Lab 8

Unit Testing In Python

Lab Objective: *One of the hardest parts of computer programming is ensuring that a program does what you expect it to do. For instance, a program may fail to meet specifications, or may work in some cases and not in others. Finding and fixing errors may be difficult and time consuming, especially if the program is very large or if many people are contributing the same program. Unit testing is a simple solution which helps to solve many of these difficulties in programming.*

In this lab, we explore unit testing in Python and apply it to the concept of Test Driven Design.

A *unit test* is a formal test that checks the smallest testable pieces of a program (often functions or classes) for correctness, independent of the rest of the code. Testing each unit of code verifies that each piece of code works as expected. When code does contain bugs, it is easier to identify which part of the code they came from. Writing unit tests also ensures that the specifications of a program are met.

Unit Testing is especially vital in corporate settings, where many developers often work on the same code or expand existing code. By keeping a collection of unit tests associated with a program, developers can ensure that when adding new features, they do not break any existing features of the program.

A well written collection of unit tests can ensure that all functions, and by extension an entire program, work correctly.

PyTest

A unit test verifies that a piece of code works by comparing actual output with expected output, given some specific input. Unit tests are often written for individual functions, but they can also test whole classes or files.

To test a function we use Python's reserved word `assert` and provide our function with input and expected output. Basic assert statements take the form

```
assert <truth statement>, "message"
```

which raises an `AssertionError` with error message `"message"` if `<truth statement>` is false, or else does nothing.

For example, we might test the following function

```
>>> def addition(a,b):
...      return a+b
```

with this test function:

```
>>> from specs import addition
>>> def test_addition():
...      assert addition(1,3) == 4, "Addition failed on positive integers"
...      assert addition(-5,-7) == -12, "Addition failed on negative integers"
...      assert addition(-6,14) == 8
```

When run, `test_addition()` displays no output since all the tests are valid. Note
that an error is optional.

Python provides an efficient way for running unit tests through a module called
PyTest. PyTest is a tool that allows you run many tests at the same time and get
more information about their results.

Try running the following from the terminal in your current directory:

```
$ py.test
```

Unless you've already written some tests, you probably got something like this:

```
=========================== test session starts ============================
platform win32 -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: C:\Users\Student\ACME, inifile:
collected 0 items


============================  in 0.13 seconds =============================
```

Even if you have already written some tests, pytest will not find them unless
their filenames have the form test_*.py or *_test.py, where the * represents any
number of characters. In addition, the test methods inside the files need to follow
suit and be named `test_*()` or `*_test()`. (If you need to change this for some rea-
son, you can consult the documentation at `http://pytest.org/latest/example/`
`pythoncollection.html`.)

For example, consider the following directory tree:

```
|-- api_tests
|    |-- test_accounts.py
|    |-- test_auth.py
|    |-- test_base.py
|    |-- test_common.py
|-- platform_tests
        |-- test_bulk.py
        |-- test_calendar.py
        |-- test_google.py
```

If you run `py.test` here, the following output is produced:

```
======================= test session starts =========================
platform win32 -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: C:\Users\Student\ACME\python_tests, inifile:
```

```
collected 29 items

api_tests/test_accounts.py ....
api_tests/test_auth.py .....
api_tests/test_base.py ....
api_tests/test_common.py ....
platform_tests/test_bulk.py ....
platform_tests/test_calendar.py ..
platform_tests/test_google.py ......
======================= 29 passed in 0.07 seconds =======================
```

Each dot represents a test passing. They show up in order, so, for example, if instead of the third dot there is an "F" (meaning test failed), you would know the third test in the respective file failed.

Coverage

More important than simply writing tests for code is writing quality tests. One way to measure the quality of your unit tests is by their coverage. Coverage measures how many lines of code are executed by at least one test case. When considering coverage, it is useful to think of a function in the form of a diagram, containing branches and loops. Each test takes a certain path through the function, traversing some branches and not others. Full coverage is reached when each branch of the function is traversed by at least one test case.

An easy way to measure coverage in Python is to use a tool called `pytest-cov`. While the pytest module comes bundled with Anaconda, pytest-cov does not. However, you should be able to install it from the terminal by running the following command:

```
$ conda install pytest-cov
```

Adding the flag `--cov` to py.test prints out a breakdown of test coverage on your code. For example, running the incomplete spec files for this lab yields the following output:

```
  $ py.test --cov
======================= test session starts =======================
platform win32 -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: C:\Users\Student\ACME, inifile:
plugins: cov-2.4.0
collected 8 items

test_specs.py ........

---------- coverage: platform win32, python 2.7.11-final-0 -----------
Name              Stmts   Miss  Cover
------------------------------------
specs.py             65     38    42%
test_specs.py        31      0   100%
------------------------------------
TOTAL                96     38    60%


=================== 8 passed in 0.03 seconds ====================
```

In the above output, `Stmts` refers to the number of lines in the code that pytest ran, while `Miss` is the number of lines of code that were not run. Notice the file `test_specs.py` has 100% coverage while `specs.py` does not. Test files will generally have 100% coverage, since pytest is designed to run these files in their entirety. However, the spec file does not have full coverage and requires additional test cases to be thoroughly tested.

> **Problem 1.** Install pytest-cov (`conda install pytest-cov`). Run `py.test --cov` in the directory which has the two files **specs.py** and **test_specs.py**. You should see an output similar to the one above.
>
> Now write some of your own tests for the addition function and the smallest factor function. Be sure to give each function its own test. (This is more of a convention than a mandate, but it's good practice and will allow you to track down bugs quickly in the future.) Be sure that your tests cover every statement of your code in those two functions. (The number of statements missed should go from 33 to 26)

Pytest as a Module

When developing software, we often want our code to raise an exception when certain input is given, instead of proceeding. Furthermore, we want the exception to give us relevant information about what went wrong and why it was raised. For example, if a user feeds a bad value to a calculator (like dividing by 0), we would like it if the calculator were very explicit as to why it wasn't able to process our input. Just as you should test your code to make sure valid cases give the correct output, like you did in the Problem 1, you should also test cases that generate an exception. Standard assert functions, however, simply will not do, because they can't catch an exception.

Instead, we can use the `raises` method from the `pytest` module, which acts like an assert statement for raising and catching exceptions. Consider the following example:

```python
>>> def divide(a,b):
...     if b==0:
...         raise ValueError("You can't give a zero for b, that breaks this ←
    function")
...     else:
...         return float(a)/float(b)
```

The above function could be tested by this test function:

```python
>>> import pytest
>>> def test_divide():
...     assert divide(1,2) == .5
...     assert divide(5,4) == 1.25
...     pytest.raises(ValueError, divide, a=4, b=0)
```

The above code tests whether or not a `ValueError` is raised when `divide` is called with the arguments `a=4` and `b=0`. However, it does not allow us to test whether the message that came with the error is what we want it to be. To do that, we can combine `raises` with the `with` statement, like so:

```
...        with pytest.raises(Exception) as excinfo:
...            divide(4,0)
...        assert excinfo.typename == 'ValueError'
...        assert excinfo.value.args[0] == "You can't give a zero for b, that breaks↩
       this function"
```

In this example, `excinfo` is an object of type `py.code.ExceptionInfo()` which contains info about the exception that was raised. Once we have raised the expection within the `with` block, we can assert whatever we want about the different attributes of `execinfo` to test whether we raised the right exception.

> **Problem 2.** Write tests for the `operator` function. Be sure to handle every statement and check against every exception. For exceptions, be sure to check both the typename and the message, that way you can say with certainty that the test is adequately testing your code. (The number of statements missed should go from 26 to 11)

Suppose you want to use the same pieces of code in different test functions. Pytest has a feature called *fixtures* for mocking data that reduces code duplication. Mocking data is useful when you don't want certain tests to be dependent on the working of completely separate code. For example, if we had a feature which was dependent upon a user service working correctly, rather than having my tests make a call to the user service I may write a fixture which simply gives existing mock data that I expect for my feature. As far as preventing code duplication, a fixture may also be utilized to create a class object which we can then call within our function.

```python
class ComplexNumber(object):
... def __init__(self, real=0, imag=0):
...     self.real = real
...     self.imag = imag
... def conjugate(self):
...     conjugate = ComplexNumber(real=self.real, imag=-self.imag)
...     return conjugate
... def norm(self):
...     magnitude = math.sqrt(self.real**2 + self.imag**2)
...     return magnitude
... def __add__(self, other):
...     real = self.real + other.real
...     imag = self.imag + other.imag
...     return ComplexNumber(real=real, imag=imag)
... def __sub__(self, other):
...     real = self.real - other.real
...     imag = self.imag - other.imag
...     return ComplexNumber(real=real, imag=imag)
```

```
... def __mul__(self, other):
...     real = self.real*other.real - self.imag*other.imag
...     imag = self.imag*other.real + other.imag*self.real
...     return ComplexNumber(real=real, imag=imag)
... def __div__(self, other):
...     if other.real==0 and other.imag==0:
...         raise ValueError("Cannot divide by zero")
...     bottom = (other.conjugate()*other*1.).real
...     top = self*other.conjugate()
...     return ComplexNumber(real=(top.real/bottom), imag=(top.imag/bottom))
... def __eq__(self, other):
...     return self.imag == other.imag and self.real == other.real
... def __str__(self):
...     return str(self.real)+('+' if self.imag>=0 else '')+str(self.imag)+'i'
```

Given the previous class, you could reduce code setup for your tests by using a test fixture like so:

```
>>> from specs import ComplexNumber
>>> import pytest
>>> @pytest.fixture
... def set_up_complex_nums():
...     number_1 = ComplexNumber(1, 2)
...     number_2 = ComplexNumber(5, 5)
...     number_3 = ComplexNumber(2, 9)
...     return number_1, number_2, number_3
>>> def test_complex_addition(set_up_complex_nums):
...     number_1, number_2, number_3 = set_up_complex_nums
...     assert number_1 + number_2 == ComplexNumber(6, 7)
...     assert number_1 + number_3 == ComplexNumber(3, 11)
...     assert number_2 + number_3 == ComplexNumber(7, 14)
...     assert number_3 + number_3 == ComplexNumber(4, 18)
>>> def test_complex_multiplication(set_up_complex_nums):
...     number_1, number_2, number_3 = set_up_complex_nums
...     assert number_1 * number_2 == ComplexNumber(-5, 15)
...     assert number_1 * number_3 == ComplexNumber(-17, 11)
...     assert number_2 * number_3 == ComplexNumber(-40, 50)
...     assert number_3 * number_3 == ComplexNumber(-80, 18)
```

Problem 3. Finish writing unit test for the complex numbers class. Be sure to utilize fixtures in order to reduce on the length of your code. Also, remember, it would be useful (and good practice) to write a different test function for each method in the ComplexNumberClass. (The number of statements missed should go from 11 to 0)

Test Driven Development

Test Driven Development (TDD) is a programming style in which the unit tests are written first and the actual code is written after. Kent Beck, the creator of extreme programming, claims to have re-discovered Test Driven Development. He said, "The original description of TDD was in an ancient book about programming.

It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output."

TDD initially sounds extreme at first, but it actually incentivizes simple design, elegant code, and gives quantifiable checkpoints in the development process.

The idea is simple enough:

```
┌──────┐
│ Idea │
└──────┘
    │
    ▼
  ╭───────╮
  │ Tests │
  ╰───────╯
    │
    ▼
┌────────────────┐
│ Implementation │
└────────────────┘
```
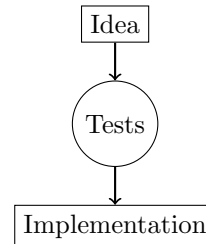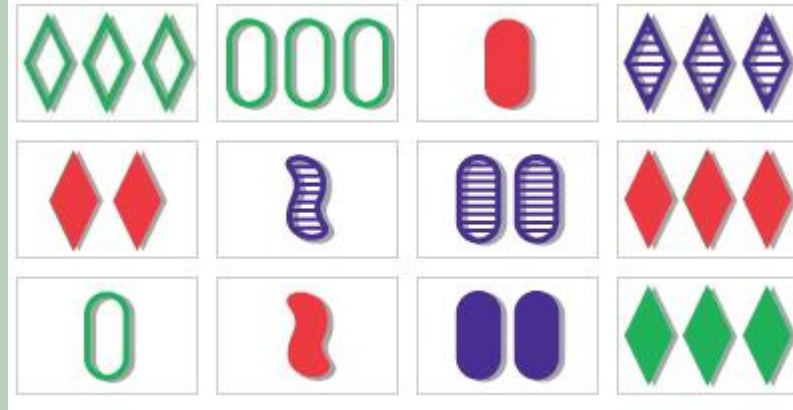
First, the developer writes a test case for each feature of the program. Since the program is as yet unwritten, all of the test cases will initially fail. Next the developer implements each feature until the test succeeds. If the test cases are sufficiently thorough, when the tests all pass, the developer knows the program is complete.

TDD eliminates the chore of testing code after it has been written. It also eliminates the fear that even though code is complete, it may contain obscure bugs. Furthermore, writing unit tests helps the developer understand the project as a whole as well as the minute details of each feature.

The idea must be a bit more concrete than "I want to add a chat room to my website." The general idea of how the chat room will look in code (like a history module, a data base, and a sketch of required functions, etc.) is required. Next that idea is transformed into tests because the inputs and outputs of required functionality are understood. Following the example of the chat room, you could write a unit test that checks what a given API returns when a user has broadcasted some message. The implementation is simply adding, changing, and editing code until all the required tests pass.

Problem 4. The game Set is a card game about finding patterns. Each card contains a design with 4 different properties: color (red, green or purple), shape (diamond, oval or sqiggly), quantity (one, two or three) and pattern (solid, striped or outlined). Twelve cards are laid on a table and the object is to recognize sets of cards. A *set* is made up of three cards which are either all the same or all different for each property. You can try playing Set online **here**.

Here is a group of 12 cards from Set.

This collection of cards contains 6 unique sets:

(a) Same in quantity and shape;
different in pattern and color

(b) Same in color and pattern;
different in shape and quantity

(c) Same in pattern; different in
shape, quantity and color

(d) Same in shape; different in
quantity, pattern and color

(e) Different in all aspects
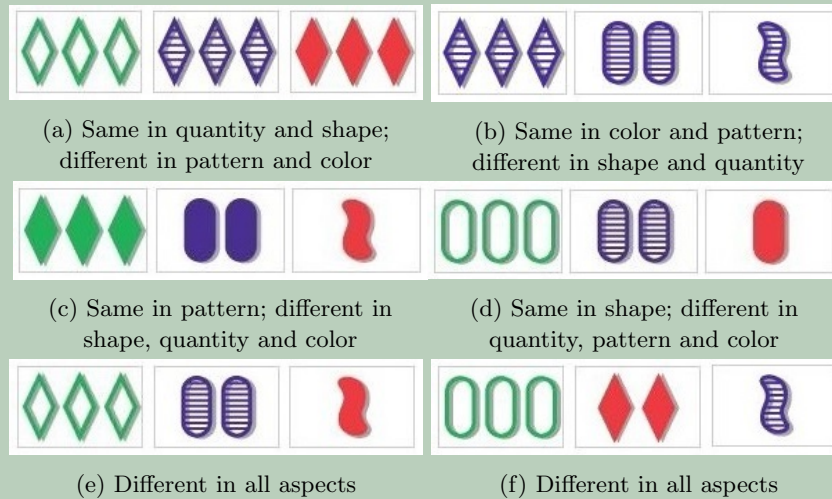
(f) Different in all aspects

Figure 8.1: Five unique sets from the above group of cards

Use the principles of Test Driven Development to write a function that takes as input a file name of a file containing 12 Set cards and returns the number of sets they contain. In this problem, **only** write test cases for the function, **not** the function itself. Focus on *what* you want to implement rather *how* you are going to implement it. Use these specifications:

- Each card in Set can be uniquely represented by a 4-bit integer in base 3, where each digit represents a different property and each property has 3 possible values. Represent a group of Set cards as a text file where each line contains a 4-bit number representing one card.[a]

- Have your program throw an error if the input is invalid (anything that is not a valid Set card as described above), if a duplicate card is entered, if less or more than 12 cards are given, or if the file name given is not valid.

- Ensure that each possible error gives an appropriate error message.

- Aggregate all your text files in a single folder called "hands" and upload the folder to your git repository along with your code.

Consider breaking this problem down into a few smaller functions and writing a unit test for each function.

---

[a]A 4-bit integer in base 3 contains four digits that are either 0, 1 or 2. For example, 0000 and 1201 are 4-bit integers in base 3, whereas 000 is not because it has only three digits, and 0123 is not because it contains the number 3.

Problem 5. Now that you have written test cases for a set game pattern recognizer, write code until all your test cases pass. Write this code at the end of `specs.py`. Rather than running this file to see if your code is working, simply run the test file to see if all the test cases pass. Here are some suggestions for writing the implementation:

Hint: Given three integers which represent cards, the cards form a set if their columns each sum to a multiple of 3.

Additional Material

Python Debugger

Python has a built in debugger called pdb (Python DeBugger) to aid in finding mistakes in code. Pdb runs in the terminal or in a Jupyter Notebook.

A *break point* is a spot where the program will pause while running. Setting break points allows you to view the state of your program while it is running.

To begin using pdb, import it in your program:

```
>>> import pdb
```

then add

```
... pdb.set_trace()
```

where you would like to set a breakpoint. Then when you run your code, instead of running in its entirety, it will run until it reaches the break point and then the pbd prompt will be displayed, waiting for a command:

```
(Pdb)
```

At this point there are several commands you can run, depending on what you want to do.

Command	Description
n	"Next": executes the next line
p <var>	"Print": displays the value of the specified variable
c	"Continue": Stops debugging and runs the program normally to the end
q	"Quit": terminates the program
l	"list": shows several lines of code around the current line
r	"Return": returns to the end of a subroutine
<Enter>	Executes the most recent command again

> **NOTE**
>
> Pdb provides the freedom to change the values of your variables while debugging. Suppose you are debugging the following code:
>
> ```python
> def func():
>     pdb.set_trace()
>     foo = 1
>     bar = 2
>     return foo + bar + 42
> ```
>
> While debugging, you realize that the value of `foo` was initialized wrong. Then after `foo` has been initialized you can enter the command
>
> ```
> (Pdb) foo = 5
> ```
>
> and the value of the variable will be changed as the program continues to run and the program will return 49 instead of 45.