Testing your Code

Testing a Function

In [5]:

```
%%writefile name_function.py
def get_formatted_name(first, last):
    """Generate a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()
```

```
Overwriting name_function.py
```

In [12]:

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give a first name: ")
    if first == 'q':
        break
    last = input("\nPlease give a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print("\tNeatly formatted name: " + formatted_name + ".")
```

```
Enter 'q' at any time to quit.

Please give a first name: UGur

Please give a last name: Tigz
        Neatly formatted name: Ugur Tigz.

Please give a first name: q
```

- we created a function which just formats the first and last name neatly
- then we imported that function
- we used a while loop to get some input and store the input in the two varaibles *first* and *last*
- we also used the 'q' mechanism to quit the loop any time
- we use the get_formatted_name() function in combination with our input
- we print this one

In [28]:

```
import unittest
from name_function import get_formatted_name

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

    def test_first_last_name(self):
        """Do names like Janis Joplin' work?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')


unittest.main(argv=[''], exit=False)
#if __name__ == '__main__':
    #unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
.
----------------------------------------------------------------------
--
Ran 1 test in 0.002s

OK
```

Out[28]:

```
<unittest.main.TestProgram at 0x10c499320>
```

- **A passing test*
- we first import the unittest module
- also we import the get_formatted_name function from our name_function file
- we make a class which is inherited from unittest.TestCase
- this class will have just one method
- each method which starts with test_ will run automatically
- this method stores the get_formatted_name() function
  - we want to test this function
  - we pass it two arguments which we want to check
- we use the asserEqual() method, which comes from the unittest module
  - this takes two arguments
    - the variable in which the get_formatted_name (which we want to test) is stored
    - the correct form ('Janis Joplin') which has to be the correct form of our two arguments ('janis', 'joplin')
    - so if our function gives the same output as the assertEqual() method, the test is okay
- we call the unittest.main() - here we give it two arguemts (special for jupyter notebooks)
- the dot is telling that just a single test is passed

```
In [34]:
```

```
%%writefile full_name_function.py

def get_formatted_name(first, middle, last):
    """Generate a neatly formatted full name."""
    full_name = first + ' ' + middle + ' ' + last
    return full_name.title()
```

```
Writing full_name_function.py
```

```
In [35]:
```

```
import unittest
from full_name_function import get_formatted_name

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

    def test_first_last_name(self):
        """Do names like Janis Joplin' work?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')


unittest.main(argv=[''], exit=False)
```

```
E
======================================================================
==
ERROR: test_first_last_name (__main__.NamesTestCase)
Do names like Janis Joplin' work?
----------------------------------------------------------------------
--
Traceback (most recent call last):
  File "<ipython-input-35-1475357b8077>", line 9, in test_first_last
_name
    formatted_name = get_formatted_name('janis', 'joplin')
TypeError: get_formatted_name() missing 1 required positional argume
nt: 'last'

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

FAILED (errors=1)
```

```
Out[35]:
```

```
<unittest.main.TestProgram at 0x10c90e668>
```

- **A Failling Test**
- we create a new function, this time it contains a middle name
- we create a new test file
- this time we get an error
- the error says, that one argument is missing (the middle name is too much here)
- so the test simply fails with an error

In [45]:

```
%%writefile middle_optional_function.py

def get_formatted_name(first, last, middle=''):
    """Generate a neatly formatted full name."""
    if middle:
        full_name = first + ' ' + middle + ' ' + last
    else:
        full_name = first + ' ' + last
    return full_name.title()
```

```
Writing middle_optional_function.py
```

In [46]:

```
import unittest
from middle_optional_function import get_formatted_name

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

    def test_first_last_name(self):
        """Do names like Janis Joplin' work?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')


unittest.main(argv=[''], exit=False)
```

```
.
----------------------------------------------------------------------
--
Ran 1 test in 0.001s

OK
```

Out[46]:

```
<unittest.main.TestProgram at 0x10c902668>
```

- **Responding to a Failed Test**
- if a test fails don't change the test
- change the function
- in this case we change the functions arguments
- we make the middle name optional
- and if we have a middle name we have a full_name with the middle name
- else we have a full name without the middle name

In [48]:

```
import unittest
from middle_optional_function import get_formatted_name

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

    def test_first_last_name(self):
        """Do names like Janis Joplin' work?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')

    def test_first_last_middle_name(self):
        """Do names like 'Wolfgang Amadeus Mozart' work?"""
        formatted_name = get_formatted_name(
        'wolfgang', 'mozart', 'amadeus')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')


unittest.main(argv=[''], exit=False)
```

```
..
--------------------------------------------------------------------
--
Ran 2 tests in 0.003s

OK
```

Out[48]:

```
<unittest.main.TestProgram at 0x10abd4fd0>
```

- **Adding New Tests**
- we know, that our function has middle names as optional
- we make a new method in our test class
  - this method starts with *test_* so it will run automatically
  - this time we want to check if also names with middle names are okay
  - we build our get_formatted_name function with some arguments (note that the middle name arguement is the last one)
  - we give the needed arguments for or assertEqual() method (which variable, what should be the output)
- we call the unuttest.main() method

Testing a Class

Method	Use
asserEqual(a, b)	Verify that a == b
asserNotEqual(a, b)	Verify that a != b
assertTrue(x)	Verify that x is True
assertFalse(x)	Verify that x is False
assertIn(*item*, list)	Verify that *item* is in list
assertNotIn(*item*, list)	Verify that *item* is not in *list*

- This table shows the different assert Methods in Python
- you can use them just in classes which inherit from unittest.TestCase

In [1]:

```
%%writefile survey.py

class AnonymousSurvey():
    """Collect anonymous answers to a survey question."""

    def __init__(self, question):
        """Store a question, and prepare to store responses."""
        self.question = question
        self.responses = []

    def show_question(self):
        """Show the survey question."""
        print(self.question)

    def store_response(self, new_response):
        """Store a single response to the survey."""
        self.responses.append(new_response)

    def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in self.responses:
            print('- ' + response)
```

```
Overwriting survey.py
```

- **A class to test**
- first we build a class
- it gets just an argument 'question'
- also it has an empty list 'responses'
- it has 3 methods
    - first shows the question
    - the second one stores the new response in our empty list
    - the last one shows the lists element each
- we have this class in our survey.py module

In [2]:

```
from survey import AnonymousSurvey

# Define a question, and make a survey.
question = "What language did you first learn to speak?"
my_survey = AnonymousSurvey(question)

# Show the question, and store responses to the question.
my_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    my_survey.store_response(response)

# Show the survey results.
print("\nThank you to everyone who participated in the survey!")
my_survey.show_results()
```

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: German
Language: Turkish
Language: English
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- German
- Turkish
- English
```

- we import from our module 'survey' the class 'AnonymousSurvey'
- we define a question and store it
- we make an instance of our class qith the argument of our defined question
- we use the first method to simply show the question
- we print that we can quit the program, by pressing 'q'
- in our while loop we get all the responses with the input method and store them into response
- if the response is 'q' we quit
- we use the store function to store the response (we pass it)
- this will put all responses into the responses list with the append method
- we use the show results method to simply show whats in the list of responses we collected eachwise

In [3]:

```python
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')

        self.assertIn('English', my_survey.responses)

unittest.main(argv=[''], exit=False)
```

```
.
--------------------------------------------------------------------
--
Ran 1 test in 0.001s

OK
```

Out[3]:

```
<unittest.main.TestProgram at 0x1076ef278>
```

- **Testing the AnonymousSurvey Class**

- with the assertIn() method we can test, if an element is in a list
- we simply check whether 'English' is in the list responses
- to test a class, we have to make an instance of a class (which we do here) 'my_survey'
- we define the question before and pass it to our class instance
- we use the store_response() method, to store 'English' in our list
- we prove that, with the assertIn() method, we check if the element 'English' is in our list responses

In [4]:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')

        self.assertIn('English', my_survey.responses)

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        responses = ['German', 'Turkish', 'English']
        for response in responses:
            my_survey.store_response(response)

        for response in responses:
            self.assertIn(response, my_survey.responses)

unittest.main(argv=[''], exit=False)
```

```
..
----------------------------------------------------------------------
--
Ran 2 tests in 0.004s

OK
```

Out[4]:

```
<unittest.main.TestProgram at 0x107280828>
```

- this time we make another test
- we have a list of responses
- we use the store method to store each of them (3) in our list
- we make another for loop which tests, if each response is correctly in the list

In [5]:

```python
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""

    def setUp(self):
        """
        Create a survey and set of responses for use in all test methods.
        """
        question = "What language did you first learn to speak?"
        self.my_survey = AnonymousSurvey(question)
        self.responses = ['German', 'Turkish', 'English']

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

unittest.main(argv=[''], exit=False)
```

```
..
--------------------------------------------------------------------
--
Ran 2 tests in 0.003s

OK
```

Out[5]:

```
<unittest.main.TestProgram at 0x105b31e48>
```

- **The setUp() Method**
- we don't need to create an instance in each method we create
- the setUp() method is for initializing
- once we set up things here we can use it in every test_ method we create