Monday: Unittest

Unittest

Before we begin writing tests, let us outline the different behaviors we want in our application

B.D.D

We want our application to:

1. Allow us to Create new contacts with properties.
2. Save contacts.
3. Display contacts.
4. Delete contacts.
5. Display contact information.

Let us create tests for these behaviors:

Create a file in our Contact-List folder and name it *contact_test.py*. This is where we will write all the tests for our `Contact` class.

contact_test.py

```
import unittest # Importing the unittest module
from contact import Contact # Importing the contact class

class TestContact(unittest.TestCase):

    '''
    Test class that defines test cases for the contact class behaviours.

    Args:
        unittest.TestCase: TestCase class that helps in creating test cases
    '''
```

To create tests in python, we first import a Python test framework called `unittest`. We also import the `Contact` class.

We then create a **subclass** class called `TestContacts`, that inherits from `unittest.TestCase`. A subclass is like a normal class but in addition to its own variables and method it also **inherits** methods and variables from another class.

We pass in the **parent class** as a parameter in the parenthesis.

First test

The first thing we want to test on, is if our objects are being instantiated correctly.

```
    # Items up here .......

    def setUp(self):
        '''
        Set up method to run before each test cases.
        '''
        self.new_contact = Contact("James","Muriuki","0712345678","james@ms.com") # create co
ntact object


    def test_init(self):
        '''
        test_init test case to test if the object is initialized properly
        '''

        self.assertEqual(self.new_contact.first_name,"James")
        self.assertEqual(self.new_contact.last_name,"Muriuki")
        self.assertEqual(self.new_contact.phone_number,"0712345678")
        self.assertEqual(self.new_contact.email,"james@ms.com")


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

`setUp` Method

The `setUp()` method allows us to define instructions that will be executed before each test method.

We have instructed our `setUp()` method to create a new instance of `Contact` class, before each test method declared, and stores it in an instance variable in the test class `self.new_contact`.

Tests are defined with methods that start with `test_`, (this is just proper convention to define your tests).

We created a test called `test_instance` to check if all our objects are instantiated correctly.

We see some new syntax here `self.assertEqual()` this is a `TestCase` method that checks for an expected result. The first argument is the expected result and the second argument is the result that is actually gotten. Here, we are checking if the name and description of our new object is what we actually inputted.

if `__name__` == `'__main__'`

Python runtime has special attributes delimited with double underscores `(_)`

`__name__` variable evaluates to `"__main__"` or the actual module name depending on how the module is being executed

So by defining the condition `if __name__ == '__main__'`: we are confirming that anything inside the if block should run only if this is the file that is currently being run. **Click here (https://stackoverflow.com/questions/419163/what-does-if-name-main-d)** to read more about this

Finally the `unittest.main()` provides a command line interface that collects all the tests methods and executes them.

Now when we run this code

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

OK
```

This test passed because we had already implemented our code for creating a new instance in our `Contact` class. Let us see how to implement the Red-Green-Refactor Method .

Second Test

Apart from creating contacts we would like to save them. Let us write a test for that.

contact_test.py

```
    def test_save_contact(self):
        '''
        test_save_contact test case to test if the contact object is saved into
         the contact list
        '''
        self.new_contact.save_contact() # saving the new contact
        self.assertEqual(len(Contact.contact_list),1)

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

Here we created a test called `test_save_contact` that calls a `save_contact` method to our newly generated object.

Then we check the length of our `contact_list` list to confirm an addition has been made to our contact list.

When we run this

```
.E
======================================================================
ERROR: test_save_contact (__main__.TestContacts)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "contact_test.py", line 31, in test_save_contact
    self.contact.save_contact()
AttributeError: 'Contact' object has no attribute 'save_contact'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (errors=1)
```

We get an error. `AttributeError: 'Contact' object has no attribute 'save_contact'` This is a good thing because we want our test to fail. We can now add some code in our `contact.py` to make this pass.

contact.py

```
contact_list = [] # Empty contact list
 # Init method up here
    def save_contact(self):

        '''
        save_contact method saves contact objects into contact_list
```

```
        '''

        Contact.contact_list.append(self)
```

We create a `save_contact` method that is called on a contact object and appends it to our `contact-list` list.

So now when we run the test file again.

```
python3.6 contact_test.py

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

OK
```

The third test

```
# Items up here...

    def test_save_multiple_contact(self):
            '''
            test_save_multiple_contact to check if we can save multiple contact
            objects to our contact_list
            '''
            self.new_contact.save_contact()
            test_contact = Contact("Test","user","0712345678","test@user.com") # new contact
            test_contact.save_contact()
            self.assertEqual(len(Contact.contact_list),2)

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

We add another test case `test_save_multiple_contact` to test if we can save multiple contacts in our contact list.

We create a new contact object called `test_contact` and save it. Then we check if the length of our contact list is equal to the number of contacts saved.

when we run this

```
..F
======================================================================
FAILED (failures=1)
```

Here we get an assertion error we expected to get `2` but we got `3`.

We are getting this error because every time we run each test we are creating instances of contact and saving them and all of them are being counted.

`tearDown` method

```
# setup and class creation up here
    def tearDown(self):
            '''
            tearDown method that does clean up after each test case has run.
            '''
            Contact.contact_list = []

# other test cases here
```

```
    def test_save_multiple_contact(self):
            '''
            test_save_multiple_contact to check if we can save multiple contact
            objects to our contact_list
            '''
            self.new_contact.save_contact()
            test_contact = Contact("Test","user","0712345678","test@user.com") # new contact
            test_contact.save_contact()
            self.assertEqual(len(Contact.contact_list),2)

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

Just like the `setUp()` method the `tearDown()` method executes a set of instructions after every test.

In the `tearDown()` method we assign the `contact_list` list in the `Contact` class as an empty list. This helps us get accurate test results every time a new test case.

When we run this now

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Fourth Test

```
# More tests above
    def test_delete_contact(self):
            '''
            test_delete_contact to test if we can remove a contact from our contact list
            '''
            self.new_contact.save_contact()
            test_contact = Contact("Test","user","0712345678","test@user.com") # new contact
            test_contact.save_contact()

            self.new_contact.delete_contact()# Deleting a contact object
            self.assertEqual(len(Contact.contact_list),1)
if __name__ == '__main__':
    unittest.main()
```

Now we want to add the feature to delete contacts . We create a test for that behavior then we confirm if the length of the `contact_list` list is equal to the number of saved contacts.

Let us run this to confirm it fails. Then let us write the code to make the test pass.

```
    def delete_contact(self):

        '''
        delete_contact method deletes a saved contact from the contact_list
        '''

        Contact.contact_list.remove(self)
```

Here we create a `delete_contact` method that uses the `remove()` method to delete the contact object from the `contact_list`.

Fifth Test

```
def test_find_contact_by_number(self):
        '''
        test to check if we can find a contact by phone number and display information
        '''

        self.new_contact.save_contact()
        test_contact = Contact("Test","user","0711223344","test@user.com") # new contact
        test_contact.save_contact()

        found_contact = Contact.find_by_number("0711223344")

        self.assertEqual(found_contact.email,test_contact.email)
```

We now create a test case that tests if we can find a contact object we use the method `find_by_number()` that takes on a phone number, and returns a contact object. Then we check if the contact object is equal to the saved contact.

Run the test to confirms if it fails. Then let us create the code to make the test pass

```
    @classmethod
    def find_by_number(cls,number):
        '''
        Method that takes in a number and returns a contact that matches that number.

        Args:
            number: Phone number to search for
        Returns :
            Contact of person that matches the number.
        '''

        for contact in cls.contact_list:
            if contact.phone_number == number:
                return contact
```

Decorators

In the above example we see a new line `@classmethod`. This is referred to as a **decorator**. Decorators allow you to make simple modifications to callable objects like functions, methods, or classes.

`@classmethod` simply informs the python interpreter that this is a method that belongs to the entire class

Just like the `self` argument refers to the object, `cls` referred to the entire class, and does not need to be passed in when we call the method.

In the `find_by_number` class method, we loop through each contact object in the `contact_list` and check if the phone number is equal to the number passed in. It then returns the contact object.