Test Driven Development

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

Instructor:

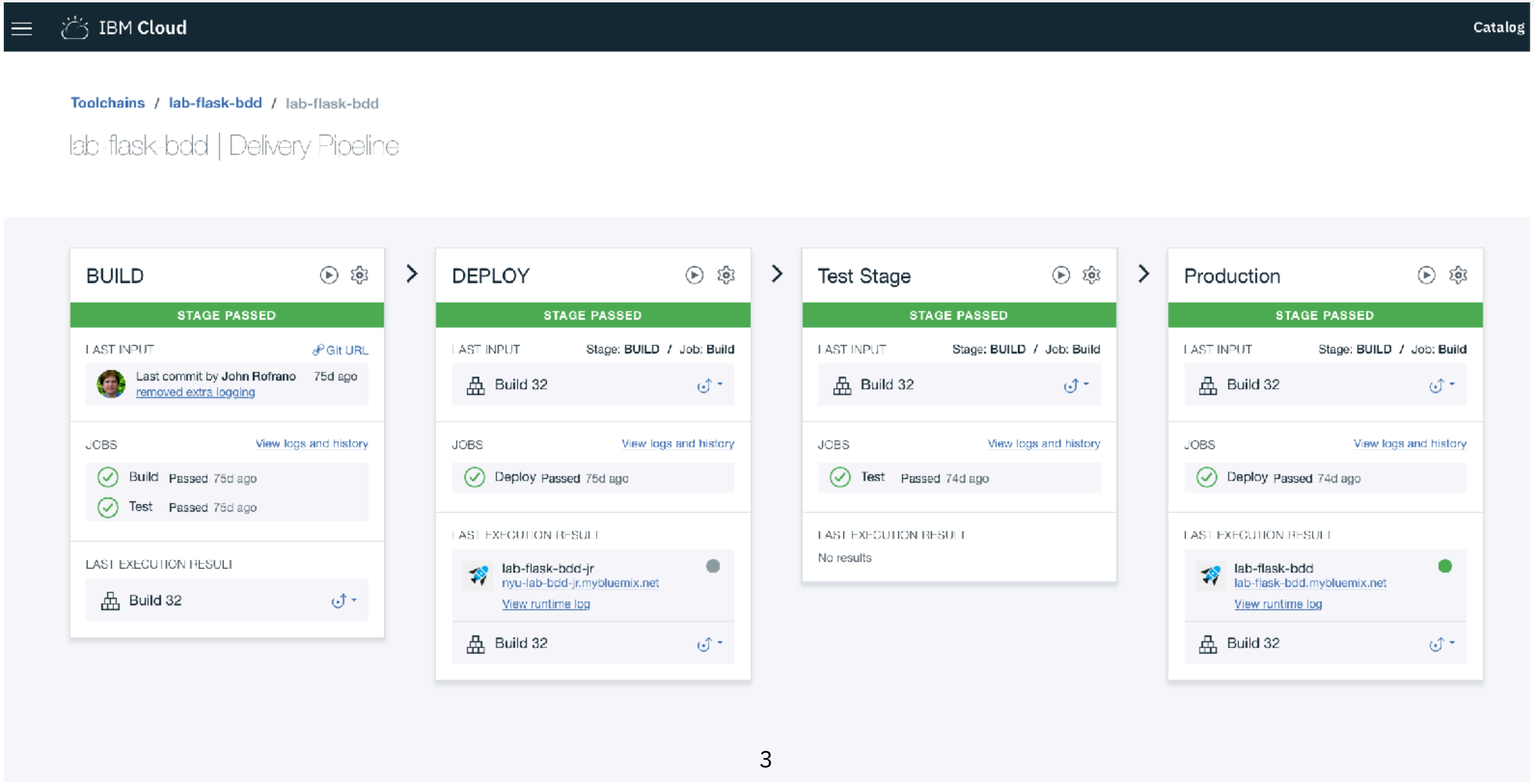
John J Rofrano

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

What is the Goal?



lab-flask-bdd | Delivery Pipeline

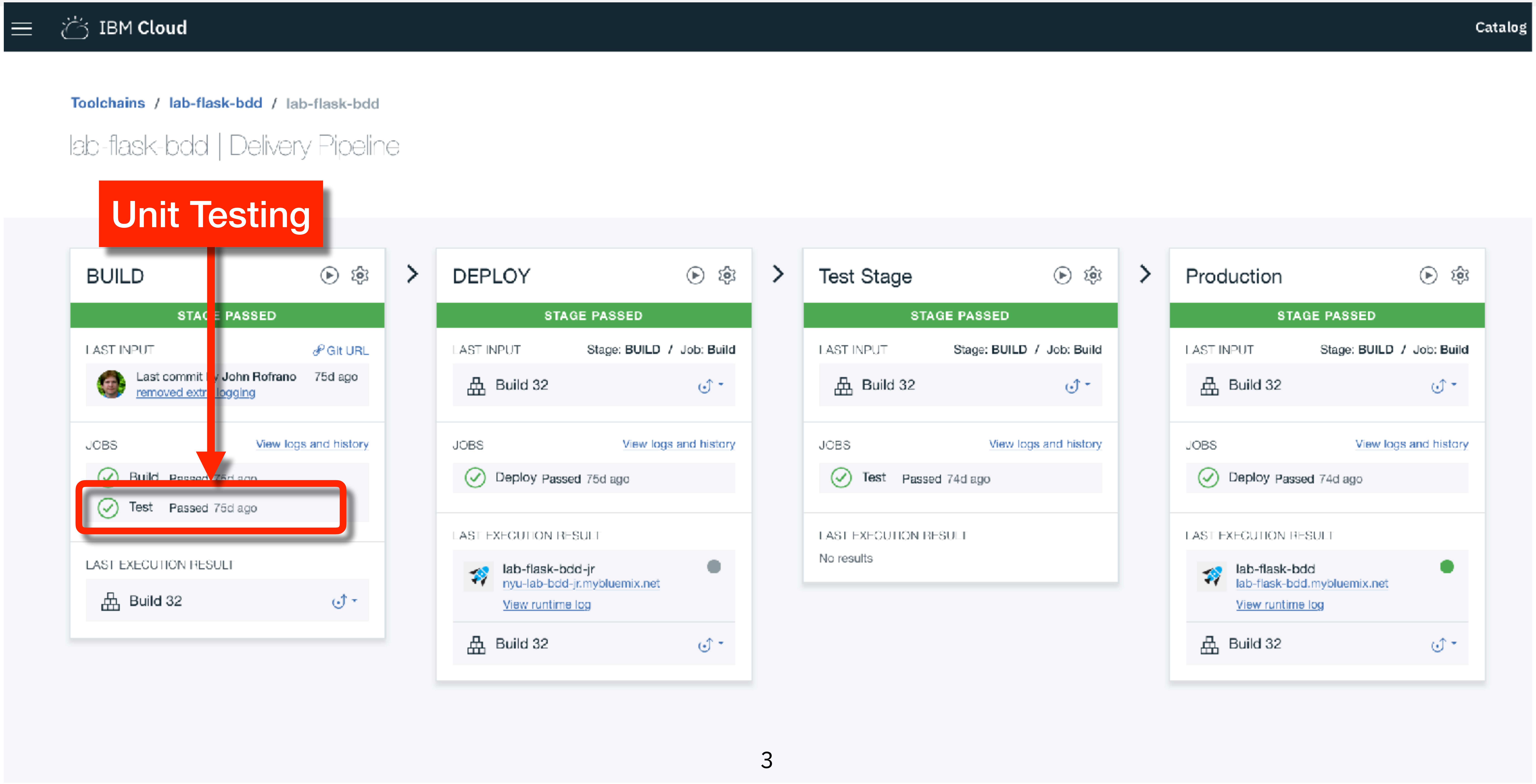
BUILD — STAGE PASSED
LAST INPUT — Git URL
Last commit by John Rofrano 75d ago — removed extra logging
JOBS — View logs and history
Build Passed 75d ago
Test Passed 75d ago
LAST EXECUTION RESULT
Build 32

DEPLOY — STAGE PASSED
LAST INPUT — Stage: BUILD / Job: Build
Build 32
JOBS — View logs and history
Deploy Passed 75d ago
LAST EXECUTION RESULT
lab-flask-bdd-jr
nyu-lab-bdd-jr.mybluemix.net
View runtime log
Build 32

Test Stage — STAGE PASSED
LAST INPUT — Stage: BUILD / Job: Build
Build 32
JOBS — View logs and history
Test Passed 74d ago
LAST EXECUTION RESULT
No results

Production — STAGE PASSED
LAST INPUT — Stage: BUILD / Job: Build
Build 32
JOBS — View logs and history
Deploy Passed 74d ago
LAST EXECUTION RESULT
lab-flask-bdd
lab-flask-bdd.mybluemix.net
View runtime log
Build 32

What is the Goal?



IBM Cloud

Catalog

Toolchains / lab-flask-bdd / lab-flask-bdd

lab-flask-bdd | Delivery Pipeline

Unit Testing

BUILD

STAGE PASSED

LAST INPUT — Git URL

Last commit by John Rofrano 75d ago
removed extra logging

JOBS — View logs and history

Build Passed 75d ago

Test Passed 75d ago

LAST EXECUTION RESULT

Build 32

DEPLOY

STAGE PASSED

LAST INPUT — Stage: BUILD / Job: Build

Build 32

JOBS — View logs and history

Deploy Passed 75d ago

LAST EXECUTION RESULT

lab-flask-bdd-jr
nyu-lab-bdd-jr.mybluemix.net
View runtime log

Build 32

Test Stage

STAGE PASSED

LAST INPUT — Stage: BUILD / Job: Build

Build 32

JOBS — View logs and history

Test Passed 74d ago

LAST EXECUTION RESULT

No results

Production

STAGE PASSED

LAST INPUT — Stage: BUILD / Job: Build

Build 32

JOBS — View logs and history

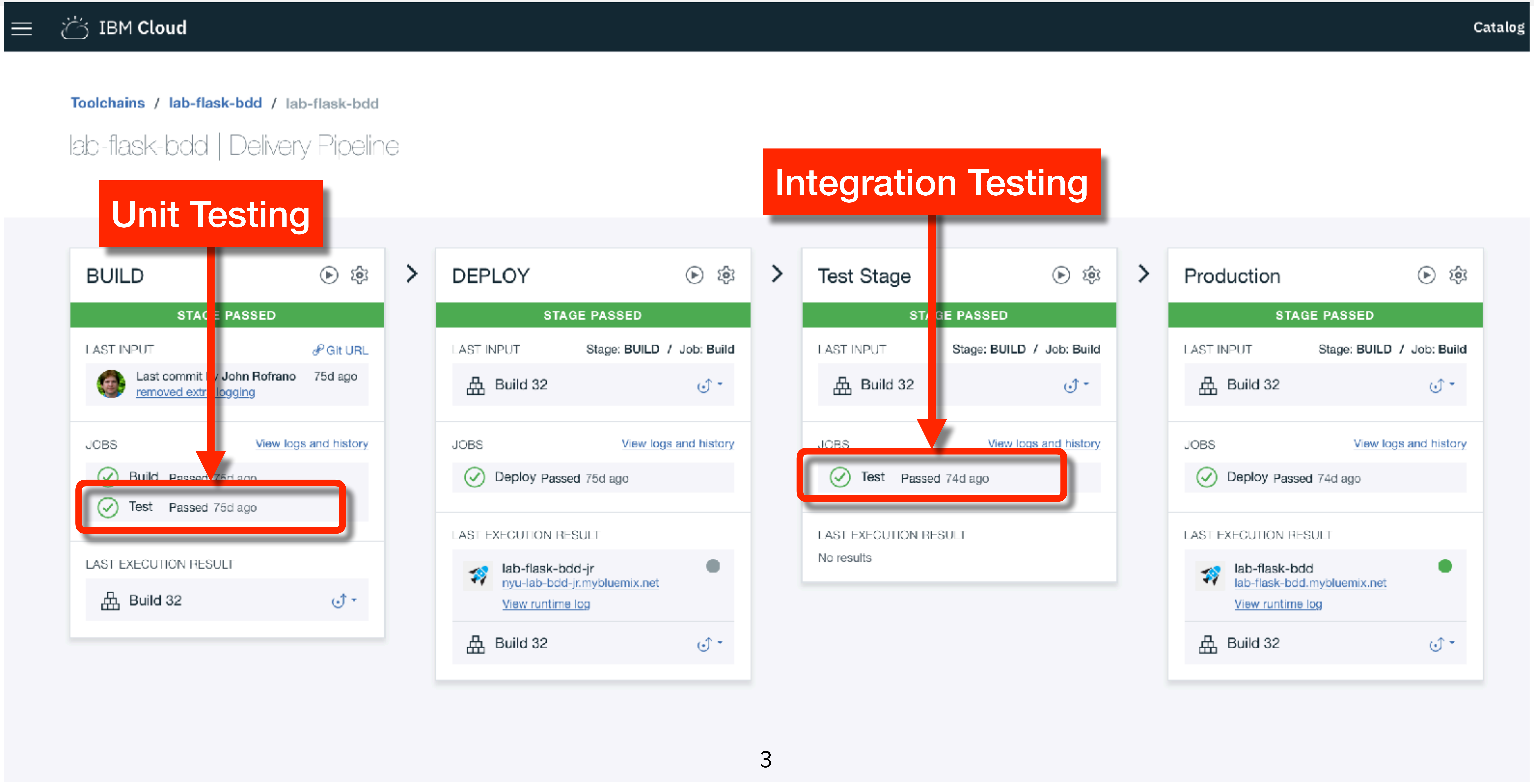
Deploy Passed 74d ago

LAST EXECUTION RESULT

lab-flask-bdd
lab-flask-bdd.mybluemix.net
View runtime log

Build 32

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

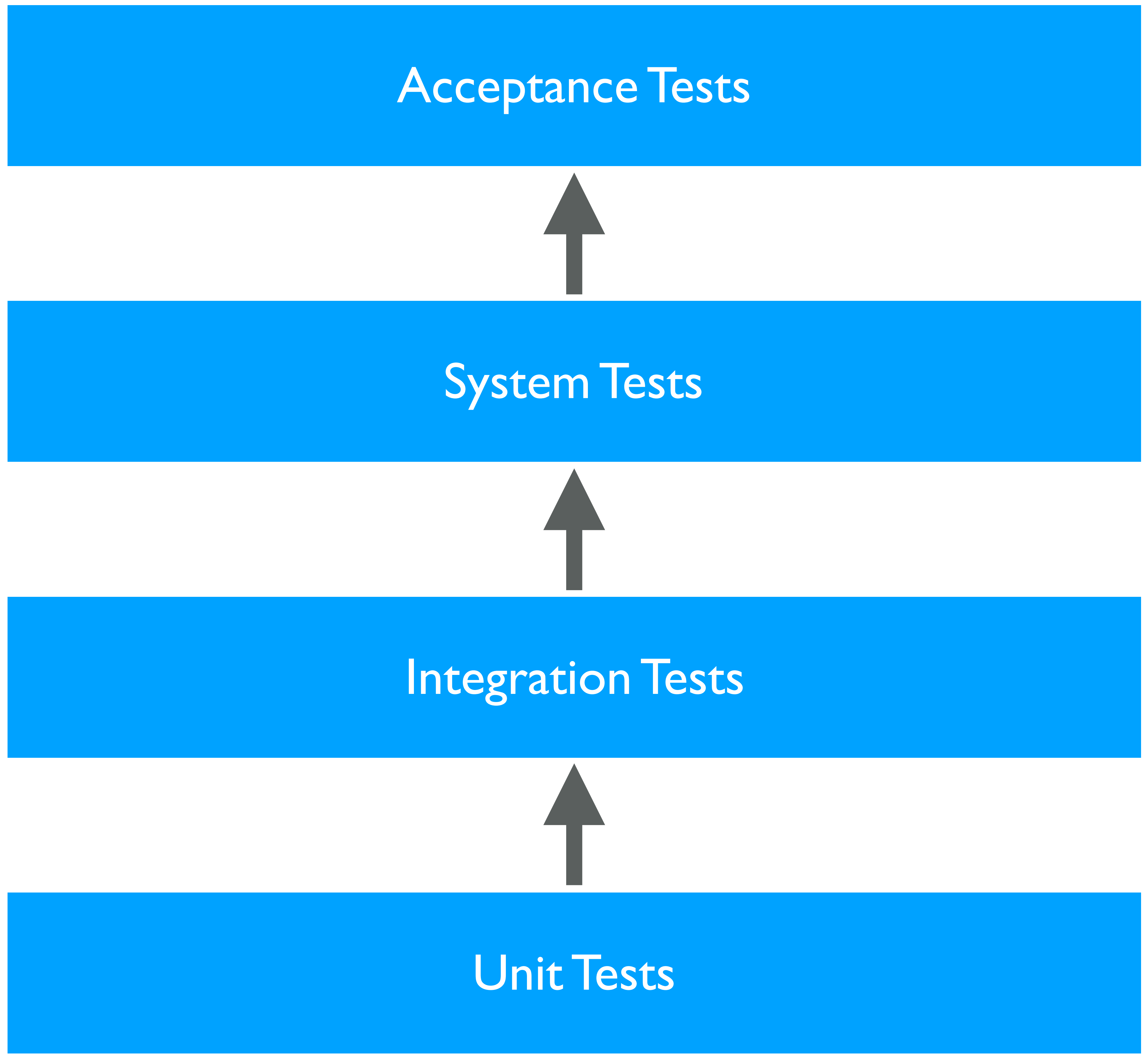
Why Developers Don’t Test

- 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?

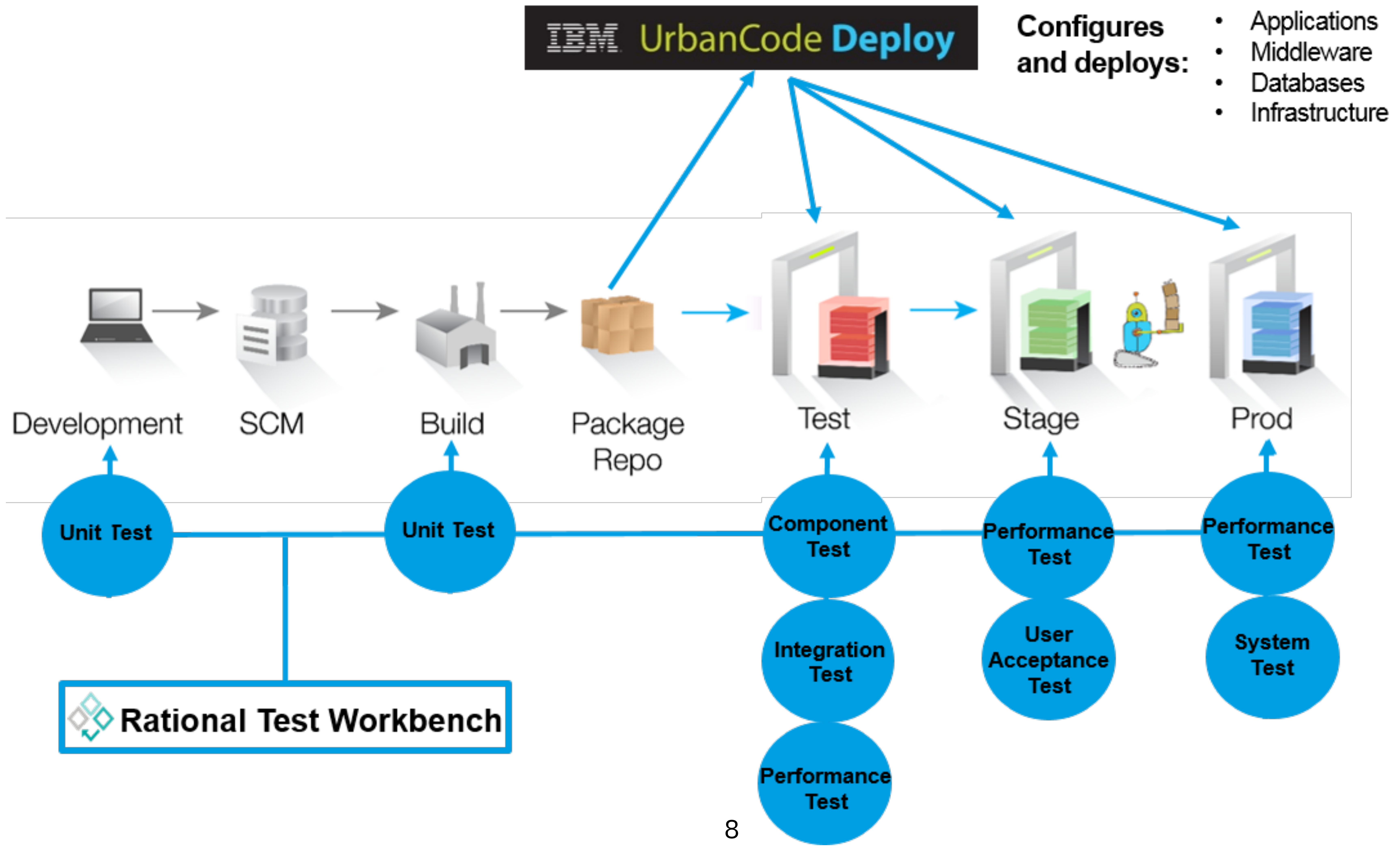
Software Testing Levels

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.



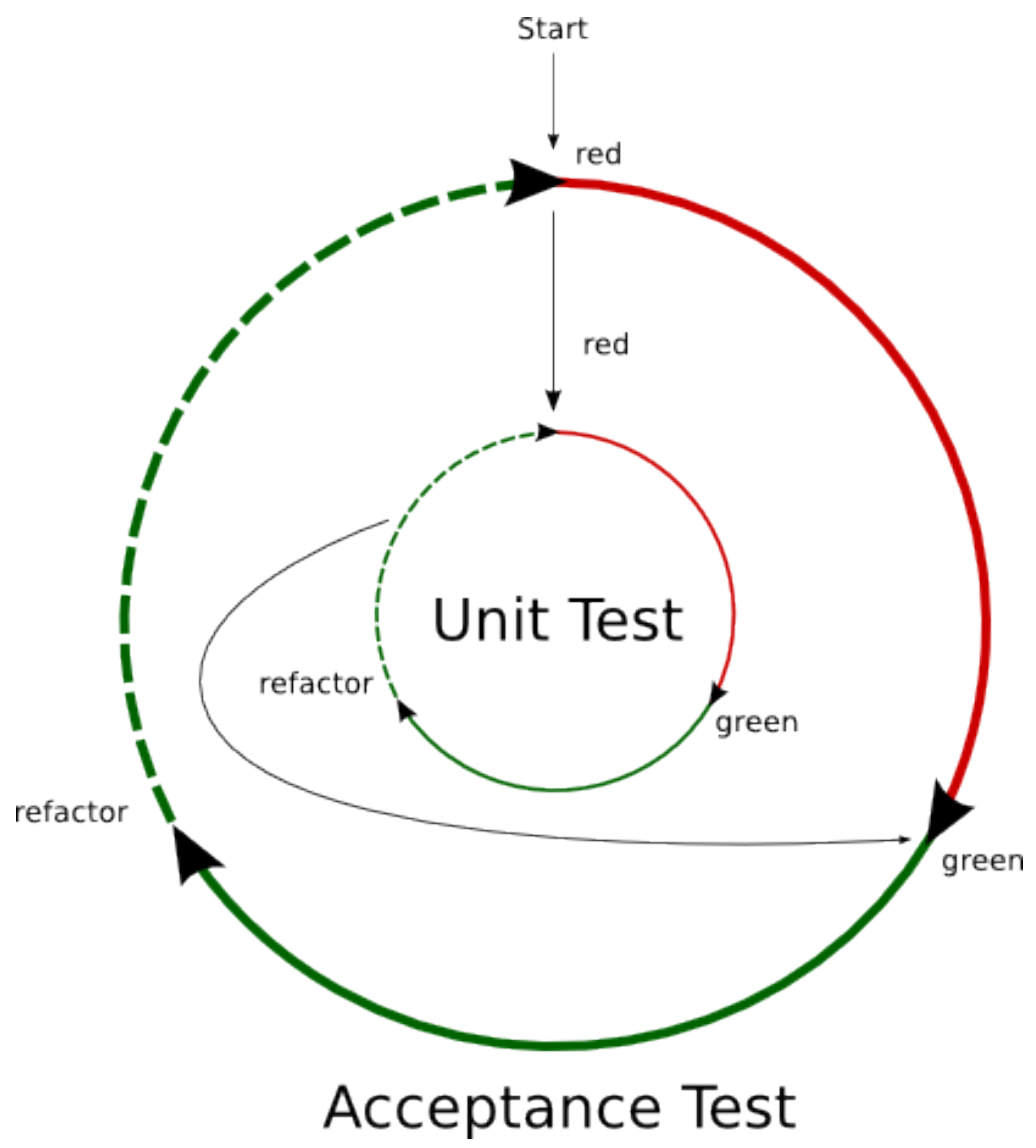
http://softwaretestingfundamentals.com/software-testing-lev

Traditional Release Cycle

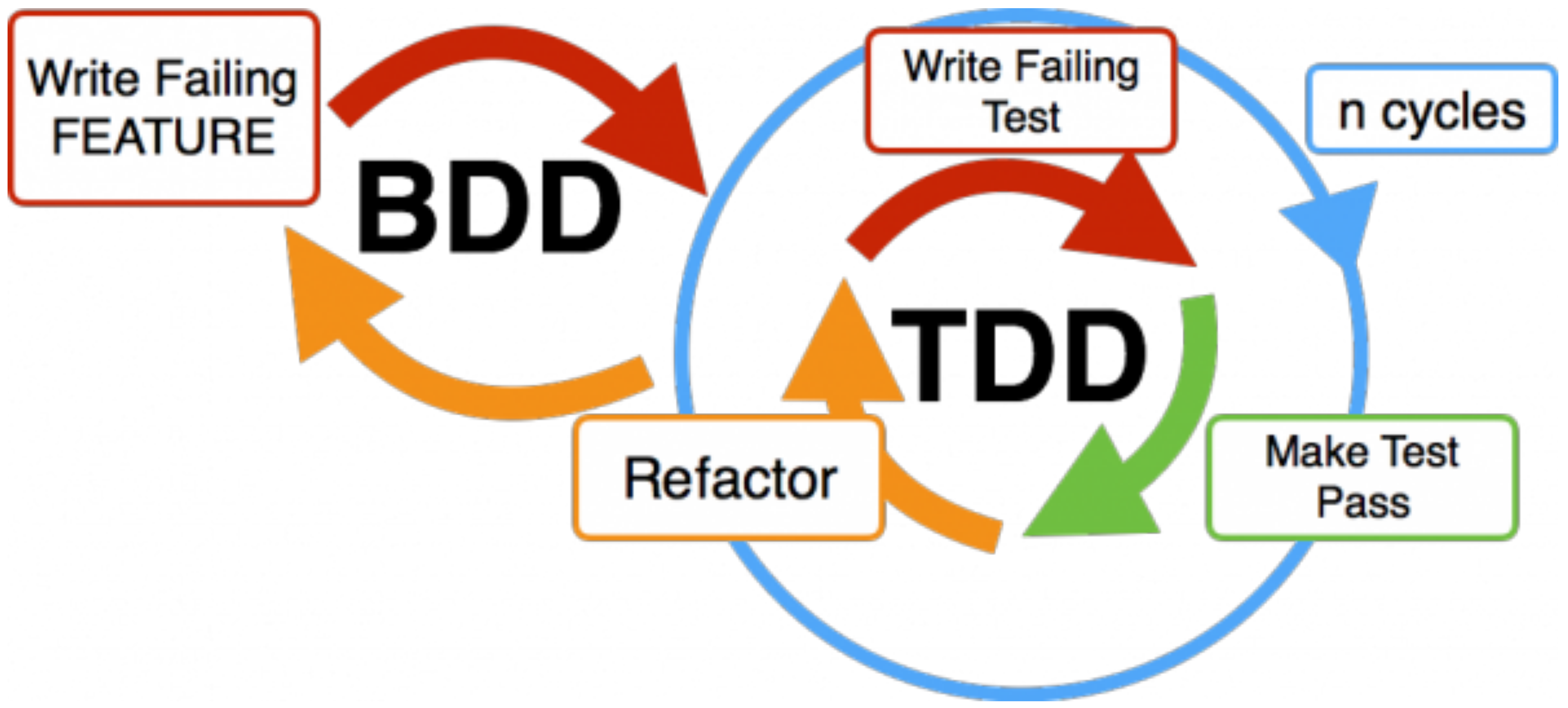


What is BDD & TDD

- 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



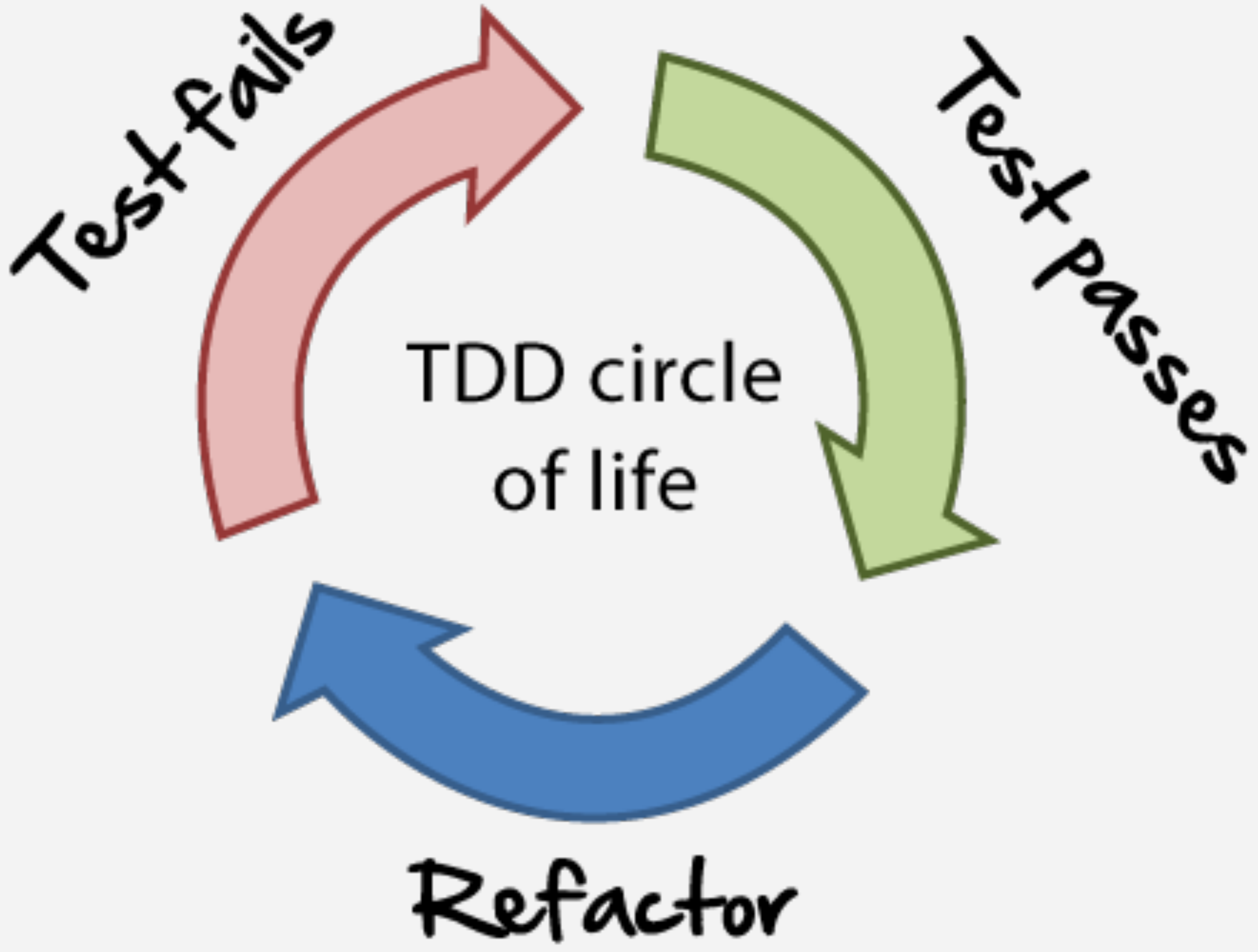
BDD & TDD Cycle



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

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!

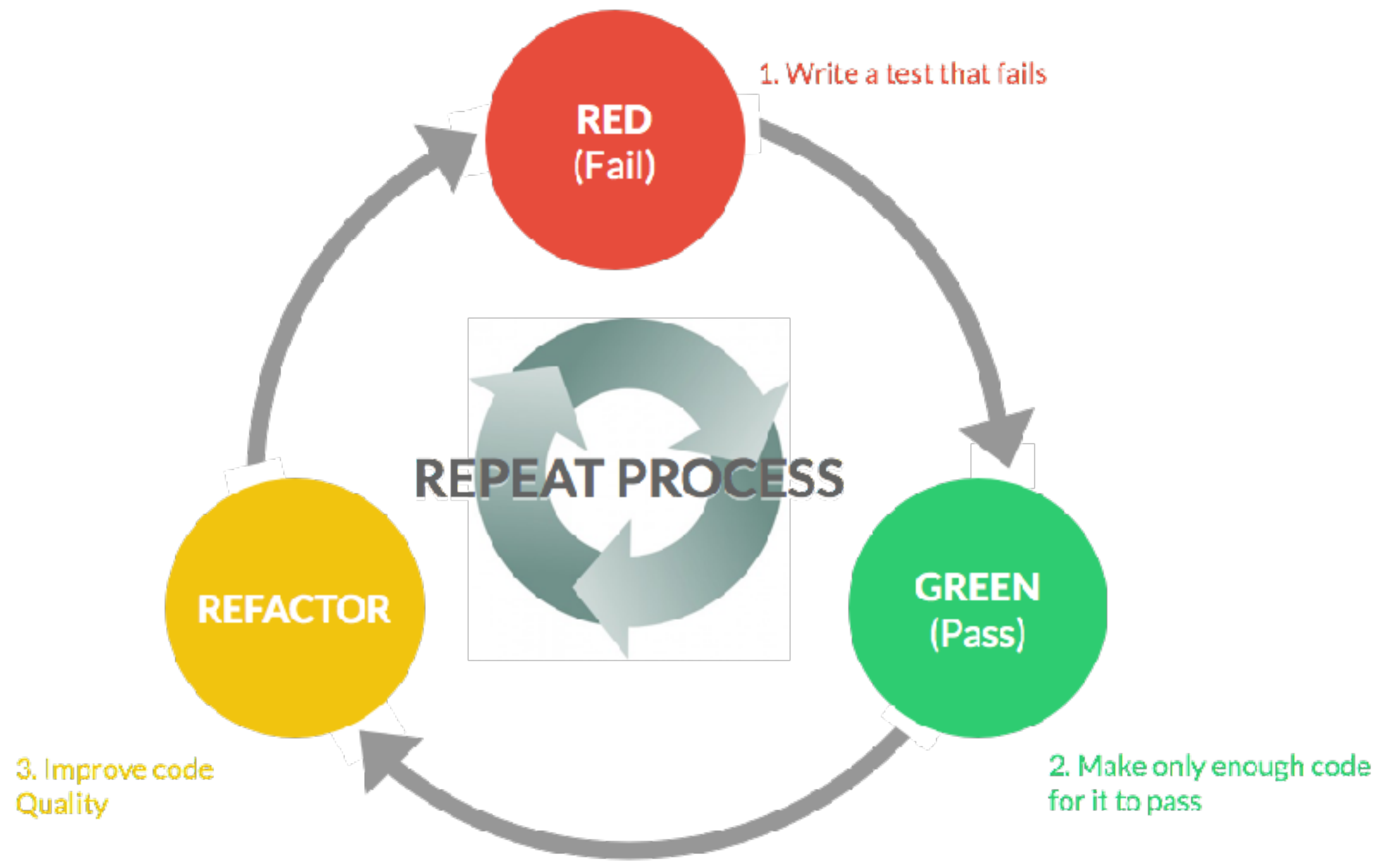
Kent Beck says Good Unit tests:

- 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


RED
(Fail)
1. Write a test that fails
GREEN
(Pass)
2. Make only enough code for it to pass
REFACTOR
3. Improve code Quality
REPEAT PROCESS

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)

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

All tests are derived from `unitest.TestCase`

```
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

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

```
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()
```

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

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

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
```

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

Tests Cases for Pet Demo

```
import unittest
import json
from server import app, db, Pet

class TestPetServer(unittest.TestCase):
   def setUp(self):
        # Set up the test database
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db/test.db'
        db.drop_all()    # clean up the last tests
        db.create_all()  # make our sqlalchemy tables
        db.session.add(Pet(name='fido', category='dog'))
        db.session.add(Pet(name='kitty', category='cat'))
        db.session.commit()
        self.app = app.test_client()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
```

Tests Cases for Pet Demo

```
def test_index(self):
    resp = self.app.get('/')
    self.assertEqual( resp.status_code, status.HTTP_200_OK )
    self.assertTrue ('Pet Demo REST API Service' in resp.data)

def test_get_pet_list(self):
    resp = self.app.get('/pets')
    self.assertEqual( resp.status_code, status.HTTP_200_OK )
    self.assertTrue( len(resp.data) > 0 )

def test_get_pet(self):
    resp = self.app.get('/pets/2')
    self.assertEqual( resp.status_code, status.HTTP_200_OK )
    data = resp.get_json()
    self.assertEqual (data['name'], 'kitty')

def test_get_pet_not_found(self):
    resp = self.app.get('/pets/0')
    self.assertEqual( resp.status_code, status.HTTP_404_NOT_FOUND )
```

Tests Cases for Pet Demo

```
def test_create_pet(self):
    # save the current number of pets for later comparison
    pet_count = self.get_pet_count()
    # add a new pet
    new_pet = {'name': 'sammy', 'category': 'snake'}
    resp = self.app.post('/pets', json=new_pet, content_type='application/json')
    self.assertEqual( resp.status_code, status.HTTP_201_CREATED )
    # Make sure location header is set
    location = resp.headers.get('Location', None)
    self.assertTrue( location != None)
    # Check the data is correct
    new_json = resp.get_json()
    self.assertEqual (new_json['name'], 'sammy')
    # check that count has gone up and includes sammy
    resp = self.app.get('/pets')
    data = resp.get_json()
    self.assertEqual( resp.status_code, status.HTTP_200_OK )
    self.assertEqual( len(data), pet_count + 1 )
    self.assertIn( new_json, data )
```

Install the Python Dependencies

- The Vagrantfile sets up a Python 3 virtual environment (so you don't have to)

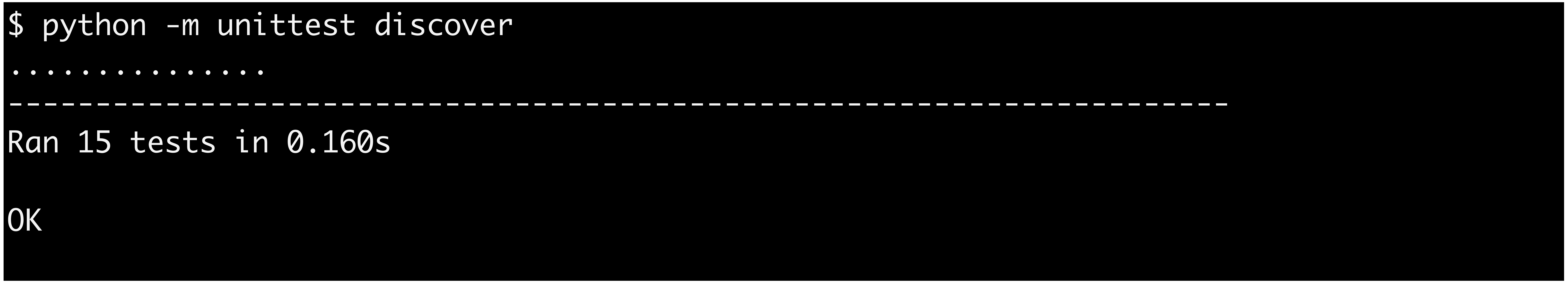
```
python3 -m venv ~/venv
```

- You just need to run pip to install the packages in requirements.txt

```
pip install -r requirements.txt
```

Run the Tests

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

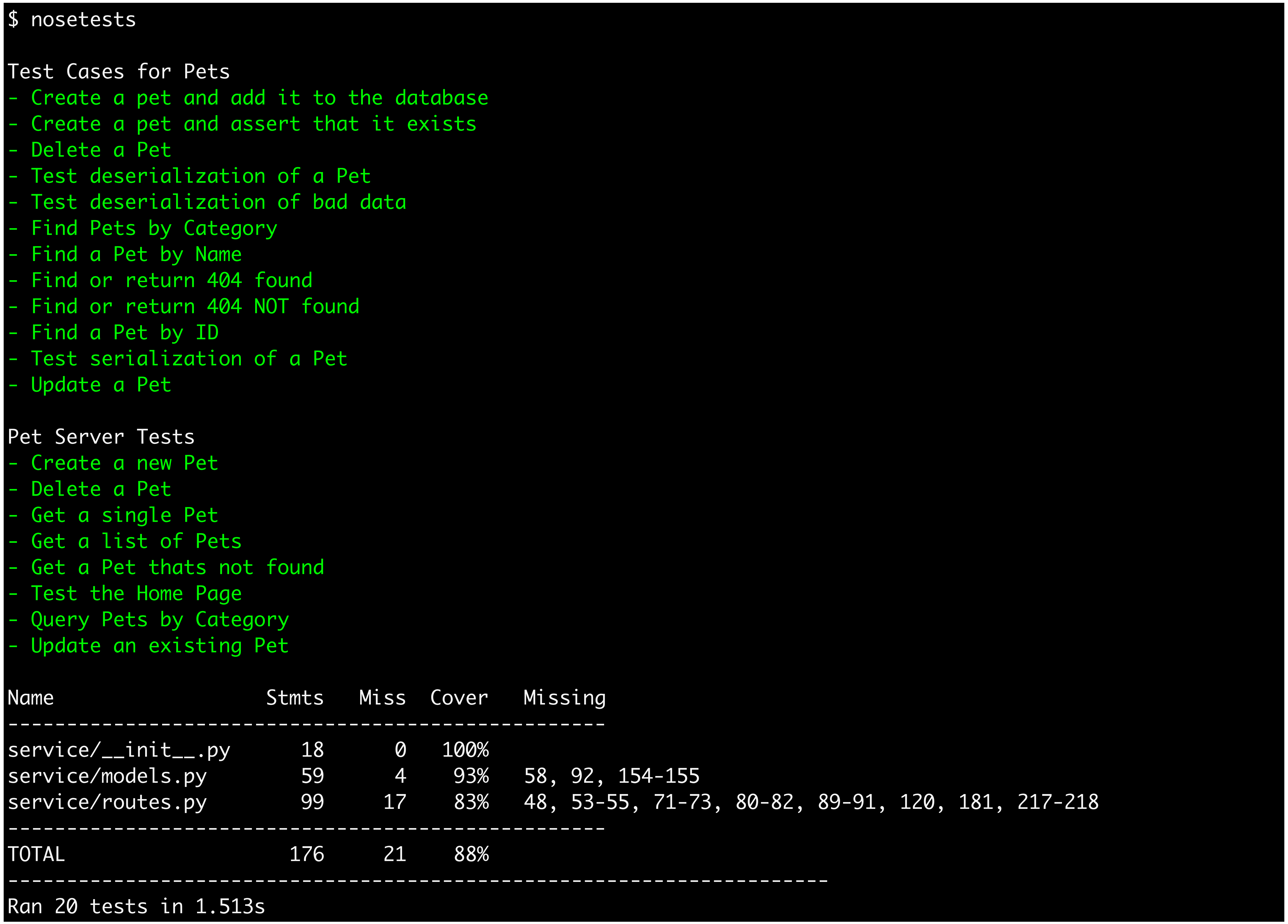


```
$ python -m unittest discover
...............
----------------------------------------------------------------------
Ran 15 tests in 0.160s

OK
```

- But we will use: `nosetests`

Run nosetests



```
$ nosetests

Test Cases for Pets
- 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      18      0   100%
service/models.py        59      4    93%   58, 92, 154-155
service/routes.py        99     17    83%   48, 53-55, 71-73, 80-82, 89-91, 120, 181, 217-218
---------------------------------------------------
TOTAL                   176     21    88%
----------------------------------------------------------------------
Ran 20 tests in 1.513s
```

@JohnRofrano

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:** controls 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 \
         --with-coverage --cover-erase --cover-package=service
```

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

Test Coverage

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

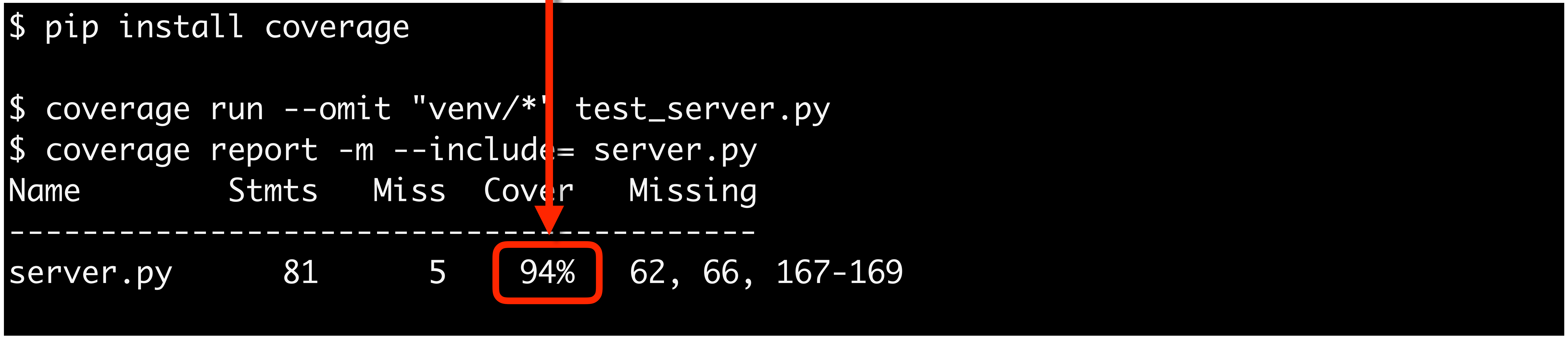
```
$ pip install coverage

$ 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

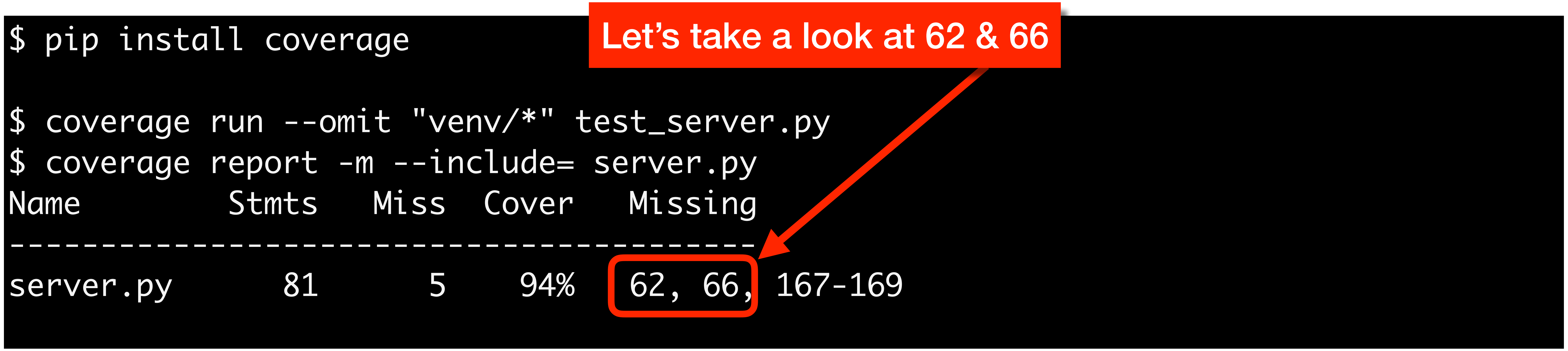


```
$ pip install coverage

$ 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

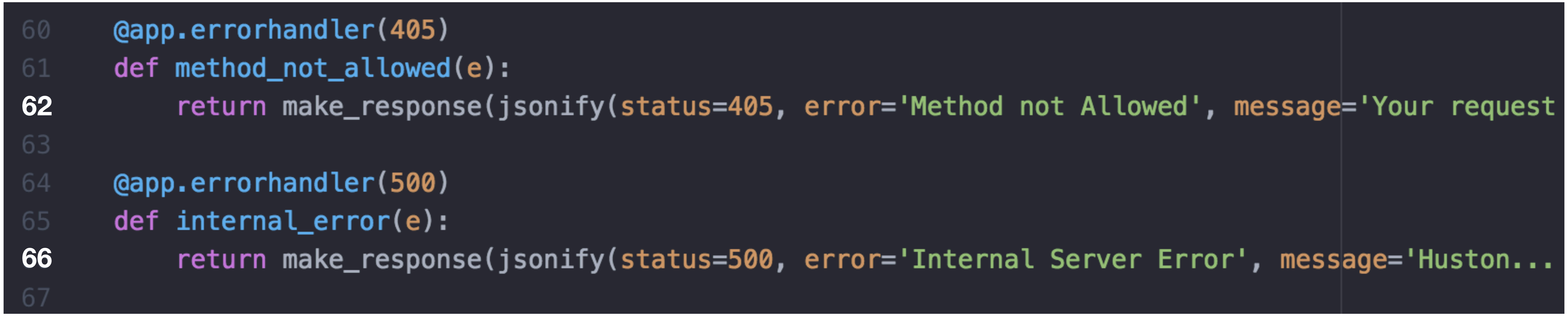
- How do you know when you have written enough test cases?
- You use a tool like: `coverage`



```
$ 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
```

Missing Coverage

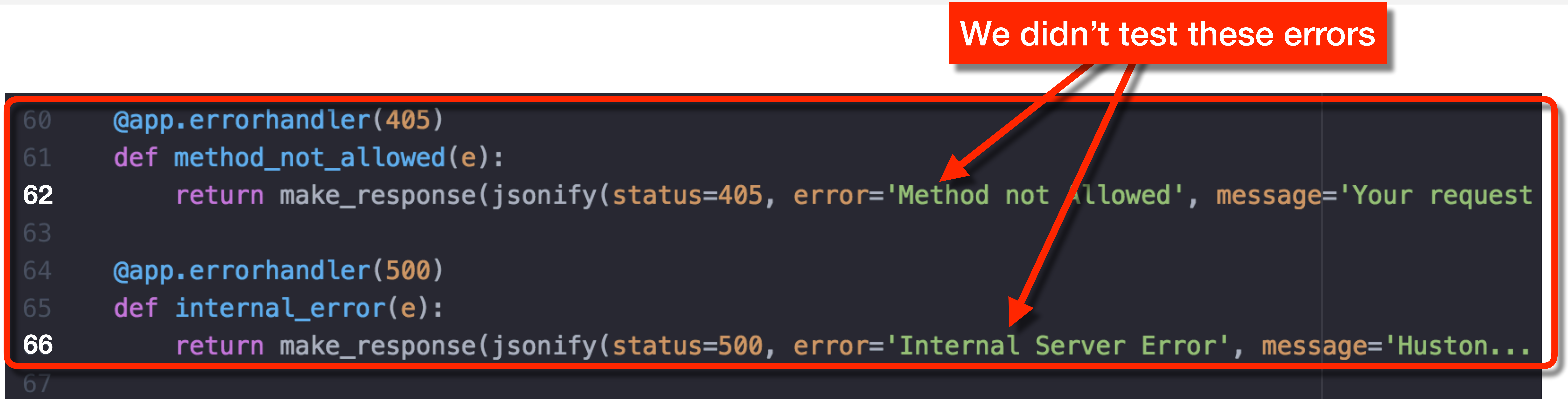


```
@app.errorhandler(405)
def method_not_allowed(e):
    return make_response(jsonify(status=405, error='Method not Allowed', message='Your request

@app.errorhandler(500)
def internal_error(e):
    return make_response(jsonify(status=500, error='Internal Server Error', message='Huston...

```

Missing Coverage



```
@app.errorhandler(405)
def method_not_allowed(e):
    return make_response(jsonify(status=405, error='Method not Allowed', message='Your request

@app.errorhandler(500)
def internal_error(e):
    return make_response(jsonify(status=500, error='Internal Server Error', message='Huston...
```

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)
```

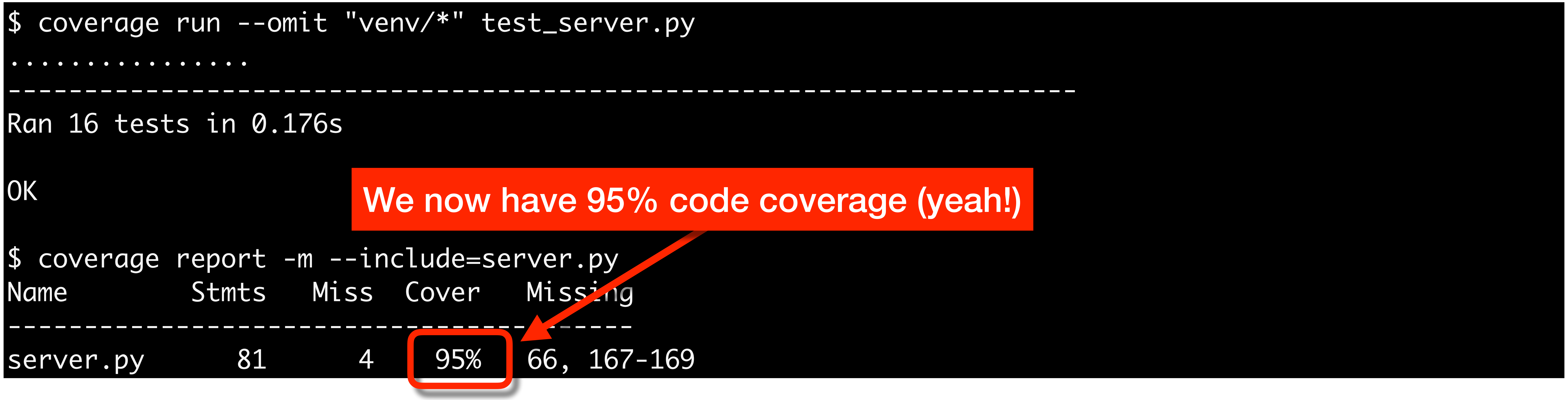
Re-run Coverage

```
$ 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
```

Let's look at some Test Cases!