest

ASPP APAC 2019 tutorial

By Brianna Laugher

Section 1 - Intro, installing, philosophy
- Discussion
- Activities
  - Install
  - Assert
  - Explore

Section 2 - Running tests (pytest as a test runner)
- Discussion
- Activities
  - Run pytest (with no options)
  - Look at pytest.ini
  - Run: pytest -v
  - Run: pytest --collectonly
  - Run: pytest tests/test_fancy_output.py
  - Run: pytest -k fancy
  - Isolate the failing test
  - Compare --tb options
  - Look at the command line options in pytest --help

Section 3 - Writing tests - basics
- Discussion
- Activities
  - Examine test_palettes.py
  - Fix a failing test
  - Add a new test
  - Use pytest.mark
  - Use pytest.mark.parametrize
  - Use pytest.mark.xfail
  - Examine pytest.raises
  - conftest.py
  - More parametrize

Section 4 - Writing tests - fixtures
- Discussion
- Activities
  - Examine conftest.py

        Create a fixture, ‘mountain’ and use it
        Fixtures
Section 5 - Writing tests - TDD
    Exercises
Section 6 - Extensions
    Discussion
    Activities
        Investigate plugins
        Set up CI
        IDE integration
        Property-based testing with Hypothesis

Section 1 - Intro, installing, philosophy

Discussion

Why are we here? What even is a test? Why test, and why pytest?

What is a test? Usually, a block of code like a function or method that has at least one assert statement in it. Assert statements in Python look like this:

```
assert datetime.datetime.today().month == 7
```
or
```
assert datetime.datetime.today().month == 7, "Isn't it July?"
```

This is equivalent to this:

```
if not(datetime.datetime.today().month == 7):
    raise AssertionError("Isn't it July?")
```

So the first bit is a condition which is evaluated as a Boolean. If it is True, we continue on. If it is False, an exception called AssertionError is raised, with an optional message.

For pytest, any test function/method that raises any exception is considered a FAIL, while any test that does not raise an exception is considered a PASS. There are some other states too, but these are the most important ones.

Why test?

Sarah Mei describes "Five Factor Testing":
1. Verify the code is working correctly
2. Prevent future regressions *("things that used to work, but don't anymore")*
3. Document the code's behavior
4. Provide design guidance
5. Support refactoring

Different tests serve different factors, and sometimes these factors will be in tension with each other too.

Why pytest? Your test suite can easily have more code in it than your source files. Treat it as a first class citizen in your code base, not "write once and forget". I find the pytest approach simple to get started, but endlessly powerful. It also has a great community of contributors and a very healthy ecosystem of plugins. I hope this tutorial will help you "get weirdly into" testing, too.
As Hynek advises, "If you use bad testing tools, you write the majority of your software with bad tools."

Pytest has given me, and I hope if will give you, the chance to "[experience] a novel sense of actually trusting [your] code". (source)

Today we're going to use a real(ish) script, with a starter test suite, to see pytest in action and jump right in. We're using a script which does some image processing.
This will be challenging - reading someone else's code usually is! It's a skill that needs practice, so try not to feel frustrated if you're finding it tough. Help each other!

Activities

Install

```
git clone https://github.com/aspp-apac/2019-testing-code.git
git pull origin master
```

Then create and activate your virtualenv, and install the dependencies:

(If you have installed and are happy to use virtualenvwrapper or pipenv, feel free to use those things instead! A Pipfile is provided.)

```
virtualenv -p python3 aspptesting
source aspptesting/bin/activate
pip install -r requirements.txt
pip install -r requirements-dev.txt
```

Let's confirm that we have things installed as we expect:

Command to run	**Expected output**
`python --version`	3+
`python colorthief.py --help`	Typical help output (no import errors)
`pytest --version`	4.1.1

If you come back to this directory later, or open a new terminal, you should only need to run the command to activate the virtualenv:

```
source aspptesting/bin/activate
```

Assert

If you haven't used the assert statement before, practice using it in a Python shell. What output do you get if you type these statements?

```
import datetime
assert datetime.datetime.today().month == 1
assert datetime.datetime.today().month == 1, "Isn't it January?"
assert datetime.datetime.today().month == 2, "Isn't it January?"
```

Explore

Use the application in your terminal. What different kinds of input does it take? Try it on some images that have on your computer. Does the output seem reasonable? Do you notice any errors or mistakes? Can you imagine what input would cause an error? What kind of things can you imagine being important to test?

There are some sample images in the 'images' directory that you can use for exploratory testing.

You should see output like this:

```
python colorthief.py images/sunset.jpg --fancy
```

Section 2 - Running tests (pytest as a test runner)

Discussion

Before we leap into writing tests, it's good to become familiar with pytest as a test runner. It has lots of useful options. When you are working in your codebase it's good to have a constant loop of *write code - run tests - write code - run tests*. But you want to only run the relevant tests, and also get any output about why a test is failing as quickly as possible.

Pytest will do test discovery or collection, execute the tests and then print the results. Failed tests will have a brief traceback output showing where an exception occurred.

We are going to explore different command-line options and see how they change which tests are executed, and how they change the output.

My most commonly used options. I use these every day. We're going to explore most of them in the activities.	Other super useful options - come back to these later.
• `--collectonly` • `-k TEST_NAME_FOO` • `-m MARKER_FOO` • `-x, --exitfirst` • `-s` • `--tb=short` • `-v`	• `--lf, --last-failed` • `-l, --showlocals` • `--junit-xml=path/to/file.txt` • `-rxs` • `--strict` • `--pdb`

Activities

Run pytest (with no options)

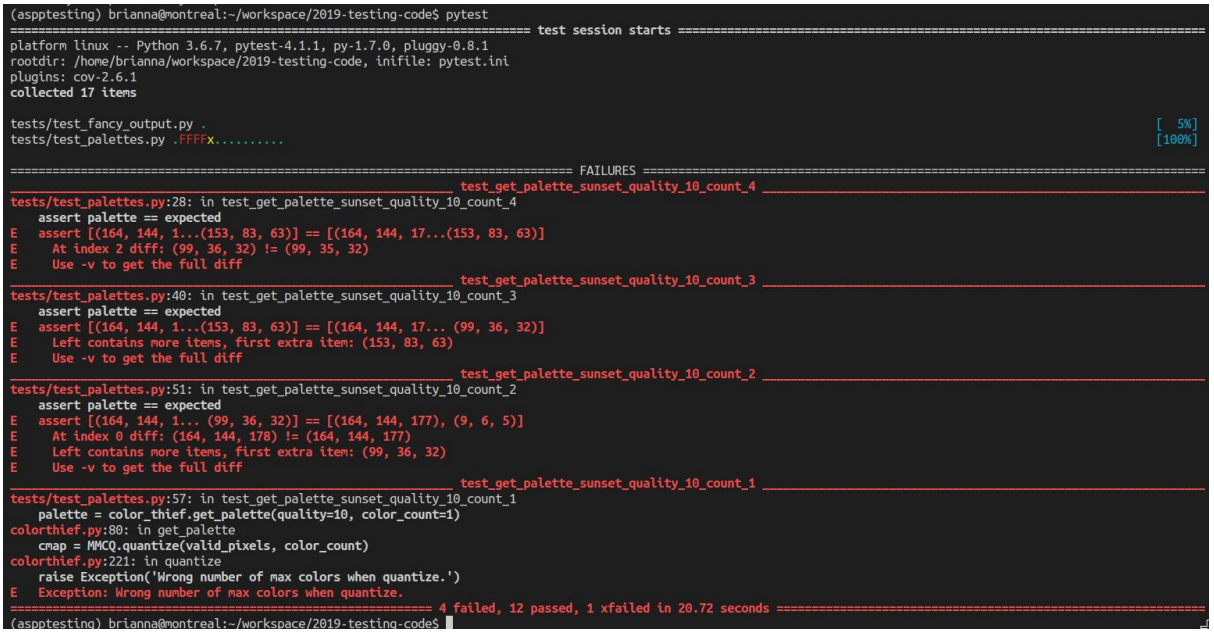
At the command-line, type:

```
pytest
```

If you encounter any import errors, try this instead:

```
python -m pytest
```

You should see some output, something like this:



After each file name there is a series of dots (.) - each one represents a passing test. There is also an 'x'! And some 'F's! And you may have other letters.

After the filenames, information about any test failures is printed out.

Look at pytest.ini

There is a file called pytest.ini which has some configuration options for pytest. At the moment we have set a few including:

- addopts - this tells pytest to always run with this option, as if we had typed it
- testpaths - this tells pytest where to look for tests. By default, if you don't supply a path, it will look in every folder under the current folder. That can be annoying if your virtualenv is located inside your project folder, you probably don't want to run all the tests for every package in your virtualenv! So we are specifying some folders to avoid that happening.

Run: `pytest -v`

-v = verbose. This makes pytest print out the name of every test, and "PASSED", rather than just a dot.

I prefer this greatly while developing, as I can verify that the test I am writing is actually being executed!

Maybe you would like to add it to pytest.ini? Change the addopts line to say:

```
addopts = --tb=short -v
```

Save the file, then run `pytest` again (without -v). Output will now always be verbose!

Run: `pytest --collectonly`

This just 'collects' the tests and doesn't execute them. It is useful in conjunction with the -k option below.

Run: `pytest tests/test_fancy_output.py`

In this example, we are passing a path that we want to be used for test discovery. So a limited subset of tests will be run.

Run: `pytest -k fancy`

-k is "keyword". This will only select tests that have the word 'output' somewhere in the path or test name. In this case it will be pretty much the same set of tests as passing the path above.

It also has some fancy things, like you can do `pytest -k 'not output'`
Then it will select all the tests that *don't* match 'output'.

Check what *would* be run, by combing -k with the --collectonly option:
`pytest -k 'not output' --collectonly`

Isolate the failing test

There is a test called test_get_palette_sunset_quality_10_count_1 that is failing. Can you find a way to run pytest so that it only runs this test? (Don't worry, we are going to fix it later!)

Hint: *[highlight this line to see the hint]*

Compare `--tb` options

--tb is the option for "traceback print mode (auto/long/short/line/native/no)". This is what is output when a test fails.

Try running the failing test with different options and compare the output, e.g:
`pytest -k test_get_palette_sunset_quality_10_count_1 --tb=line`

--tb=short is my personal favourite, which is why I have added it to the pytest.ini. But feel free to remove it or change it if you prefer a different style.

Look at the command line options in `pytest --help`

There are so many options!! You might find it easier to copy the output into a file to read later.

Try to find what options you need to do the following:

- get a code coverage report
  "Code coverage" means "what percentage of my code got executed when I run the tests?" The more code tested = the closer to 100%.
  (This option comes from the pytest-cov plugin.)
  Answer: *[Highlight to read]*
  E.g. try this: *[Highlight to read]*
- get a report on the 5 slowest tests
  Answer: *[Highlight to read]*

Section 3 - Writing tests - basics

Discussion

A recap on decorators.
The @ symbol at the start of a line, applies a decorator to the function/method that follows. The decorator “wraps” the function and has many different uses.

```
@pytest.mark.xfail()
def test_what_month_is_it():
      today = datetime.datetime.today()
      assert today.month == 1, "Isn't it January?"
```

In this example, ‘pytest.mark.xfail’ is the decorator, that is wrapping the function test_what_month_is_it.

For today we are not concerned with “how” decorators work, just trust that they attach some metadata to the function/method that immediately follows them.

Functions/methods can also have more than one decorator applied to them.In this case, each one is on its own line. Example:

```
@pytest.mark.xfail()
@pytest.mark.january
@pytest.mark.today
def test_what_month_is_it():
      today = datetime.datetime.today()
      assert today.month == 1, "Isn't it January?"
```

How does pytest recognise tests?
By default it looks for:

- Filename: test_foo.py or foo_test.py
- Function: name test_foo
- Class: TestFoo, with no __init__ method
    - Methods: name test_foo

If you have a class in your source code called “TestX” or “XTest” it will probably cause you some pain! These patterns can also be configured, though.

Basic features
Yep, these are mostly decorators!

- pytest.mark
    - Ad-hoc tags to label or group tests

    - These can be used to build more powerful things on top of them.
- pytest.mark.skipif
    - Conditions to skip (not run) tests
    - Useful if a test only applies to certain environments (eg Python version, operating system)
- pytest.mark.parametrize
    - Feeding different input/output through the same test steps
    - Extremely powerful and succinct. **#1 pytest feature!**
- pytest.mark.xfail
    - When a test is expected (required) to fail
    - Your test should always show what is the “correct” behaviour, even if it is not the “current” behaviour. If your test reproduces a bug but you are not fixing the bug yet, mark the test as xfail. The person who comes to fix the bug later will love you!
- pytest.raises
    - Context manager (‘with’ statement) for when you are expecting an exception
- pytest.ini and conftest.py
    - pytest.ini - we’ve already seen - for configuration
    - conftest.py - for test building blocks like fixtures, automatically available to all tests.

Pytest result types:

.	PASSED	No exception raised during test execution
F	FAILED	An exception was raised during execution *Could be an AssertionError from an assert statement, or any other exception*
s	SKIPPED	Test was collected but not executed as it was marked to be skipped *e.g. test is for Windows, but currently environment is Linux*
x	xfail	Test was marked as “expected to fail”, and it did fail
X	XPASS	Test was marked as “expected to fail”, but it passed! 🤔

Activities

Examine test_palettes.py

Open colorthief.py (locate for the get_palette method) and tests/test_palettes.py side by side.

Note how the test file is much longer than the method! This is not unusual.

Fix a failing test

I don’t know about you but that failing test test_get_palette_sunset_quality_10_count_4 is bugging me!! Can we figure out how to fix it?!

Answer:

[Highlight to read]

Add a new test

Add a test_get_palette_mountain for a different input image. How will you calculate the expected output?
Be sure to run the test, too! Does it pass?

Answer/example:

[Highlight to read]

Use pytest.mark

Some of the test functions have decorators in the style `pytest.mark.foo`. These are arbitrary “tags” that we can use to group tests.

The -m option can be used to select a subset of tests based on marks. Run `pytest -m sunsetimage` and see which tests are executed. Are they the ones you expected?

Add a similar mark to your new test. Be sure to run `pytest -m your_new_mark --collectonly` to check that it is working correctly.
Example: *[Highlight to read]*

Use pytest.mark.parametrize

Near the bottom of the file, there is a test_get_palette_count function with a multi-line decorator called pytest.mark.parametrize.

Run this test by running: `pytest -k test_get_palette_count -v`

How many tests are executed? Yep, it’s 10, not 1!

Notice that in the test method, there are arguments `quality` and `expected_count`, and in the parametrize there is also a tuple `('quality', 'expected_count')`. These arguments are getting “fed in” from the parametrize mark. The strings/names always need to match exactly, otherwise pytest will report an error.

Add a new test to the parametrize. After adding it run the above command again and make sure it is getting executed!

Use pytest.mark.xfail

test_get_palette_count_10 has an xfail mark on it. Try removing or commenting out this line and executing this test. How does the output differ?

Answer:

[Highlight to read]

Examine pytest.raises

Look for a test called test_get_palette_sunset_quality_10_count_1. It fails as an exception is raised. Where is the exception raised from?

Given the input we provided, it seems that the code is telling us color_count=1 is not a reasonable value. So what if we invert our test - to say that if we specify color_count=1, we do expect an exception to be raised? (Use the `pytest.raises` context manager.)

Answer:

[Highlight to read]

conftest.py

Open this file and examine it.
At the moment there is a function called sunset which has a @pytest.fixture decorator.
Intriguing… we will return to this soon!

More parametrize

Can you imagine how to collapse the test_get_palette_sunset_quality_10_count_*n* test functions into a single test function using parametrize? If you want to try it, start by converting/combining just one or two.

Answer/hint:

[Highlight to read]

Section 4 - Writing tests - fixtures

Discussion

Test structure

Parts of a test		**Naive test structure**
Given (preconditions)	Test setup	`def test_foo_with_bar(foo):`

When (action)	Test body	`thing = do_setup()`
		`result = foo.method(bar)`
		`assert result == expected`
		`thing.do_cleanup()`

Then (expected)	Test body (assertion)	
[cleanup]	Test cleanup/teardown	

Naive test structure	**Pytest fixture structure**

`def test_foo_with_bar(foo):`
`  thing = do_setup()`
`  result = foo.method(bar)`
`  assert result == expected`
`  thing.do_cleanup()`
	`@pytest.fixture`
	`def foo():`
	`  thing = do_setup()`
	`  yield thing`
	`  thing.do_cleanup()`
	
	`...`
	
	`def test_foo_with_bar(foo):`
	`  result = foo.method(bar)`
	`  assert result == expected`

😫 Cleanup is not executed if the test fails 😕 Test body is harder to read	😁 Fixture can easily be reused in multiple tests (Don't Repeat Yourself) 😊 Fixture scope can be controlled (to only run setup once per session, if appropriate) 😍 A test can call on multiple fixtures, for easy separation of concerns

What is a fixture?
It's something your test needs to run.
Could be some data in Python or in the database, or the database itself.
In Django it tends to mean "json of data to feed into database for this test", but in pytest it can be broader.
Often have a notion of 'setup' and 'teardown'/'cleanup'.

In pytest, they have a defined scope - function, class, module, session - and they can be built on top of each other.

Fixtures separate the set-up of your test from your actual test code, making it easier to "get to the point".

There are some builtin fixtures (such as monkeypatch) and many plugins supply them, so you don't need to write them!

Fixture in action
This test in tests/test_fancy_output.py, has a fixture called capsys ("capture sys.stdout/sys.stderr"):

```
def test_analyse_image_not_fancy(capsys):
     imgpath = 'images/sunset.jpg'
     analyse_image(imgpath, do_print_fancy=False)
     expected_stdout = """(163, 143, 178)
(9, 6, 5)
(99, 35, 32)
(246, 222, 171)
(151, 82, 64)
"""
     captured = capsys.readouterr()
     assert captured.out == expected_stdout
```

Generally, they look like arguments to test functions that make you wonder, "hey, where is that coming from?"

The capsys fixture is supplied by pytest itself. It captures/catches the stdout (ie print statements) and stderr output that is written while your test runs, and lets you make asserts about it.

Activities

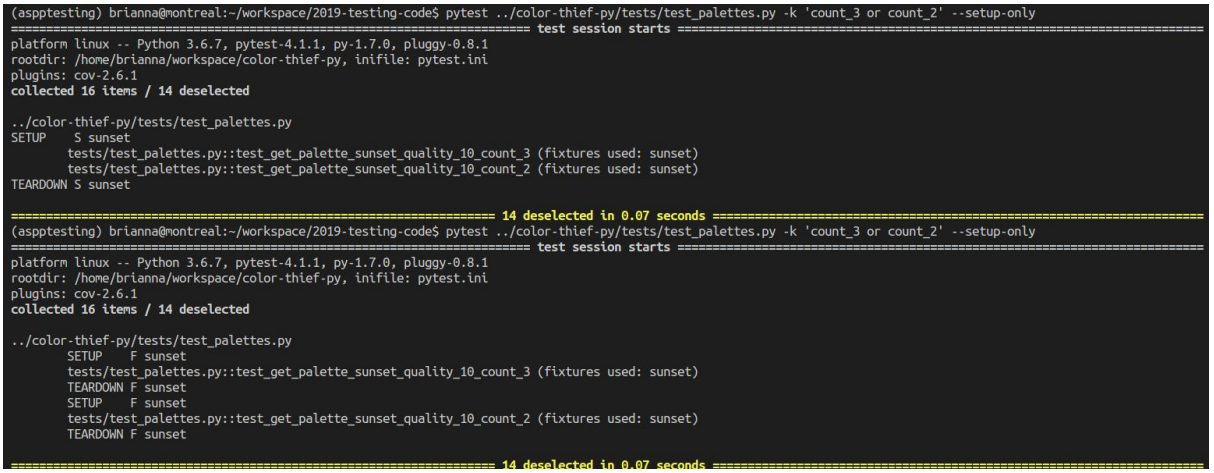
Examine conftest.py

This is where we can write our own fixtures. We can then access them in our test files without needing to import them. There is one fixture already defined, "sunset". Notice that we use a decorator to 'declare' our fixture to pytest.

Which tests is it being used in?

Can you refactor more tests to make use of it?

The sunset fixture is defined with `scope="session"`. If we run pytest with `--setup-only`, pytest will show us when it will execute fixture setup and teardown. What difference do you see if you change the scope to the default value, "function"?



Create a fixture, ‘mountain’ and use it

Refactor the test you created for mountain.jpg previously to make use of your fixture.

By the way - you can put your fixtures in the test file if you want, as long as you use the @pytest.fixture decorator. But putting them in conftest.py means they can be used in all test files without importing.

Fixtures

What kind of things would make sense to build as fixtures in the code you normally write?

Section 5 - Writing tests - TDD

Test driven development!

"Asserts before reverts": a good philosophy. Write tests (assert statements), so that you don't have to remove (revert) your code after it fails.

Monkeypatch or DI?
It's not really one vs the other, they even make sense to use together. Nevertheless…

Monkeypatch
- Safely replace a method/attribute on the 'real' object
- "Safely" = this fixture will clean up after itself
- Excellent for exercising error conditions
- Thanks, interpreted language environment!

Dependency injection
- When you can refactor your code, or it's nicely designed to start with!
- Could be 'injecting' a light weight 'real' thing, or a monkeypatched 'real' thing, or a mock object

Caveat emptor:
Too much monkeypatch, or too much mock, can make your tests brittle.
DI with lightweight objects can miss bugs where those objects differ to the 'real' ones. eg SSL

So what about mock
Mocking, in general, is an approach where you create an object that accepts whatever is thrown at it. Any method you call, any attribute, any property, it will answer the call.

Mocking works well in Python because we tend to follow 'duck typing'. "If it walks like a duck and it quacks like a duck, then it must be a duck."
In our application - *"Do I care if I have a 'real' Location instance or do I just want something with a .country() method?"*

To be honest, in 8 years of using Python as my main programming language at work and 5+ years of pytest, I don't use mock. Monkeypatch gets me where I want to be. I am sure there is a reason it is so popular, though!

Exercises

Use the TDD “Red Green Refactor” idea to add the following features to analyse_image:

- options to specify quality and color_count
- an option to output hex values instead of RGB

Section 6 - Extensions

Discussion

This section is a grab-bag, see what interest you the most.

Plugins.
Plugins can modify test collection, execution or output. They often give you new command-line options or fixtures. A good starting point: http://plugincompat.herokuapp.com

We have already been using:

- pytest-cov

Also check out:

- pytest-bdd - for behaviour driven development/Cucumber/Gherkin style tests (Given When Then), most commonly used with web applications
- Pytest-splinter - for controlling a browser for testing a web application
- Pytest-django - for Django applications
- Pytest-catchlog - for asserting on log file contents
- pytest-xdist - run tests in parallel/distributed (this is packaged with pytest core)
- pytest-sugar - Progress bar and enhanced output from running tests.
- pytest-pep8 - Causes a test run to fail if a source file violates the PEP8 style guide.
- Also pytest-pylint, pytest-flakes
- There is probably a plugin for your favourite framework - django, flask, twisted, pyramid, aiohttp, asyncio, qt...

Continuous Integration.
“The tests must run, and pass, on every commit to a source code repository”
A CI server enforces this!
It keeps your test suite passing.
Jenkins, Travis CI, Bamboo (Atlassian), Gitlab CI…
Travis CI can be easily configured/integrated with your Github repository.

Activities

Investigate plugins

What do you think would be difficult to test? See if there is a pytest plugin for it.

Set up CI

Set up Travis CI for your fork.

IDE integration

Look for a plugin to run pytest from your IDE. Can you make it run every time you save a change to any file? (Hint, also check the pytest command-line options.)

Property-based testing with Hypothesis

A starting point to fuzz testing. Instead of thinking of test cases by hand, let Hypothesis find the troubling cases for you. https://hypothesis.works/