Beginner's Python Cheat Sheet - Testing Your Code

Why test your code?

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know whether or not you've broken existing behavior by running your tests.

A *unit test* verifies that one specific aspect of your code works as it's supposed to. A *test case* is a collection of unit tests which verify that your code's behavior is correct in a wide variety of situations.

The output in some sections has been trimmed for space.

Testing a function: a passing test

The `pytest` library provides tools for testing your code. To try it out, we'll create a function that returns a full name. We'll use the function in a regular program, and then build a test case for the function.

A function to test

Save this as full_names.py

```
def get_full_name(first, last):
    """Return a full name."""

    full_name = f"{first} {last}"
    return full_name.title()
```

Using the function

Save this as names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Installing pytest

Installing pytest with pip

```
$ python -m pip install --user pytest
```

Testing a function (cont.)

Building a testcase with one unit test

To build a test case, import the function you want to test. Any functions that begin with `test_` will be run by `pytest`. Save this file as test_full_names.py.

```
from full_names import get_full_name

def test_first_last():
    """Test names like Janis Joplin."""
    full_name = get_full_name('janis',
            'joplin')
    assert full_name == 'Janis Joplin'
```

Running the test

Issuing the `pytest` command tells `pytest` to run any file beginning with test_. `pytest` reports on each test in the test case.

The dot after test_full_names.py represents a single passing test. `pytest` informs us that it ran 1 test in about 0.01 seconds, and that the test passed.

```
$ pytest
============= test session starts =============
platform darwin -- Python 3.11.0, pytest-7.1.2
rootdir: /.../testing_your_code
collected 1 item

test_full_names.py .                      [100%]
============== 1 passed in 0.01s ==============
```

Testing a function: A failing test

Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.

Modifying the function

We'll modify `get_full_name()` so it handles middle names, but we'll do it in a way that breaks existing behavior.

```
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

Using the function

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test

When you change your code, it's important to run your existing tests. This will tell you whether the changes you made affect existing behavior.

```
$ pytest
============= test session starts =============
test_full_names_failing.py F              [100%]

================== FAILURES ===================
_______________ test_first_last _______________
>       full_name = get_full_name('janis',
                'joplin')
E       TypeError: get_full_name() missing 1
        required positional argument: 'last'

=========== short test summary info ===========
FAILED test_full_names.py::test_first_last...

============== 1 failed in 0.04s ==============
```

Fixing the code

When a test fails, the code needs to be modified until the test passes again. Don't make the mistake of rewriting your tests to fit your new code, otherwise your code will break for anyone who's using it the same way it's being used in the failing test.

Here we can make the middle name optional:

```
def get_full_name(first, last, middle=''):
    """Return a full name."""

    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"

    return full_name.title()
```

Running the test

Now the test should pass again, which means our original functionality is still intact.

```
$ pytest
============= test session starts =============
test_full_names.py .                      [100%]

============== 1 passed in 0.01s ==============
```

Adding new tests

You can add as many unit tests to a test case as you need. To write a new test, add a new function to your test file. If the file grows too long, you can add as many files as you need.

Testing middle names

We've shown that `get_full_name()` works for first and last names. Let's test that it works for middle names as well.

```
from full_names import get_full_name

def test_first_last():
    """Test names like Janis Joplin."""
    full_name = get_full_name('janis',
            'joplin')
    assert full_name == 'Janis Joplin'

def test_middle():
    """Test names like David Lee Roth."""
    full_name = get_full_name('david',
            'roth', 'lee')
    assert full_name == 'David Lee Roth'
```

Running the tests

The two dots after test_full_names.py represent two passing tests.

```
$ pytest
============= test session starts =============
collected 2 items
test_full_names.py ..                     [100%]

============== 2 passed in 0.01s ==============
```

A variety of `assert` statements

You can use `assert` statements in a variety of ways, to check for the exact conditions you want to verify.

Verify that a==b, or a != b

```
assert a == b
assert a != b
```

Verify that x is True, or x is False

```
assert x
assert not x
```

Verify an item is in a list, or not in a list

```
assert my_item in my_list
assert my_item not in my_list
```

Running tests from one file

In a growing test suite, you can have multiple test files. Sometimes you'll only want to run the tests from one file. You can pass the name of a file, and pytest will only run the tests in that file:

```
$ pytest test_names_function.py
```

Testing a class

Testing a class is similar to testing a function, since you'll mostly be testing its methods.

A class to test

Save as account.py

```
class Account():
    """Manage a bank account."""

    def __init__(self, balance=0):
        """Set the initial balance."""
        self.balance = balance

    def deposit(self, amount):
        """Add to the balance."""
        self.balance += amount

    def withdraw(self, amount):
        """Subtract from the balance."""
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as test_accountant.py.

```
from account import Account

def test_initial_balance():
    """Default balance should be 0."""
    account = Account()
    assert account.balance == 0

def test_deposit():
    """Test a single deposit."""
    account = Account()
    account.deposit(100)
    assert account.balance == 100
```

Running the test

```
$ pytest
============= test session starts =============
collected 2 items
test_account.py ..                        [100%]

============== 2 passed in 0.01s ==============
```

When is it okay to modify tests?

In general you shouldn't modify a test once it's written. When a test fails it usually means new code you've written has broken existing functionality, and you need to modify the new code until all existing tests pass.

If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out, and no one is using your code yet.

Using fixtures

A fixture is a resource that's used in multiple tests. When the name of a fixture function is used as an argument to a test function, the return value of the fixture is passed to the test function.

When testing a class, you often have to make an instance of the class. Fixtures let you work with just one instance.

Using fixtures to support multiple tests

The instance `acc` can be used in each new test.

```
import pytest
from account import Account

@pytest.fixture
def account():
    account = Account()
    return account

def test_initial_balance(account):
    """Default balance should be 0."""
    assert account.balance == 0

def test_deposit(account):
    """Test a single deposit."""
    account.deposit(100)
    assert account.balance == 100

def test_withdrawal(account):
    """Test a deposit followed by withdrawal."""
    account.deposit(1_000)
    account.withdraw(100)
    assert account.balance == 900
```

Running the tests

```
$ pytest
============= test session starts =============
collected 3 items
test_account.py ...                       [100%]

============== 3 passed in 0.01s ==============
```

`pytest` flags

`pytest` has some flags that can help you run your tests efficiently, even as the number of tests in your project grows.

Stop at the first failing test

```
$ pytest -x
```

Only run tests that failed during the last test run

```
$ pytest --last-failed
```

Weekly posts about all things Python
mostlypython.substack.com