Test Driven Development

...if it's worth developing; it's worth testing

Instructor:
John J Rofrano
Senior Technical Staff Member, DevOps Champion
IBM T.J. Watson Research Center
rofrano@us.ibm.com
@JohnRofrano

What Will You Learn?

- Why automated testing is important to DevOps
- How Test Driven Development makes you think about the requirements first
- How Test Driven Development can improve your code and save you time
- How to use Code Coverage to ensure all paths are tested

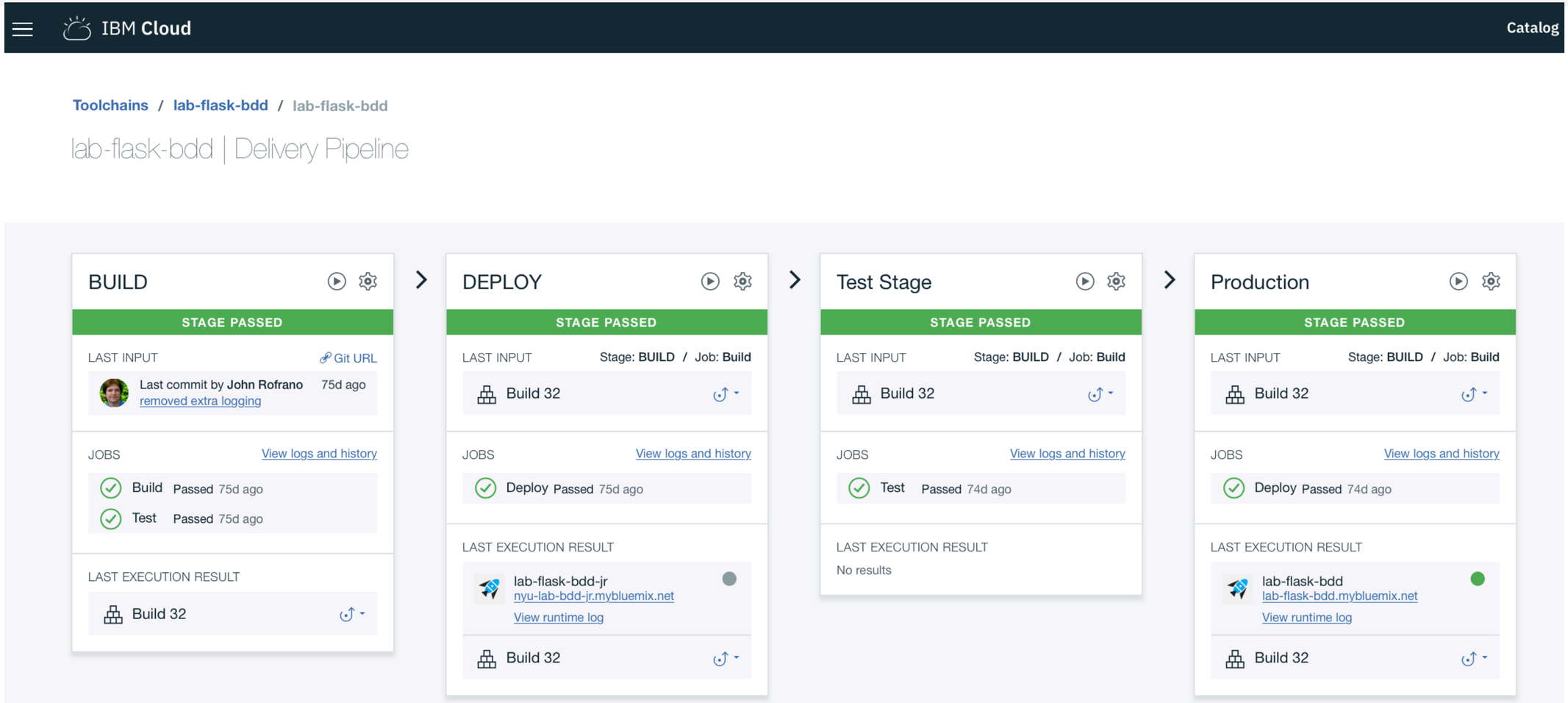
“Hi Prof. Rofrano, I wanted to thank you for driving home the benefits of using TDD. Having worked in startups, we would always say that we don't have time to write tests only to see bugs post release + waste hours debugging away on postman. Working on my capstone project, I had to create a set of API routes to be consumed by our mobile app and I wrote tests for every route.

It saved me a lot of time by quickly identifying errors in my SQL queries and made me more confident of the work I did.

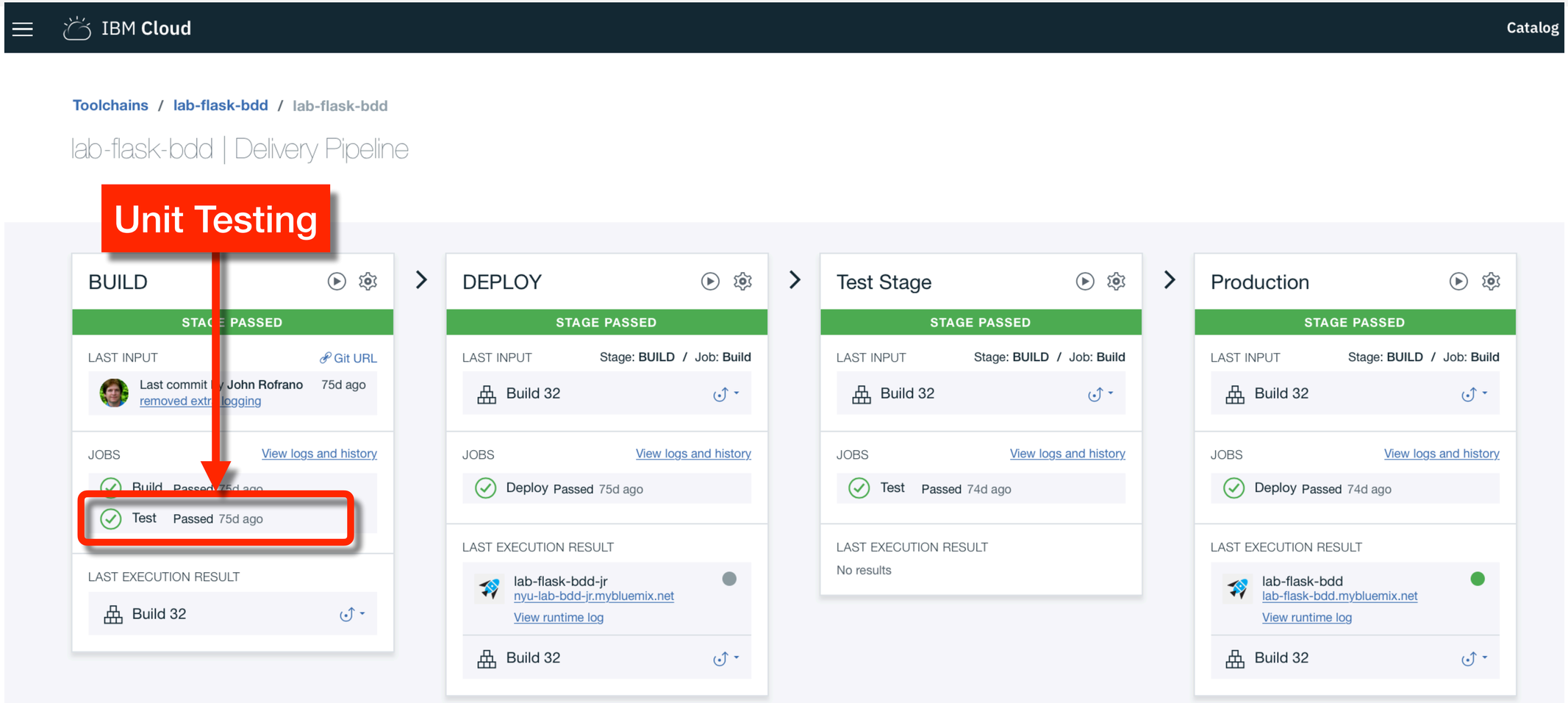
Thanks for teaching us about it!”

–NYU DevOps Student, Fall 2019

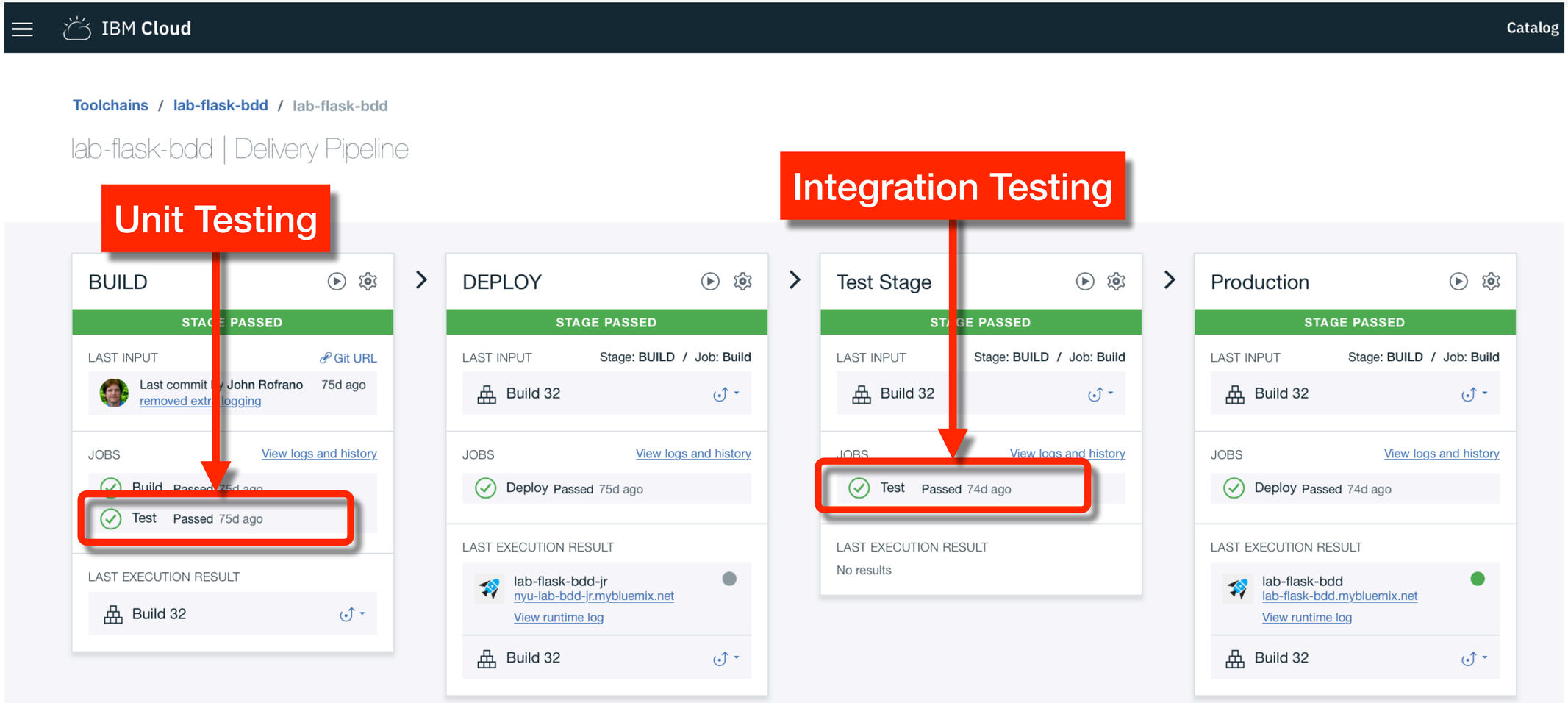
What is the Goal?



What is the Goal?



What is the Goal?



“If it's worth building, it's worth testing.
If it's not worth testing, why are you wasting your time working on it?”

–*agiledata.org*

Why Developers Don’t Test

Why Developers Don’t Test

- I already know it works!
  - Others who work on your code in the future won’t know if they broke something

Why Developers Don’t Test

- I already know it works!
  - Others who work on your code in the future won’t know if they broke something
- I don’t write broken code!
  - Sometimes the environment changes and other future libraries read your code

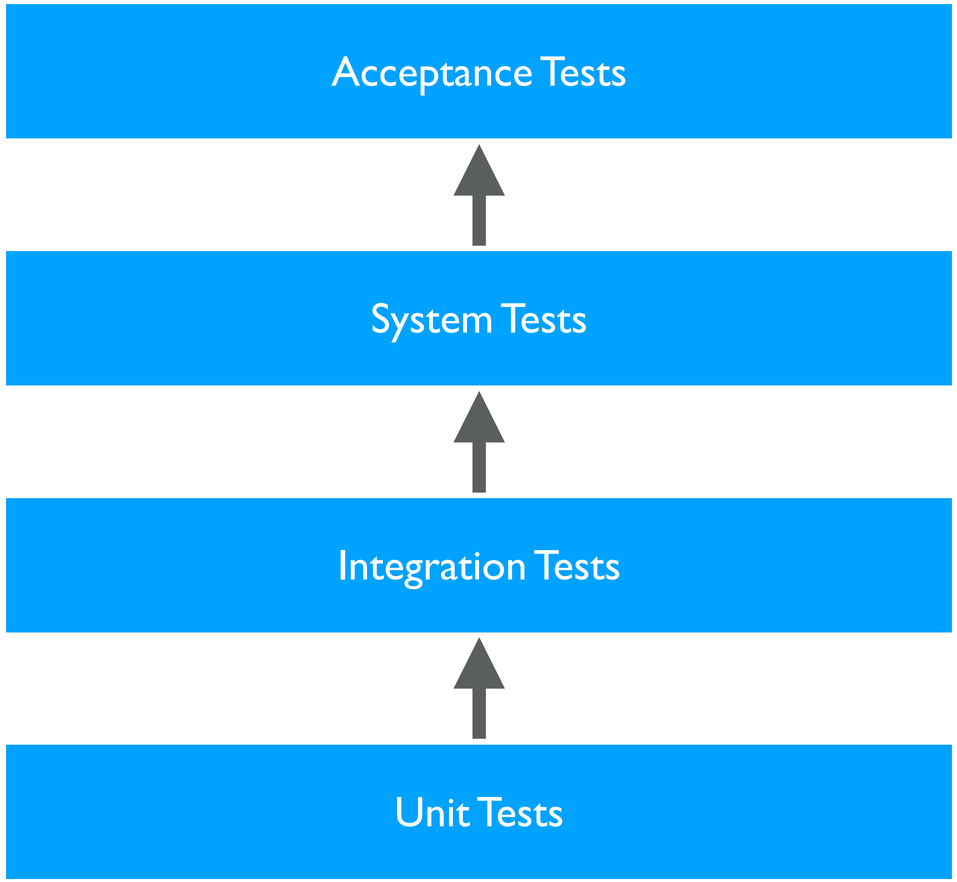
Why Developers Don’t Test

- I already know it works!
  - Others who work on your code in the future won’t know if they broke something
- I don’t write broken code!
  - Sometimes the environment changes and other future libraries read your code
- I have no time!
  - Testing actually saves you time (and stress) in the long run

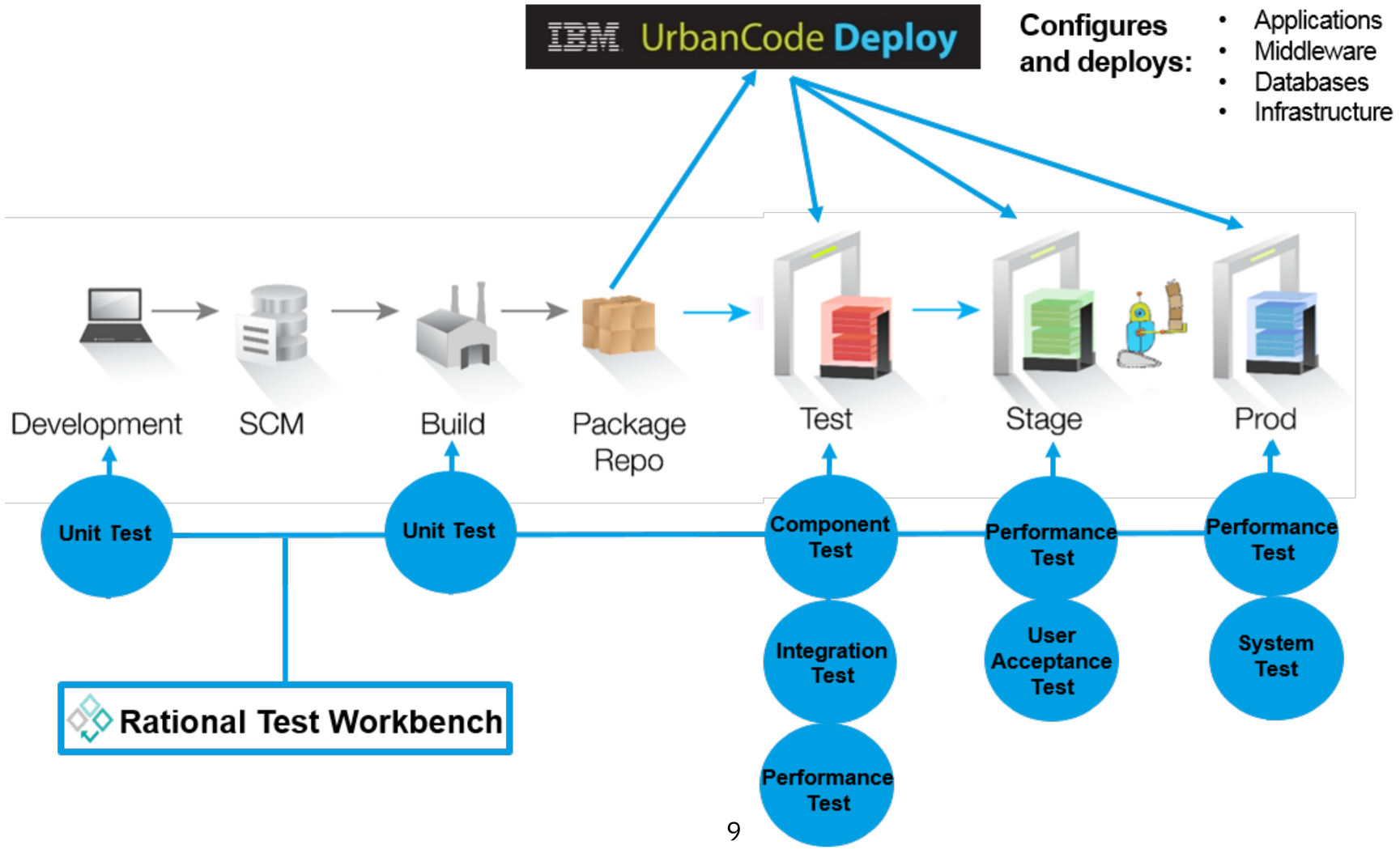
Why Do We Need To Test?

Software Testing Levels

Acceptance Testing	A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.
System Testing	A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
Integration Testing	A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
Unit Testing	A level of the software testing process where individual units/ components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.

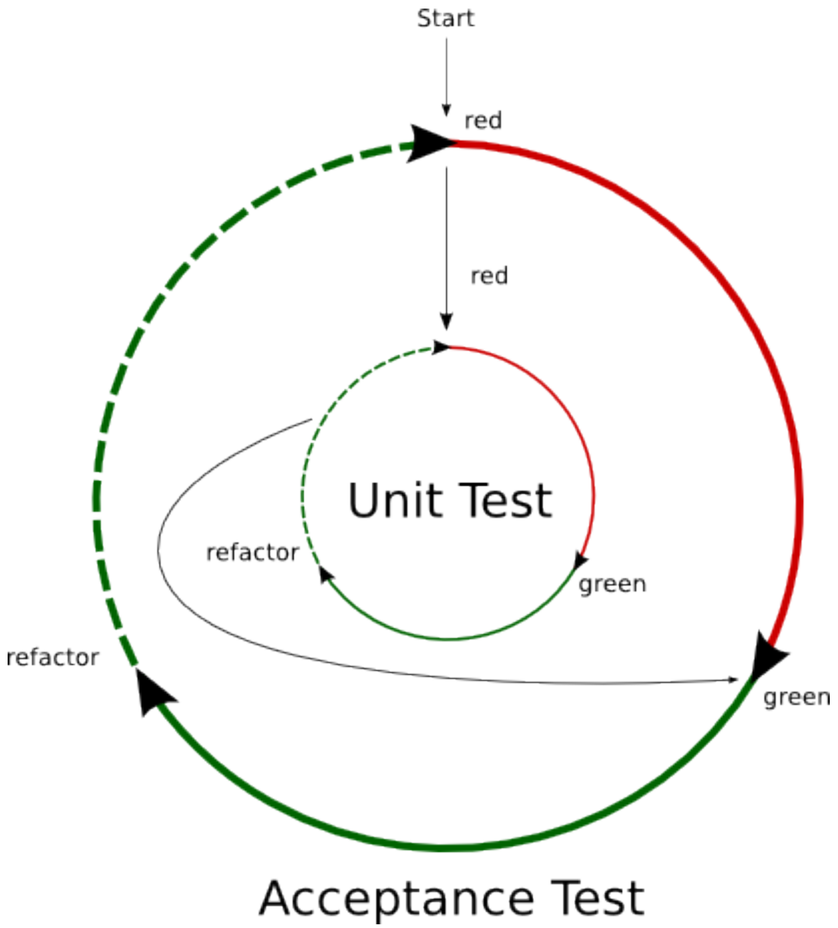


Traditional Release Cycle



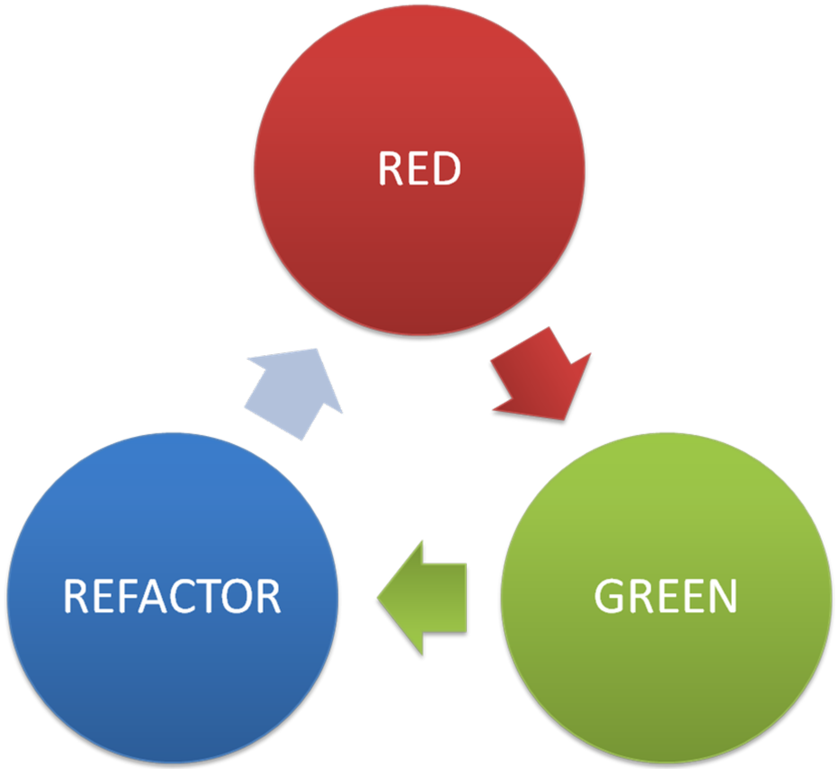
What is BDD & TDD

- Behavior-Driven Development (BDD)
  - Describes the behavior of the system from the outside in
  - Used for Integration / Acceptance Testing
- Test Driven Development (TDD)
  - Tests the functions of the system from the inside out
  - Used for unit testing

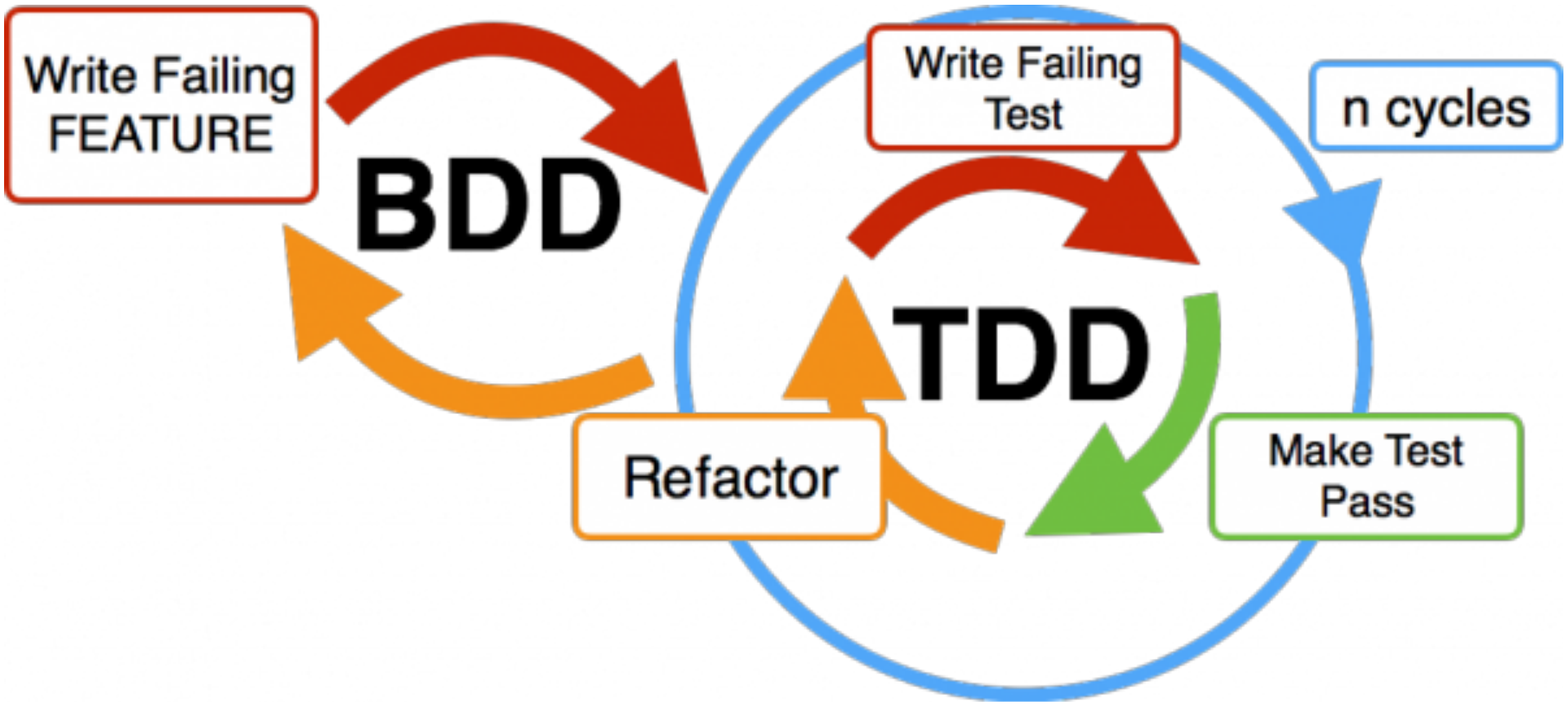


TDD Workflow

- Write a test case and watch it **FAIL**
- Write the code to make it **PASS**
- **REFACTOR** the code to make it great knowing that the test case will let you know if you broke anything



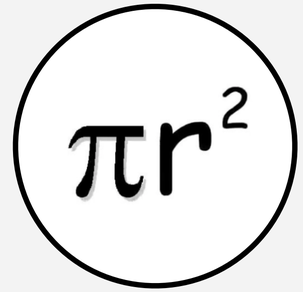
BDD & TDD Cycle



“BDD is building the *right thing*, and TDD is building the *thing right*.”

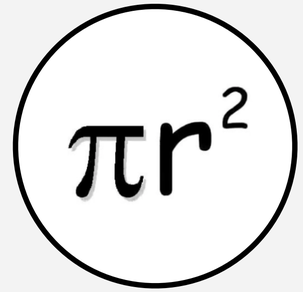
Testing Case Study

Calculate the Area of a Circle



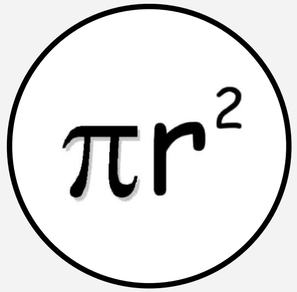
Socratica -https://www.youtube.com/watch?v=1Lfv5tUGsn8

Calculate the Area of a Circle



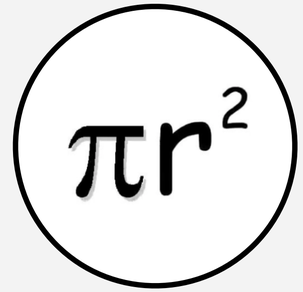

```
from math import pi
```

Calculate the Area of a Circle



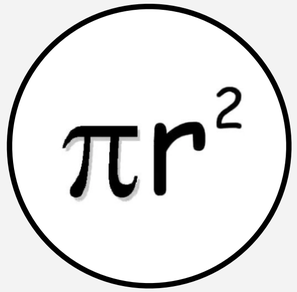

```
from math import pi

def area_of_circle(r):
```

Calculate the Area of a Circle



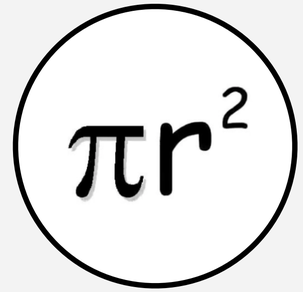

```
from math import pi

def area_of_circle(r):
    return pi*(r**2)
```

Calculate the Area of a Circle



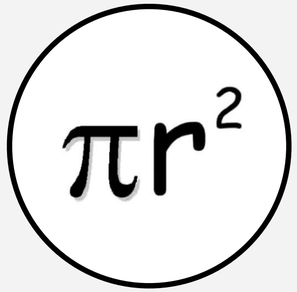

```
from math import pi

def area_of_circle(r):
    return pi*(r**2)

# Test case
```

Calculate the Area of a Circle



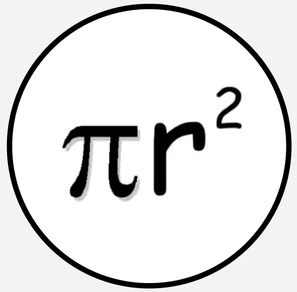

```
from math import pi

def area_of_circle(r):
    return pi*(r**2)

# Test case
radii = [2, 0, -3, True, "radius"]
```

Calculate the Area of a Circle



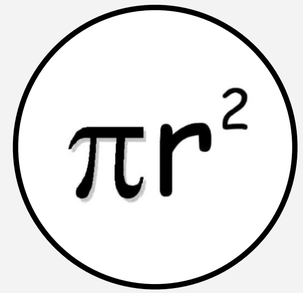

```
from math import pi

def area_of_circle(r):
    return pi*(r**2)

# Test case
radii = [2, 0, -3, True, "radius"]
message = "Area of circle with radius {radius} is {area}"
```

Calculate the Area of a Circle



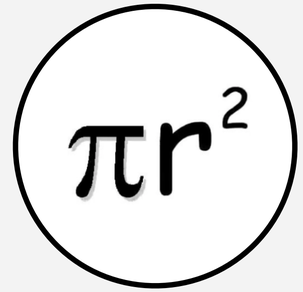

```
from math import pi

def area_of_circle(r):
    return pi*(r**2)

# Test case
radii = [2, 0, -3, True, "radius"]
message = "Area of circle with radius {radius} is {area}"

for r in radii:
```

Calculate the Area of a Circle



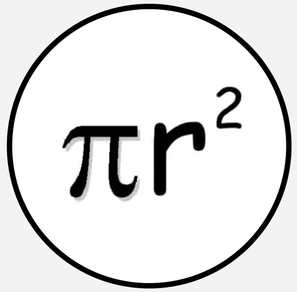

```
from math import pi

def area_of_circle(r):
    return pi*(r**2)

# Test case
radii = [2, 0, -3, True, "radius"]
message = "Area of circle with radius {radius} is {area}"

for r in radii:
   print(message.format(radius=r, area=area_of_circle(r))
```

Calculate the Area of a Circle



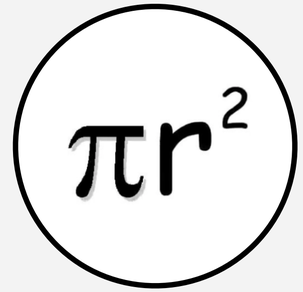

```
$ python circle.py
```

Calculate the Area of a Circle




```
$ python circle.py

Area of circle with radius 2 is 12.566370614359172
```

Calculate the Area of a Circle




```
$ python circle.py

Area of circle with radius 2 is 12.566370614359172
Area of circle with radius 0 is 0.0
```

Calculate the Area of a Circle

```
$ python circle.py

Area of circle with radius 2 is 12.566370614359172
Area of circle with radius 0 is 0.0
Area of circle with radius -3 is 28.274333882308138
```

Calculate the Area of a Circle

```
$ python circle.py

Area of circle with radius 2 is 12.566370614359172
Area of circle with radius 0 is 0.0
Area of circle with radius -3 is 28.274333882308138
Area of circle with radius True is 3.141592653589793
```

Calculate the Area of a Circle

```
$ python circle.py

Area of circle with radius 2 is 12.566370614359172
Area of circle with radius 0 is 0.0
Area of circle with radius -3 is 28.274333882308138
Area of circle with radius True is 3.141592653589793
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in area_of_circle
TypeError: unsupported operand type(s) for ** or pow():
'str' and 'int'
```

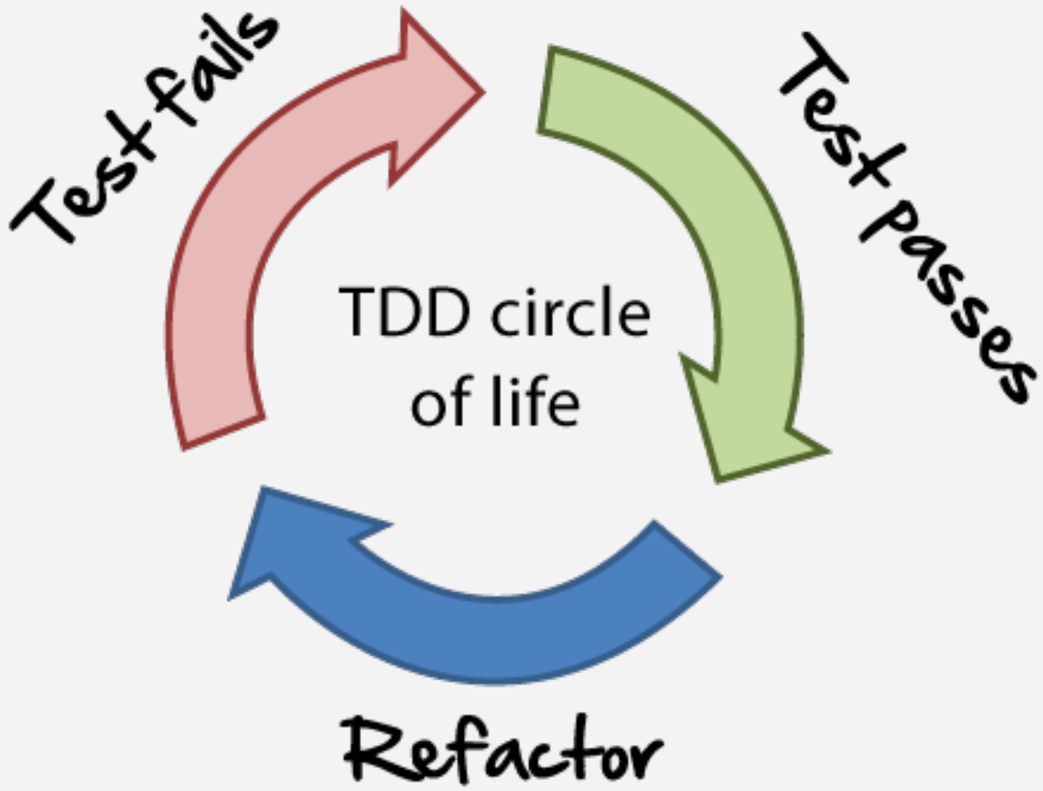
“Code that hasn't been tested should not be trusted to work”

Calculate the Area of a Circle

```
from math import pi

def area_of_circle(r):
    """ Calculates the area given a non-negative number """

    if type(r) not in [int, float]:
        raise TypeError("The radius must be a number")

    if r < 0:
        raise ValueError("The radius cannot be negative")

    return pi*(r**2)
```

Test Driven Development

Using PyUnit



What is TDD?

- Test Driven Development means that your test cases drive the design and development of your code
- You write the tests first for the *code you wish you had*, then you write the code to make the test pass
- This keeps you focused on the purpose of the code (i.e., what is it supposed to do)

Why is automated Testing Important to DevOps?

- First and foremost it saves time when developing!
- It allows you to run faster because you are more confident
- It insures that your code is working as you expected
- It insures that future changes don’t break your code
- In order to use a DevOps Pipeline, all testing must be automated

I’m not a great programmer; I’m just a good programmer with great habits

— Kent Beck —

Get in the habit of testing early and often!

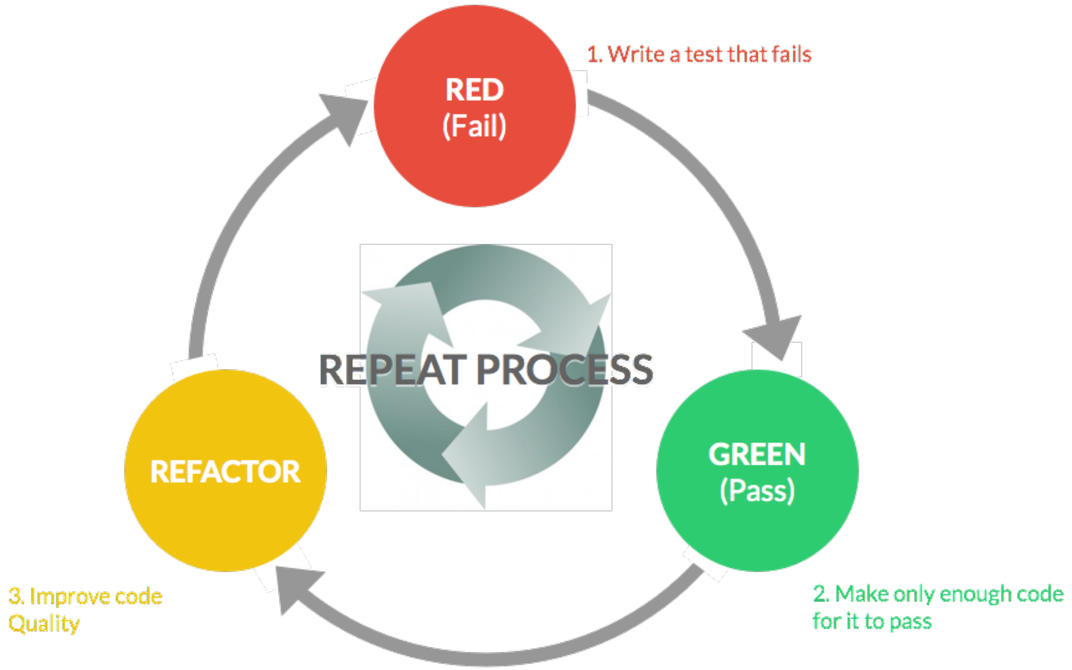
Kent Beck says Good Unit tests:

- Run fast (they have short setups, run times, and break downs).
- Run in isolation (you should be able to reorder them).
- Use data that makes them easy to read and to understand.
- Use real data (e.g. copies of production data) when they need to.
- Represent one step towards your overall goal.

The Basic TDD Workflow

- Write a failing unit test for the code you wish you had
- Write just enough code to make the unit test pass
- Refactor the code and repeat

Also know as: Red, Green, Refactor



Popular Python Test Tools

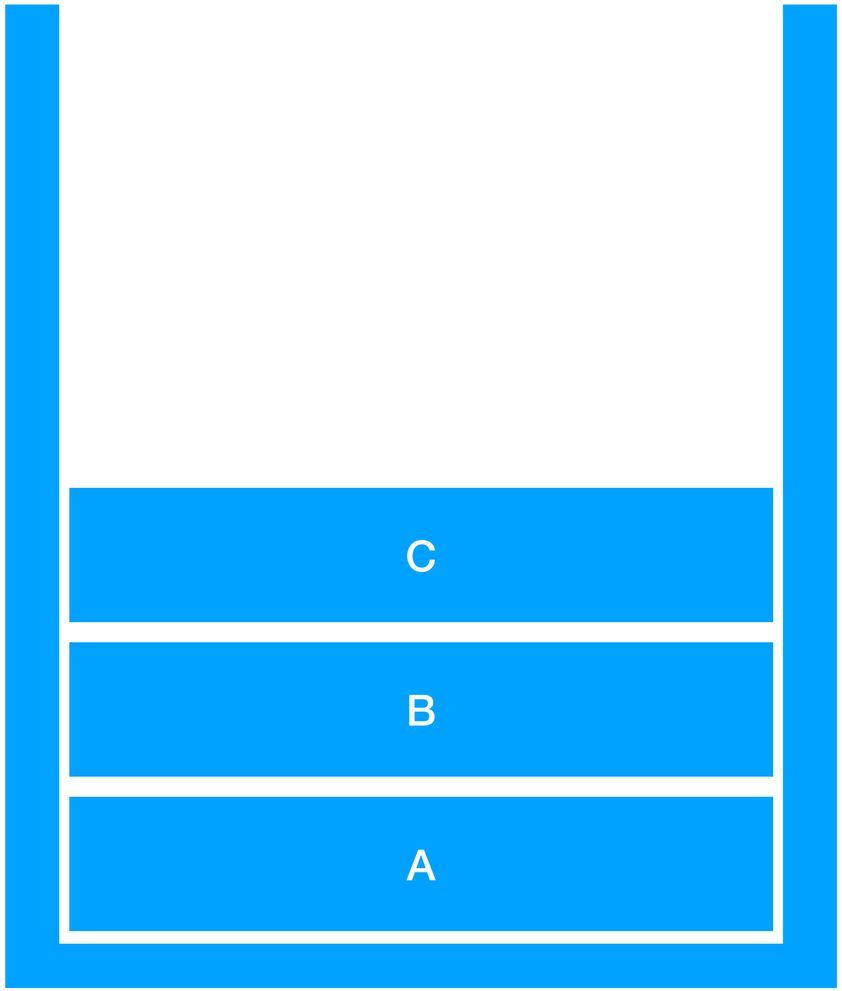
- **PyUnit:** This is what we will use. It is the standard unittest module like JUnit
- **Py.test:** Good for multiple levels of setup/ teardowns but may leads to highly unstructured and hard to read unit tests
- **Doctest:** is OK for simple things, but it's limiting and doesn't really scale for complex and highly interactive code
- **Nose:** isn't really a unit testing framework. It's a test runner and a great one at that. It can run tests created using unittest, py.test or doctest (we will also use it)

pytest

Anatomy of a Test Case

A Stack

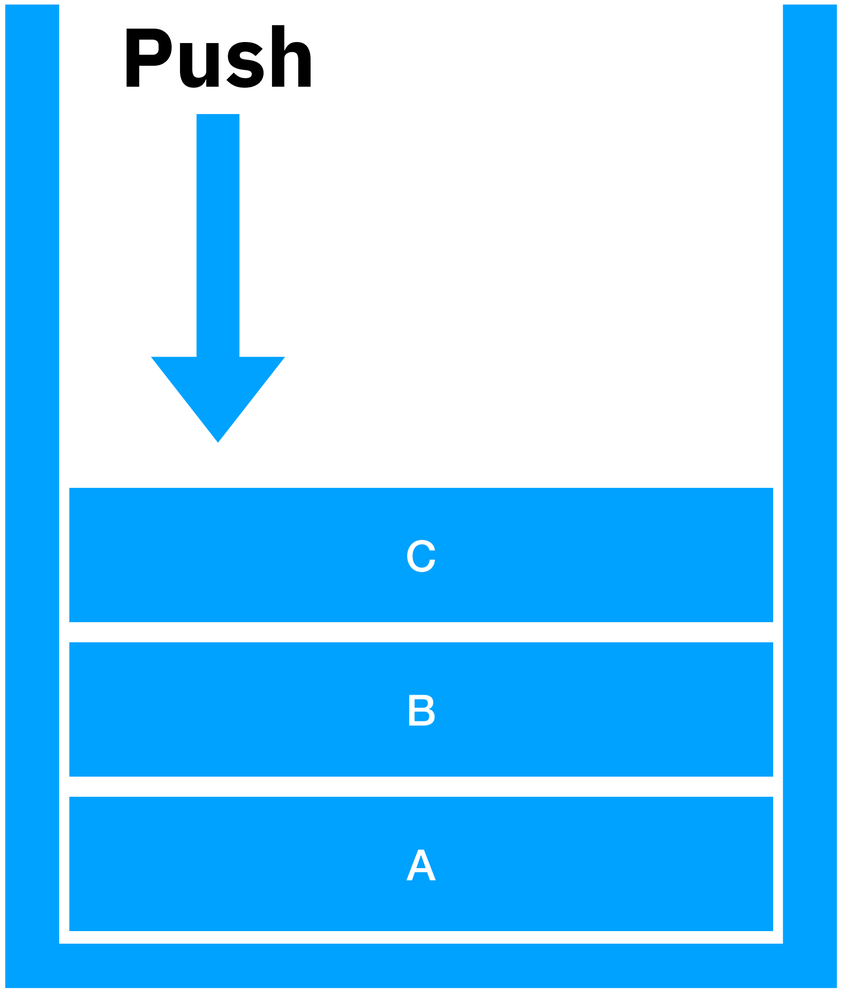
Last In, First Out (LIFO)



Anatomy of a Test Case

A Stack

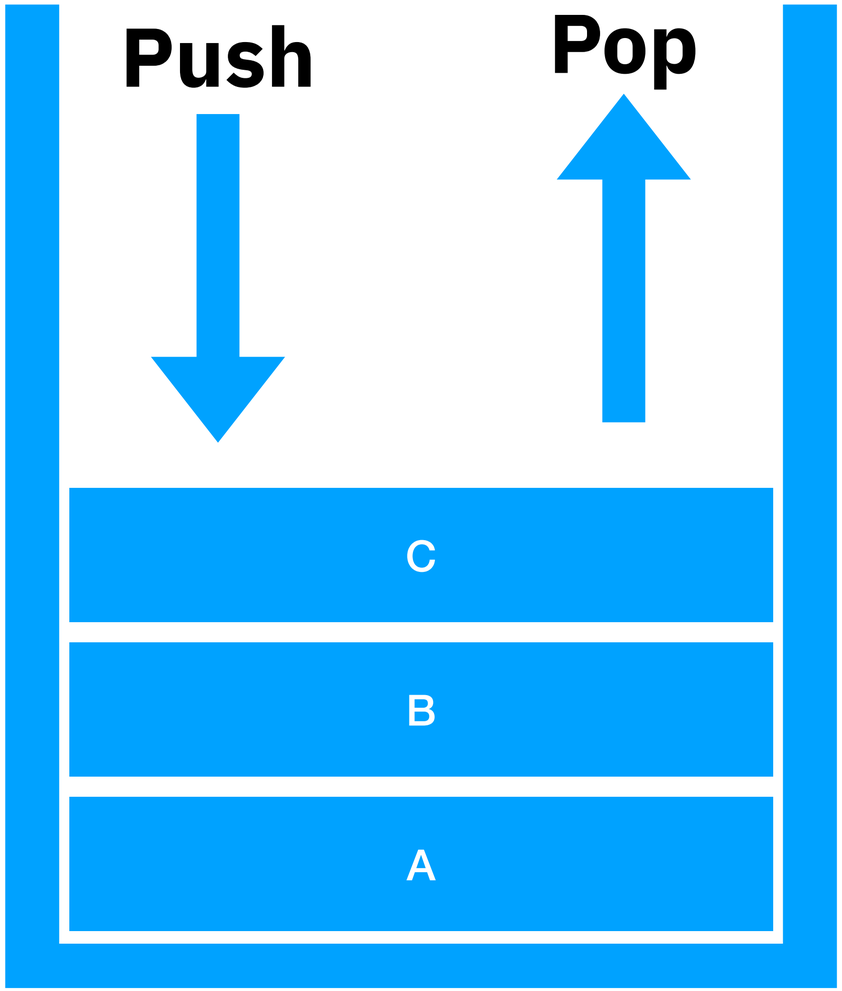
Last In, First Out (LIFO)



Anatomy of a Test Case

A Stack

Last In, First Out (LIFO)



Anatomy of a Test Case

```
import unittest
from stack import Stack

class StackTestCase(unittest.TestCase):
    def setUp(self):
       self.stack = Stack()

    def tearDown(self):
       self.stack = None

    def test_push(self):
        self.stack.push(9)
        self.assertEqual(self.stack.peek(), 9)

    def test_pop(self):
        self.stack.push(9)
        self.assertEqual(self.stack.pop(), 9)
        self.assertTrue(self.stack.isEmpty())

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

Anatomy of a Test Case

import the `unittest` module and your code

```
import unittest
from stack import Stack

class StackTestCase(unittest.TestCase):
    def setUp(self):
        self.stack = Stack()

    def tearDown(self):
        self.stack = None

    def test_push(self):
        self.stack.push(9)
        self.assertEqual(self.stack.peek(), 9)

    def test_pop(self):
        self.stack.push(9)
        self.assertEqual(self.stack.pop(), 9)
        self.assertTrue(self.stack.isEmpty())

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

Anatomy of a Test Case

```
import unittest
from stack import Stack

class StackTestCase(unittest.TestCase):
    def setUp(self):
        self.stack = Stack()

    def tearDown(self):
        self.stack = None

    def test_push(self):
        self.stack.push(9)
        self.assertEqual(self.stack.peek(), 9)

    def test_pop(self):
        self.stack.push(9)
        self.assertEqual(self.stack.pop(), 9)
        self.assertTrue(self.stack.isEmpty())

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

All tests are derived from `unitest.TestCase`

Anatomy of a Test Case

Called before and after each test case method

```
import unittest
from stack import Stack

class StackTestCase(unittest.TestCase):
    def setUp(self):
        self.stack = Stack()

    def tearDown(self):
        self.stack = None

    def test_push(self):
        self.stack.push(9)
        self.assertEqual(self.stack.peek(), 9)

    def test_pop(self):
        self.stack.push(9)
        self.assertEqual(self.stack.pop(), 9)
        self.assertTrue(self.stack.isEmpty())

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

Anatomy of a Test Case

Any method that starts with '`test_`' is assumed to be a test case

```
import unittest
from stack import Stack

class StackTestCase(unittest.TestCase):
    def setUp(self):
        self.stack = Stack()

    def tearDown(self):
        self.stack = None

    def test_push(self):
        self.stack.push(9)
        self.assertEqual(self.stack.peek(), 9)

    def test_pop(self):
        self.stack.push(9)
        self.assertEqual(self.stack.pop(), 9)
        self.assertTrue(self.stack.isEmpty())

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

Anatomy of a Test Case

```
import unittest
from stack import Stack

class StackTestCase(unittest.TestCase):
    def setUp(self):
        self.stack = Stack()

    def tearDown(self):
        self.stack = None

    def test_push(self):
        self.stack.push(9)
        self.assertEqual(self.stack.peek(), 9)

    def test_pop(self):
        self.stack.push(9)
        self.assertEqual(self.stack.pop(), 9)
        self.assertTrue(self.stack.isEmpty())

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

Runs the tests

How do we know what state we are in before the test?

Test fixtures

- A test fixture is a fixed state of a set of objects used as a baseline for running tests.
- The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.
- Examples of fixtures:
  - Preparation of input data and setup/creation of fake or mock objects
  - Loading a database with a specific, known set of data
  - Copying a specific known set of files creating a test fixture will create a set of objects initialized to certain states.

Unittest Fixtures

```
def setUpModule():                  <- runs once before any tests

def tearDownModule():               <- runs once after all tests

class MyTestCases(TestCase):

    @classmethod
    def setUpClass(cls):            <- runs once before test class

    @classmethod
    def tearDownClass(cls):         <- runs once after test class

    def setUp(self):                <- runs before each tests

    def tearDown(self):             <- runs after each tests
```

Test Fixture Example

```
def setUp(self):
    """ Runs before each test """
    service.init_db()    # initialize database
    db.drop_all()        # clean up the last tests
    db.create_all()      # create new tables
    self.app = service.app.test_client()

def tearDown(self):
    """ Runs after each test """
    db.session.remove()  # disconnect from database
    db.drop_all()        # delete all tables
```

Test Assertions

- Assertions are a check that evaluate to `True` or `False`
- Used for determining if the results of the test passed or failed
- Example:

```
sum = add(2, 3)
assert(sum == 5)
```

- Assertions will raise an exception if `False` marking the test as failed

Assertions for PyUnit

- **`assert`**: base assert allowing you to write your own assertions
- **`assertEqual(a, b)`**: check a and b are equal
- **`assertNotEqual(a, b)`**: check a and b are not equal
- **`assertIn(a, b)`**: check that a is in the item b
- **`assertNotIn(a, b)`**: check that a is not in the item b
- **`assertFalse(a)`**: check that the value of a is False
- **`assertTrue(a)`**: check the value of a is True
- **`assertIsInstance(a, TYPE)`**: check that a is of type "TYPE"
- **`assertRaises(ERROR, a, args)`**: check that when a is called with args that it raises ERROR

Test Cases for Area of Circle

```
import unittest
from circle import area_of_circle
from math import pi

class TestCircleArea(unittest.TestCase):
    def test_area(self):
        """ Test areas when radius is >= 0 """
        self.assertAlmostEqual(area_of_circle(1), pi)
        self.assertAlmostEqual(area_of_circle(0), 0)
        self.assertAlmostEqual(area_of_circle(2.1), pi * 2.1**2)

    def test_values(self):
        """ Test that ValueError is raised for bad values """
        self.assertRaises(ValueError, area_of_circle, -3)

    def test_types(self):
        """ Test that TypeError is raised with bad types """
        self.assertRaises(TypeError, area_of_circle, True)
        self.assertRaises(TypeError, area_of_circle, "radius")
```

Run the Tests

- Normally you can run the tests using:
  `python -m unittest discover`

```
$ python -m unittest discover
....................
----------------------------------------------------------------------
Ran 20 tests in 0.785s

OK
```

- But we will use: `nosetests`

Nosetests + Pinocchio

- We can get more meaningful color output by using plug-ins like nosetests and pinocchio

```
Test Cases for Pet Model
- Create a pet and add it to the database
- Create a pet and assert that it exists
- Delete a Pet
- Test deserialization of a Pet
- Test deserialization of bad data
- Find Pets by Category
- Find a Pet by Name
- Find or return 404 found
- Find or return 404 NOT found
- Find a Pet by ID
- Test serialization of a Pet
- Update a Pet

Pet Server Tests
- Create a new Pet
- Delete a Pet
- Get a single Pet
- Get a list of Pets
- Get a Pet thats not found
- Test the Home Page
- Query Pets by Category
- Update an existing Pet

Name                  Stmts   Miss  Cover   Missing
-----------------------------------------------------
service/__init__.py      26      4    85%   42, 51-54
service/models.py        64      3    95%   106, 169-170
service/service.py       84     17    80%   48, 54-56, 78-80, 93-95, 108-110, 149, 209, 248-249
-----------------------------------------------------
TOTAL                   174     24    86%
----------------------------------------------------------------------
Ran 20 tests in 1.426s

OK
```

Windows Users Beware!

- Windows cannot handle execute bits so `nosetests` won't run on a Windows share from within Linux without an additional parameter

```
$ nosetests --exe
```

This has been fixed in the Vagrantfile to force file permissions
But you still need to be aware of it happening

Nosetests Options

- Some useful command line options that you may wish to keep in mind include:
  - **-v:** gives more verbose output, including the names of the tests being executed.
  - **-s** or **-nocapture:** allows output of print statements, which are normally captured and hidden while executing tests. Useful for debugging.
  - **--nologcapture:** allows output of logging information.
  - **--rednose:** an optional plugin, which can be downloaded here, but provides colored output for the tests.
  - **--tags=TAGS:** allows you to place an @TAG above a specific test to only execute those, rather than the entire test suite

Use `setup.cfg` for Common Options

```
nosetests --verbosity 2 --with-spec --spec-color --nologcapture \
        --with-coverage --cover-erase --cover-package=app
```

```
[nosetests]
verbosity=2
with-spec=1
spec-color=1
nologcapture=1
with-coverage=1
cover-erase=1
cover-package=app
```

Test Coverage

- How do you know when you have written enough test cases?
- You use a tool like: `coverage`

```
$ pip install coverage

$ coverage run --omit "venv/*" test_server.py
$ coverage report -m --include= server.py
Name         Stmts   Miss  Cover   Missing
-----------------------------------------
server.py       81      5    94%   62, 66, 167-169
```

Test Coverage

- How do you know when you have written enough test cases?
- You use a tool like: `coverage`

We have 94% code coverage

```
$ pip install coverage

$ coverage run --omit "venv/*" test_server.py
$ coverage report -m --include= server.py
Name         Stmts   Miss  Cover   Missing
-----------------------------------------
server.py       81      5    94%   62, 66, 167-169
```

Test Coverage

- How do you know when you have written enough test cases?
- You use a tool like: `coverage`

```
$ pip install coverage

$ coverage run --omit "venv/*" test_server.py
$ coverage report -m --include= server.py
Name         Stmts   Miss  Cover   Missing
-----------------------------------------
server.py       81      5    94%   62, 66, 167-169
```

Let's take a look at 62 & 66

Missing Coverage

```
60    @app.errorhandler(405)
61    def method_not_allowed(e):
62        return make_response(jsonify(status=405, error='Method not Allowed', message='Your request
63
64    @app.errorhandler(500)
65    def internal_error(e):
66        return make_response(jsonify(status=500, error='Internal Server Error', message='Huston...
67
```

Missing Coverage

New test Case for 405

```
def test_method_not_allowed(self):
    resp = self.app.put('/pets')
    self.assertEqual(resp.status_code, status.HTTP_405_METHOD_NOT_ALLOWED)
```

New test Case for 405


```
def test_method_not_allowed(self):
    resp = self.app.put('/pets')
    self.assertEqual(resp.status_code, status.HTTP_405_METHOD_NOT_ALLOWED)
```

New test Case for 405

```
def test_method_not_allowed(self):
    resp = self.app.put('/pets')
    self.assertEqual(resp.status_code, status.HTTP_405_METHOD_NOT_ALLOWED)
```

There is no endpoint for PUT /pets

...so it should throw a 405 error

Re-run Coverage

```
$ coverage run --omit "venv/*" test_server.py
................
----------------------------------------------------------------------
Ran 16 tests in 0.176s

OK

$ coverage report -m --include=server.py
Name        Stmts   Miss  Cover   Missing
---------------------------------------
server.py      81      4    95%   66, 167-169
```

Re-run Coverage

```
$ coverage run --omit "venv/*" test_server.py
................
----------------------------------------------------------------------
Ran 16 tests in 0.176s

OK

$ coverage report -m --include=server.py
Name        Stmts   Miss  Cover   Missing
-------------------------------------------
server.py      81      4    95%   66, 167-169
```

We now have 95% code coverage (yeah!)

Hands-On

“live session”

That’s All There Is To It

- **RED**, **GREEN**, **REFACTOR**
- Continue to write Test Cases that have no implementation
- Write the code to make the tests pass
- Refactor the code as needed knowing that the tests will tell you if you broke something

What did we learn?

- You just created your first TDD PyUnit tests and run them
- You learned how to use Coverage to make sure your code is tested thoroughly
- You now have enough to automate your DevOps testing

Further Reading

- Test Driven Development with Python: http://chimera.labs.oreilly.com/books/1234000000754/

- Testing Flask: http://flask.pocoo.org/docs/0.11/testing/