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 that you haven't broken existing behavior.

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 your code's behavior in a wide variety of situations.

Testing a function: A passing test

Python's unittest module 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

```python
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function
Save this as names.py

```python
from full_names import get_full_name

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

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

Testing a function (cont.)

Building a testcase with one unit test
To build a test case, make a class that inherits from unittest.TestCase *and write methods that begin with* test_*. Save this as test_full_names.py*

```python
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

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

unittest.main()
```

Running the test
Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

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.*

```python
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                middle, last)
    return full_name.title()
```

Using the function

```python
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 affected existing behavior.

```
E
==========================================
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
------------------------------------------
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
  positional argument: 'last'


------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

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.) Here we can make the middle name optional.

```python
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                                middle, last)
    else:
        full_name = "{0} {1}".format(first,
                                last)
    return full_name.title()
```

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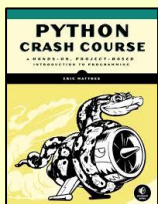*Now the test should pass again, which means our original functionality is still intact.*

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse

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 method to your test case class.

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.*

```python
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

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

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

unittest.main()
```

Running the tests
The two dots represent two passing tests.

```
..
----------------------------------
Ran 2 tests in 0.000s

OK
```

A variety of assert methods
Python provides a number of assert methods you can use to test your code.

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

```python
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that x is True, or x is False

```python
assertTrue(x)
assertFalse(x)
```

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

```python
assertIn(item, list)
assertNotIn(item, list)
```

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

A class to test
Save as accountant.py

```python
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        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.

```python
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

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.

The setUp() method
When testing a class, you usually have to make an instance of the class. The `setUp()` *method is run before every test. Any instances you make in* `setUp()` *are available in every test you write.*

Using setUp() to support multiple tests
The instance `self.acc` *can be used in each new test.*

```python
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def setUp(self):
        self.acc = Accountant()

    def test_initial_balance(self):
        # Default balance should be 0.
        self.assertEqual(self.acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)

unittest.main()
```

Running the tests

```
...
----------------------------------
Ran 3 tests in 0.001s

OK
```