An introduction to Test Driven Development (TDD)

credit: The Hitchiker's Guide to Python

Python 3 Documentation: unitttest

RealPythonBlog: Getting Started with Testing in Python

How to use TDD in a Data Science Workflow

If and when you should use TDD

Objectives

- Explain the Test Driven Development (TDD) paradigm, and contrast it with what you're doing now
- List the steps of the TDD cycle
- Use unittest to test a unit of code in Python
- Use unittest to perform a suite of tests in Python
- Discuss when TDD is appropriate in the work of a Data Scientist

Test Driven Development

TDD ... relies on the repetition of a short software development cycle:

- 1. requirements are turned into specific test cases, then ...
- 2. software is improved to pass the new tests, only.

This is in contrast to allowing code to be added that is not proven to meet requirements. (and were requirements ever made?)

--Wikipedia

Why?

- It results in cleaner, more maintainable, easier to understand code with fewer bugs.
- It helps protect working code from later code contributions.
- Science! Thesis: <u>Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects</u>

TDD steps

Based on software functionality that has been broken down into specific requirements, for each requirement:

- add a test (we'll be using unittest in Python to do this)
- run the test make sure the code fails the test (red)
- write code for the desired requirement
- run the test make sure the code passes the test (green)
- refactor and repeat

red -> green -> refactor

When the (new) test is run, that often refers to an entire test suite of previously written (and passed) tests for your codebase in the project.

Testing code in Python

<u>unittest</u> is the test module in the Python standard library

There are other ways to test:

```
doctest
pytest
nose
tox
```

Many of these are attempting to "standardize" and "make testing easier" in Python. DSI instructors don't agree on the best one.

Will just focus on unittest for the remainder of this lecture.

unittest - preliminary

- The basic building blocks of unit testing are test cases single scenarios that must be set up and checked for correctness.
- The testing code of a TestCase instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.
- The simplest TestCase subclass will simply implement a test method (i.e. a method whose name starts with **test**).
- In order to test something, assert*() methods are provided by the TestCase base class. If the test fails, an exception will be raised with an explanatory message, and unittest will identify the test case as a failure.

Requirement: write a function that tests if a series of integers in a list is a <u>Fibonacci</u> sequence

Requirement: write a function that tests if a series of integers in a list is a <u>Fibonacci</u> sequence

Step 1: Write the tests:

```
test fibonacci.py:
 1 # requirement:
 2 # test if a sequence of at least three integers
 3 # in a list is a Fibonacci sequence
 5 # code is in fibonacci.py
 6 import fibonacci
 7 import unittest
 9
10 class TestFibonacci(unittest.TestCase):
11
       def test_is_fibonacci(self):
           self.assertEqual(fibonacci.is_fibonacci([1,1,2,3]), True)
12
           self.assertEqual(fibonacci.is_fibonacci([1,1,2,4]), False)
13
           self.assertEqual(fibonacci.is_fibonacci([0,1,1]), True)
14
           self.assertEqual(fibonacci.is_fibonacci([34,55,89,144,233]), True)
15
16
17 if __name__ == '__main__':
18
       unittest.main()
19
```

Requirement: write a function that tests if a series of integers in a list is a <u>Fibonacci</u> sequence

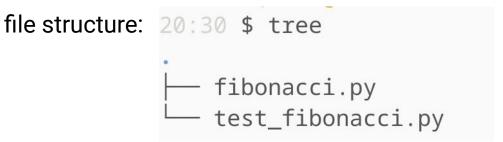
Step 2: Write the code (and make sure it will fail)

```
fibonacci.py:

1 def is_fibonacci(lst_ints):
2    '''Test if the integers in the list are a
3     Fibonacci sequence.
4    '''
5    pass
6
7 if __name__ == '__main__':
8    lst = [0, 1, 1]
9    check = is_fibonacci(lst)
10
```

Requirement: write a function that tests if a series of integers in a list is a <u>Fibonacci</u> sequence

Step 3: Test it (and verify that it fails)



running the test:

Requirement: write a function that tests if a series of integers in a list is a <u>Fibonacci</u> sequence

Step 4: Modify the code to pass the test.

```
fibonacci.py:
 1 import numpy as np
 3 def is_fibonacci(lst_ints):
       '''Test if the integers in the list are a
          Fibonacci sequence.
       1 1 1
      fib = True
      for i in range(len(lst_ints)-2):
           val = lst ints[i] + lst ints[i+1]
          if val != lst_ints[i+2]:
10
11
               fib = False
12
               break
13
      return fib
14
15 if __name__ == '__main__':
   lst = [0, 1, 1]
16
       check = is_fibonacci(lst)
```

Requirement: write a function that tests if a series of integers in a list is a <u>Fibonacci</u> sequence

Step 5: Test it (and verify that it passes)

```
file structure: 20:30 $ tree

fibonacci.py
test_fibonacci.py
```

running the test:

Requirement: write a function that tests if a series of integers in a list is a

Fibonacci sequence

Step 6: Refactor (and verify that it still passes)

```
fibonacci.py:
 1 import numpy as np
 3 def is_fibonacci(lst_ints):
       '''Test if the integers in the list are a
          Fibonacci sequence.
 6
       a = lst_ints[:-2]
       b = 1st ints[1:-1]
       c = lst_ints[2:]
       sums = [i1 + i2 \text{ for } i1, i2 \text{ in } zip(a,b)]
10
       return np.all(c == sums)
11
```

Reference: assertions in unittest

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
<pre>assertIsNotNone(x)</pre>	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	<pre>isinstance(a, b)</pre>
<pre>assertNotIsInstance(a, b)</pre>	not isinstance(a, b)

Method	Checks that
<pre>assertAlmostEqual(a, b)</pre>	round(a-b, 7) == 0
<pre>assertNotAlmostEqual(a, b)</pre>	round(a-b, 7) != 0
assertGreater(a, b)	a > b
<pre>assertGreaterEqual(a, b)</pre>	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	<pre>not r.search(s)</pre>
assertCountEqual(a, b)	a and b have the same elements in the same numbe regardless of their order.

And there are more - see the documentation

Reference: getting help on terminal syntax

```
$ python -m unittest -h
```

unittest - more advanced

- Tests can be numerous, and their set-up can be repetitive. Set-up code can be implementing using a method called setUp(), which the testing framework will automatically call for every single test we run.
 - Good for classes.
- It is recommended that you use <code>TestCase</code> implementations to group tests together according to the features they test. <code>unittest</code> provides a mechanism for this: the test suite, represented by the <code>TestSuite</code> class.
 - In most cases, calling unittest.main() will do the right thing and collect all the module's test cases for you and execute them.
- However, should you want to customize the building of your test suite, you can do it yourself.

TDD & test suite example

Goal: Develop a game in python where two users each are given an n-sided die and they roll them. Whoever rolls the higher number wins.

TDD & test suite example

Goal: Develop a game in python where two players are each given an n-sided die and they roll them. Whoever rolls the higher number wins.

Requirements:

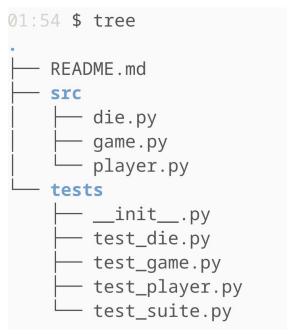
```
+ make a Die class
attributes: n-sides
face-up value
method:
roll
```

- + make a Player class attribute: die
- make a game script instantiates players and dice and plays the game

TDD & test suite example

Goal: Develop a game in python where two players are each given an n-sided die and they roll them. Whoever rolls the higher number wins.

directory structure:



running one test:

```
01:59 $ python -m unittest tests/test_die.py -v
test_n_sides_default (tests.test_die.TestDie) ... ok
test_roll (tests.test_die.TestDie) ... ok
Ran 2 tests in 0.000s
```

running all of them:

```
02:01 $ python -m unittest discover -v
```

TDD for Data Scientists

- TDD is useful for making robust software. If your code is going into a software development project you will likely be pushed to use it.
- TDD affects your thinking about coding the problem (probably in a good way!)
- But TDD may not be worth the effort:
 - \circ EDA
 - Proof of concept
 - You are the sole contributor (but what if you come back to code after 3 months and want to change it!?! tests would be useful).

Objectives

- Explain the Test Driven Development (TDD) paradigm, and contrast it with what you are likely doing now
- List the steps of the TDD cycle
- Use unittest to test code a unit of code in Python
- Use unittest to perform a suite of tests in Python
- Discuss when TDD is appropriate in the work of a Data Scientist