A Wildly Brief 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
Frank Burkholder

"Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object- oriented or well encapsulated it is."

- Michael Feathers

"Tests are stories we tell the next generation of programmers on a project."

- Roy Osherove

"TDD isn't something that comes naturally. It's a discipline, like a martial art, and just like in a Kung Fu movie, you need a bad-tempered and unreasonable master to force you to learn the discipline."

- Harry Percival

"Once you've worked on a system with extensive automated tests, I don't think you'll want to go back to working without them. You get this incredible sense of freedom to change the code, refactor, and keep making frequent releases to end users."

- Emily Bache

Objectives

galvanize

- Explain the Test Driven Development (TDD) paradigm
    - 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

galvanize

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

Test Driven Development

galvanize

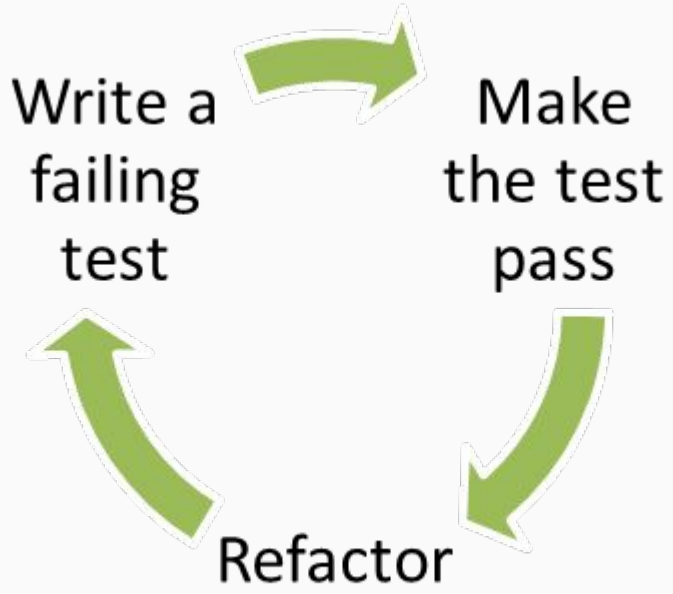
Why?

- Code is cleaner
- Code is more maintainable
- Code has fewer bugs
- Working code is protected
- Code is more easily trusted
- Science! Thesis: Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects
  - Showed that TDD leads to software that is less complex, less error-prone (buggy), and more tightly-coupled

TDD Cycle

galvanize

1. Establish acceptance criteria
2. **Write failing test(s)**
3. Write code-under-test
4. **Observe passing tests**
5. Read acceptance criteria
6. **Refactor (improvement)**
7. Repeat



TDD steps

galvanize

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

galvanize

`unittest` is the test module in the Python standard library

There are *many* other tools for testing, including:
`Doctest`, `pytest`, `nose2`, `tox`, `robot`, `testify`, `hypothesis`

Many of these are attempting to “standardize” and “make testing easier” in Python. In my experience, if you have at least three data scientists, you won’t have agreement on which is the best one.

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

`unittest` – preliminary

galvanize

- 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*.

`unittest` Example

galvanize

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

Requirement:

Write a function that tests if a series of integers in a list is a Fibonacci sequence

unittest Example

galvanize

Requirement: write a function that tests if a series of integers in a list is a Fibonacci 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
4
5 # code is in fibonacci.py
6 import fibonacci
7 import unittest
8
9
10 class TestFibonacci(unittest.TestCase):
11     def test_is_fibonacci(self):
12         self.assertEqual(fibonacci.is_fibonacci([1,1,2,3]), True)
13         self.assertEqual(fibonacci.is_fibonacci([1,1,2,4]), False)
14         self.assertEqual(fibonacci.is_fibonacci([0,1,1]), True)
15         self.assertEqual(fibonacci.is_fibonacci([34,55,89,144,233]), True)
16
17 if __name__ == '__main__':
18     unittest.main()
```

unittest Example

galvanize

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

`unittest` Example

galvanize

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

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

file structure:

```
20:30 $ tree
.
├── fibonacci.py
└── test_fibonacci.py
```

running the test:

```
20:54 $ python -m unittest test_fibonacci.py
F
======================================================================
FAIL: test_is_fibonacci (test_fibonacci.TestFibonacci)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/frankburkholder/g/lectures_dsi/test-driven-development/frank/test_fibonacci.py", line 12, in test_is_fibonacci
    self.assertEqual(fibonacci.is_fibonacci([1,1,2,3]), True)
AssertionError: None != True

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

`unittest` Example

galvanize

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

Step 4: Modify the code to pass the test.

fibonacci.py:

```
 1 import numpy as np
 2
 3 def is_fibonacci(lst_ints):
 4     '''Test if the integers in the list are a
 5        Fibonacci sequence.
 6     '''
 7     fib = True
 8     for i in range(len(lst_ints)-2):
 9         val = lst_ints[i] + lst_ints[i+1]
10         if val != lst_ints[i+2]:
11             fib = False
12             break
13     return fib
14
15 if __name__ == '__main__':
16     lst = [0, 1, 1]
17     check = is_fibonacci(lst)
```

`unittest` Example

galvanize

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

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

file structure:

```
20:30 $ tree
.
├── fibonacci.py
└── test_fibonacci.py
```

running the test:

```
20:55 $ python -m unittest test_fibonacci.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

unittest Example

galvanize

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:

```python
import numpy as np

def is_fibonacci(lst_ints):
    '''Test if the integers in the list are a
       Fibonacci sequence.
    '''
    a = lst_ints[:-2]
    b = lst_ints[1:-1]
    c = lst_ints[2:]
    sums = [i1 + i2 for i1, i2 in zip(a,b)]
    return np.all(c == sums)
```

```
21:17 $ python -m unittest test_fibonacci.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Reference: assertions in `unittest`

galvanize

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`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Method	Checks that
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`
`assertGreater(a, b)`	`a > b`
`assertGreaterEqual(a, b)`	`a >= b`
`assertLess(a, b)`	`a < b`
`assertLessEqual(a, b)`	`a <= b`
`assertRegex(s, r)`	`r.search(s)`
`assertNotRegex(s, r)`	`not r.search(s)`
`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

galvanize

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

`unittest` – more advanced

galvanize

- 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 `TestCase` implementations to group tests together according to the features they test. `unittest` provides a mechanism for this: the *test suite*, represented by the `TestSuite` 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

galvanize

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

galvanize

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

galvanize

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:

```
01:54 $ tree
.
├── README.md
├── src
│   ├── die.py
│   ├── game.py
│   └── player.py
└── tests
    ├── __init__.py
    ├── test_die.py
    ├── test_game.py
    ├── test_player.py
    └── test_suite.py
```

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

galvanize

- 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:
  - 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

galvanize

- 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