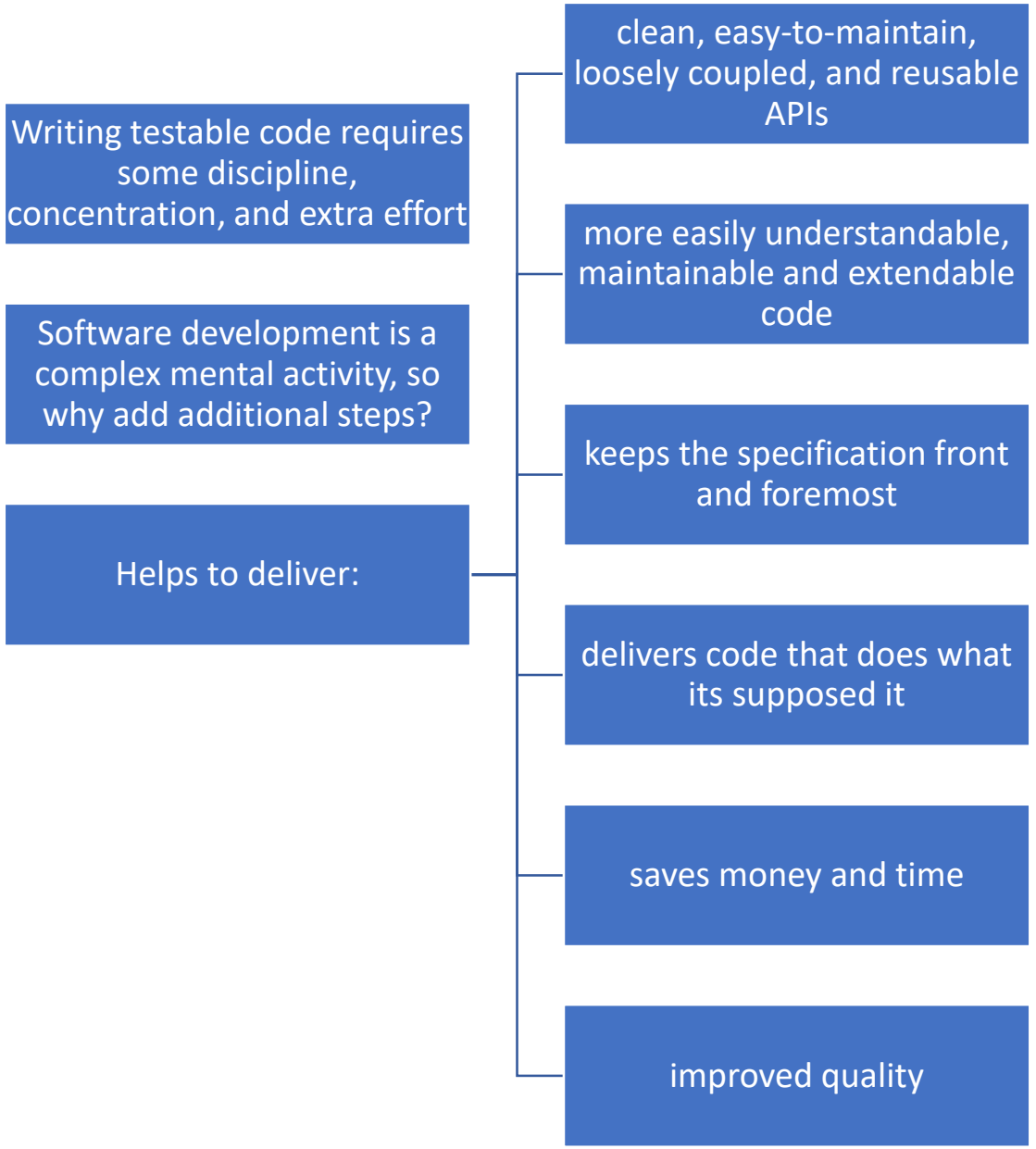
Overview of Code Testing

Coffee and Coding
12th November 2019

What is a test?

- Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not
- Testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements

Why test?


Writing testable code requires some discipline, concentration, and extra effort
Software development is a complex mental activity, so why add additional steps?
Helps to deliver:
clean, easy-to-maintain, loosely coupled, and reusable APIs
more easily understandable, maintainable and extendable code
keeps the specification front and foremost
delivers code that does what its supposed it
saves money and time
improved quality

Types of testing

- Function
- Performance
- Security
- Unit
- System
- Integration
- User interface
- OS
- Harness
- Automatic
- Manual
- ……. And many more!

What tests to use?

What might be useful for us?

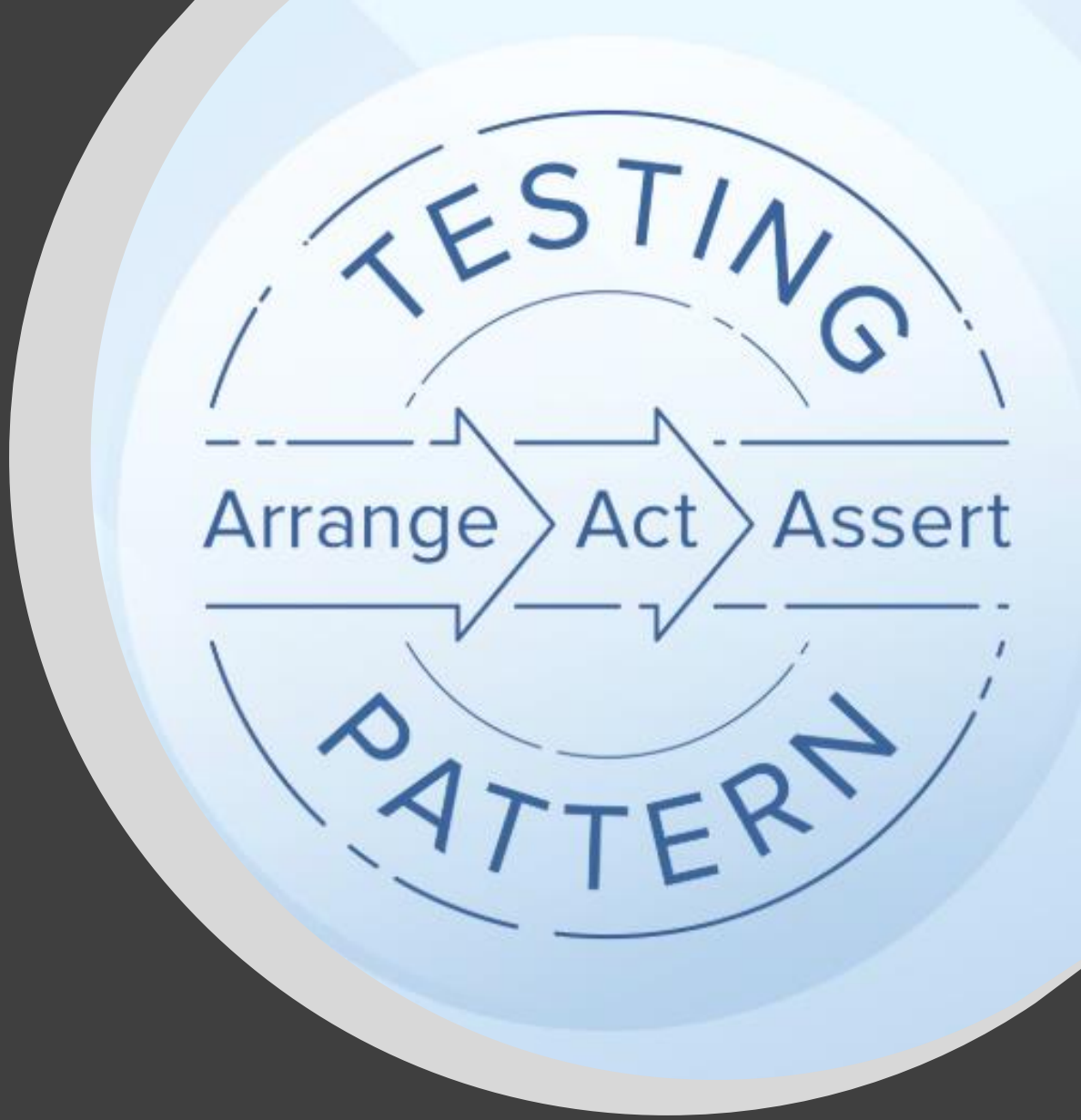
What is Unit Testing?

Unit Testing is where a small portion of software is tested in isolation without affecting anything else

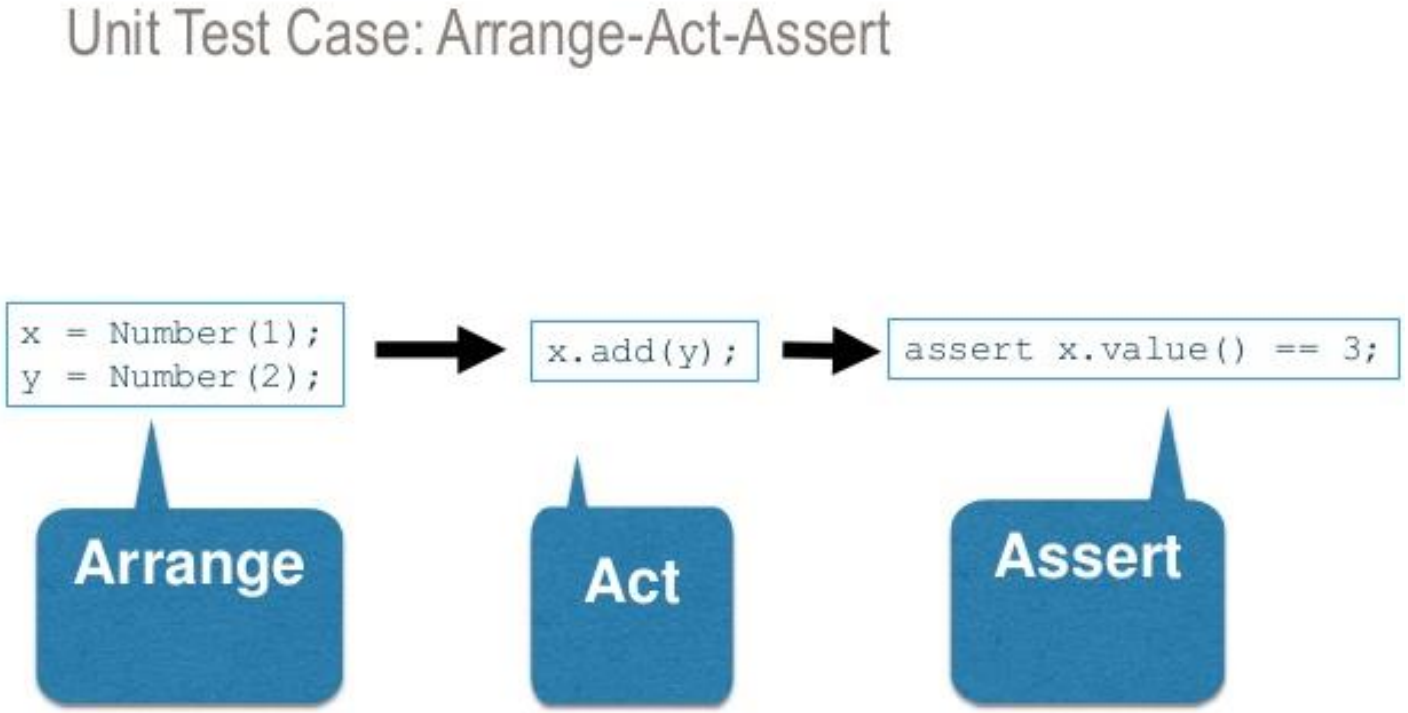
This is used to validate that each unit of the software performs as designed, independently from other parts

1. Initialise small part of application (SUT)
2. Applies stimulus to the SUT (ask it to do something)
3. Observe the resulting behaviour

If the behaviour is consistent with expectations it passes, otherwise it fails



What might this look like?



Building a system

- Lets build a chimera from different animal parts
- How do we know each part works?

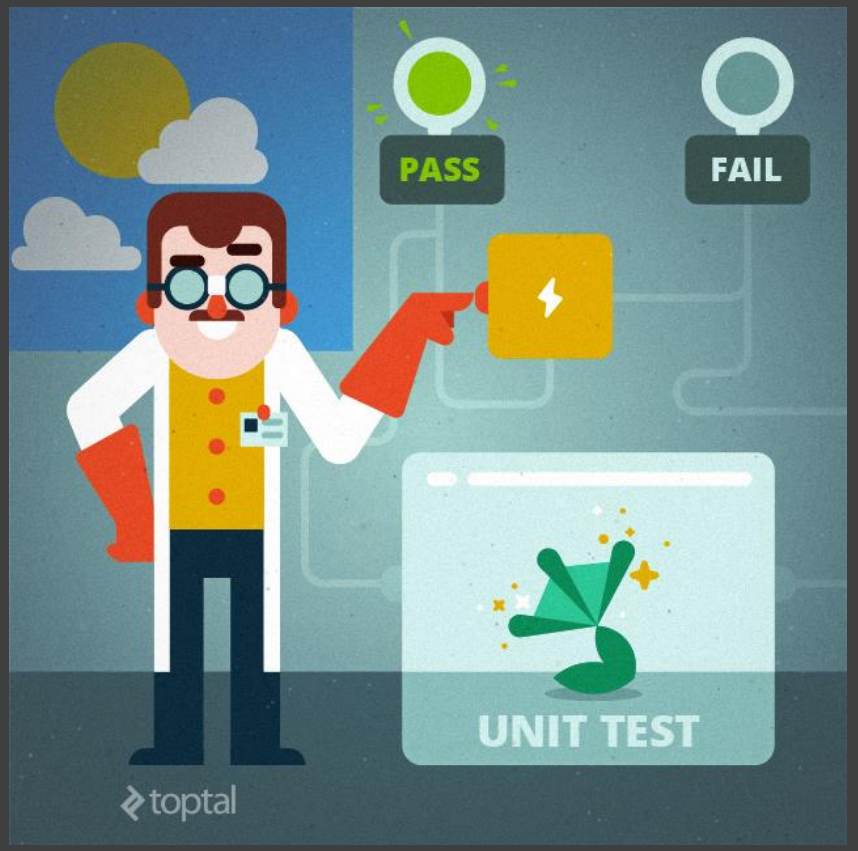
What type of test?

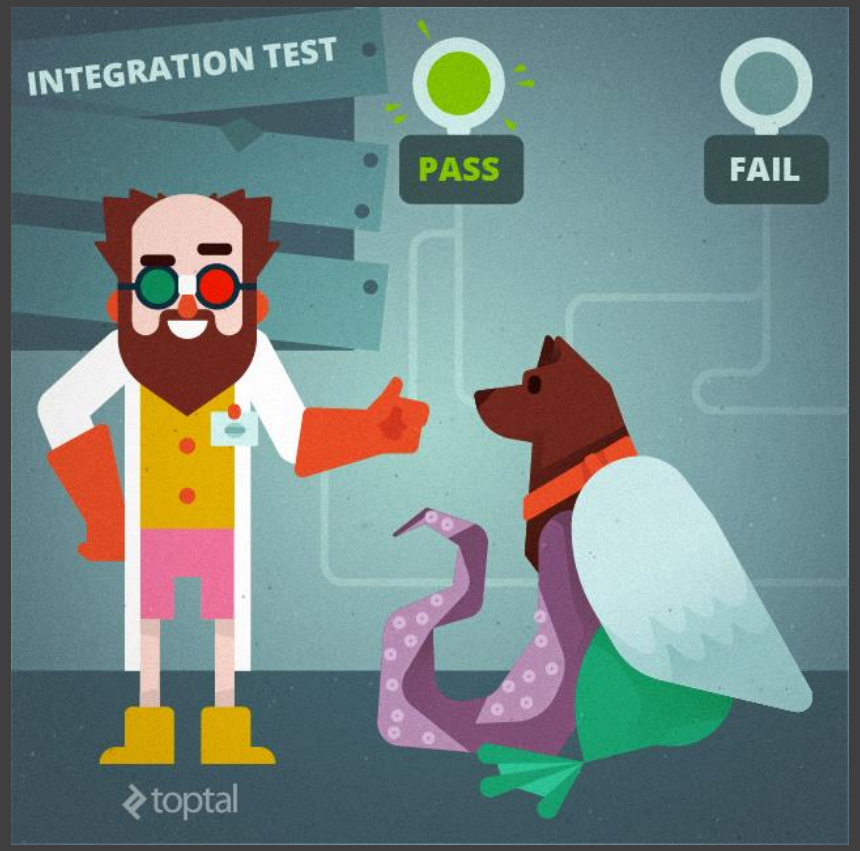
Electrical stimulus to a single frog leg

Testing to see if it can fly

Ensuring the tentacles can grip a pole

Testing if the chimera can fly and land





Different types of testing

Different Types of testing

Testing a part – narrow and independent

Integration testing – testing that independent parts work together in the real life environment

Real life systems have to test of complex scenarios and often require external sources

Make sure you know the test type

- Unit testing should be narrow and focussed
- Its relatively easy to implement
- If you are writing a unit test and find you need external resources then it might be worth reviewing your design

Unit tests should be….

- Easy to write
- Readable
- Reliable
- Fast
- Decoupled and isolated
- Maintainable

unit tests are quite easy; the real problems that complicate unit testing, and introduce expensive complexity, are a result of poorly-designed, untestable code

Example function

- E.g., lets assume you have a function that returns time of day
- It gets the time from the system and returns a string indicating the portion of the day
- How do you test this? How could you automate this test so it doesn't need to be manually adjusted each run?

```
def time_of_day():
    # Get the current hour
    hour = datetime.now().hour

    # Return approproiate description
    if hour >= 0 and hour < 6:
        return 'Night'
    elif hour >= 9 and hour < 12:
        return "Morning"
    elif  hour >= 12 and hour < 18:
        return "Afternoon"
    else:
        return "Evening"
```

What would your assert statement be?

assert time_of_day == ??????

What is the problem?

datetime.now()

is a hidden input that changes between test runs

We would have to somehow get the system time or change it before each test

Difficult to write	Slow	Requires environment to be set up (not a unit test)	Tightly coupled to data source – you cant pass it data from other sources	Breaks SRP – it gets data and processes it	Hides where it gets the data from	Hard to predict and maintain

Write your code so its testable

- Decouple the datetime and the processing
- Follow the Single Responsibility Principle (SRP)
- Your tests will now be much easier

```python
from datetime import datetime


class TestClass:
    # Difficult to automate test this as depends on time of day!
    def test_morning1(self):
        assert time_of_day() == "Night"

    # Far easier to test
    def test_morning2(self):
        assert time_of_day_2(9) == "Morning"
        assert time_of_day_2(13) == "Afternoon"
        assert time_of_day_2(0) == "Night"
        assert time_of_day_2(19) == "Evening"


def time_of_day_2(hour):

    # Return approproiate description
    if hour >= 0 and hour < 6:
        return 'Night'
    elif hour >= 9 and hour < 12:
        return "Morning"
    elif  hour >= 12 and hour < 18:
        return "Afternoon"
    else:
        return "Evening"
```

Testing the old function was unpredictable!

Testing is now easy

Hour is now an input, Function focusses only on returning the appropriate output

The tests sit alongside the function in a file

Write the tests FIRST

Run the tests against the function

```
(base) bash-3.2$ pytest pytest1.py
============================================= test session starts ==============================================
platform darwin -- Python 3.6.5, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: /Users/pughd/Documents/Code/aaa, inifile:
plugins: remotedata-0.2.1, openfiles-0.3.0, doctestplus-0.1.3, arraydiff-0.2
collected 4 items

pytest1.py .FF.                                                                                          [

=================================================== FAILURES ===================================================
_____________________________________________ TestClass.test_two ______________________________________________

self = <pytest1.TestClass object at 0x110e0f3c8>

    def test_two(self):
>       assert small_function(2,6) == 85
E       assert 8 == 85
E        +  where 8 = small_function(2, 6)

pytest1.py:25: AssertionError
_____________________________________________ TestClass.test_three ____________________________________________

self = <pytest1.TestClass object at 0x110e0f5f8>

    def test_three(self):
>       assert small_function(2,89) == 85
E       assert 91 == 85
E        +  where 91 = small_function(2, 89)

pytest1.py:28: AssertionError
========================================== 2 failed, 2 passed in 0.03 seconds ==========================================
```

Details of failures

Exercise

1. Get the Spec
   - Write a function that takes inputs, x and y
   - If x> y, return 0
   - If x<= y then return x^2 – 2*y
2. Write the tests
3. Write the code
4. Test the code

Example

```
# Create a class to hold our tests
class TestClass:

    def test_eq_100(self):
        assert small_function(100,5) == 100

    def test_gtr_100(self):
        assert small_function(102,6) == 100

    def test_lt_100(self):
        assert small_function(2,89) == 91


# Define our function
def small_function(x,y):
    if x< 100:
        return x + y
    else:
        return 100
```

Check for different inputs

The spec develops....

Spec change

- If x> y and x + y < 10, return 0
- If x> y and x + y >= 10, return 10
- If x<= y then return x^2 – m*y, where m is the number of entries in a database table
- Can take integers or strings

- Go through the process again

Example

```
# Test class to hold our tests
class TestClass:

    # test when x > y and x + y < 10
    def test_x_gtr_y_x_add_y_lt_10(self):
        assert example_function(5,1, 9) == 0

    # test when x > y and x + y > = 10
    def test_x_gtr_y_x_add_y_gte_10(self):
        assert example_function(10,9, 9) == 10

    #test that x<=y
    def test_x_lte_y(self):
        assert example_function(2,9,2) == -14

    #test string integers
    def test_str_ints(self):
        assert example_function("2","9", 2) == -14

    # test neg string integers
    def test_neg_str_ints(self):
        assert example_function("-2","-1", 2) == 6

    #test str inputs
    def test_str_word_entries(self):
        assert example_function("test","6", 4) == -24

def example_function(x,y,m):
    #idigit only detects positive so use cast and catch
   x = is_int(x)
   y = is_int(y)
   m = is_int(m)


   if x > y and x + y < 10:
        return 0
   elif x > y and x + y >=10 :
        return 10
   else:
        return (x*x) - (m*y)


def is_int(s):
   try:
        x = int(s)
        return x
   except ValueError:
        return 0
```

Check for input levels

Check with string inputs, incl negatives

Convert to int, Check to see if it is a number

Plan your testing early!

- Writing testable code requires some discipline, concentration, and extra effort
- Software development is a complex mental activity be careful and avoid recklessly throwing together new code
- Plan testing in early – before you code – so that you develop clean, easy-to-maintain, loosely coupled, and reusable APIs
- The ultimate advantage of testable code is not only the testability itself, but the ability to easily understand, maintain and extend that code as well

Go Try It!

See PyTest documentation for more details

github.com/David-Pugh/a-a-a/ for todays code and slides