Test Automation

- Getting started with testing
- Using PyHamcrest matchers
- Testing techniques

Getting started with testing

- Setting the scene
- Python test frameworks
- Example class-under-test
- How to write a test
- Example test
- Running tests
- Arrange / Act / Assert
- Testing for exceptions
- Setup and teardown code

Setting the scene

Unit testing verifies the correct behaviour of your code artefacts in isolation

- In Python, a "unit" is usually a function

You typically write several unit tests per function

- To exercise all the possible paths through the function

The FIRST principles of unit testing:

- Fast
- Isolated / independent
- Repeatable
- Self-validating
- Timely

Python test frameworks

There are several test frameworks available for Python

- Unittest (aka PyUnit) - part of standard library (OO-based)
- PyTest - simple and fast, most widely used (function-based)
- TestProject - generates HTML reports, use with PyTest or Unittest
- Behave - BDD test framework
- Robot - primarily for Acceptance Testing
- Etc.

We'll use PyTest

- Install as follows:

```
1   pip install pytest
```

Test installation as follows:

```
1   py.test -h
```

Example class-under-test

In the next few slides we'll see how to test this simple Python class

```
1   class BankAccount:
2
3       def __init__(self, name):
4           self.name = name
5           self.balance = 0;
6
7       def deposit(self, amount):
8           self.balance += amount
9
10      def withdraw(self, amount):
11          if amount > self.balance:
12              raise Exception("Insufficient funds")
13          self.balance -= amount
14
15      def __str__(self):
16          return "{}, {}".format(self.name, self.balance)
17
```

How to write a test

To write tests in PyTest:

- Define a separate .py file
- The filename must be test_xxx.py or xxx_test.py

Each test is a separate function

- The function name must be test_yyy()

Each test function should focus on a particular scenario, and should have a meaningful name

- E.g. test_functionName_Scenario
- E.g. test_functionName_StateUnderTest_ExpectedBehaviour

Example test

When writing your tests, go for the low-hanging fruit first

- Test the simplest functions and scenarios first
- Then test the more complex functions and scenarios later

Here's a simple first test

```
1   from bankAccount import BankAccount
2
3   def test_accountCreated_zeroBalanceInitially():
4       fixture = BankAccount("David")
5       assert fixture.balance == 0
```

Notes:

- assert is a standard Python keyword
- If the test returns false, it throws an AssertionError
- This causes your test function to terminate immediately

Running tests

To run tests in PyTest, run the following command:

```
py.test
```

Arrange / Act / Assert

It's common for a test function to have 3 parts

- Arrange
- Act
- Assert

Example

```
1   def test_deposit_singleDeposit_correctBalance():
2
3       # Arrange.
4       fixture = BankAccount("David")
5
6       # Act.
7       fixture.deposit(100)
8
9       # Assert.
10      assert fixture.balance == 100
```

Testing for exceptions (1/2)

When you write industrial-strength code, sometimes you actually want the code to throw an exception

- The code should be robust enough to detect exceptional situations
- You can write a test to verify the code throws an exception

```
1   import pytest
2   …
3
4   def test_deposit_withdrawalsExceedLimits_exceptionOccursV1():
5
6       # Arrange.
7       fixture = BankAccount("David")
8
9       # Act.
10      fixture.deposit(600)
11
12      # Verify expected exception occurs
13      with pytest.raises(Exception):
14          fixture.withdraw(601)
15
```

Testing for exceptions (2/2)

If you want to examine the exception that was thrown:

```
import pytest
…

def test_deposit_withdrawalsExceedLimits_exceptionOccursV2():

    # Arrange.
    fixture = BankAccount("David")

    # Act.
    fixture.deposit(600)

    # Verify expected exception occurs
    with pytest.raises(Exception) as excinfo:
        fixture.withdraw(601)

    # Assert the exception type and error message are correct
    assert excinfo.typename == "Exception"
    assert excinfo.value.args[0] == "Insufficient funds"

```

Setup and teardown code

Sometimes you have common setup/teardown tasks that you want to perform before/after each test

- You can define a "fixture function" to do the before/after code

```
import pytest
…

acc = None

@pytest.fixture(autouse=True)
def run_around_tests():
    # Code that will run before each test.
    print("Do something before a test")
    global acc
    acc = BankAccount("David")

    # A test function will be run at this point
    yield

    # Code that will run after each test.
    print("Do something after a test")
```

Note: To see console output with PyTest, use the -s option

Using PyHamcrest matchers

- A reminder about simple assertions
- Introducing PyHamcrest
- Getting started with PyHamcrest
- Example class-under-test
- Example test
- Defining a custom PyHamcrest matcher
- Using a custom PyHamcrest matcher

A reminder about simple assertions

As we've seen, Python has a simple assert keyword

- assert a == b

What if you want to write some specific tests, such as:

- Does a collection contains a value?
- Does a variable point to a particular type of subclass?
- Does an integer value lie in a certain range?
- Does a double value equal a variable, to a specified accuracy?

Introducing PyHamcrest

The PyHamcrest library provides a higher-level vocabulary for writing your tests, with "matcher" functions such as:

- equal_to, close_to
- not
- greater_than, greater_than_or_equal_to
- less_than, less_than_or_equal_to
- starts_with, ends_with, contains_string, is_empty
- all_of, any_of
- contains_string, contains_exactly, contains_in_any_order
- has_item, has_items
- has_entry, has_entries, has_key, has_keys, has_value, has_values
- instance_of
- … etc.

Getting started with PyHamcrest

Install PyHamcrest as follows:

```
pip install PyHamcrest
```

You can then use PyHamcrest as follows in your tests:

```
from hamcrest import *
…

assert_that(… … …)
```

Example class-under-test

To illustrate PyHamcrest, we'll test the following class:

```
class Product:

    def __init__(self, description, price, *ratings):
        self.description = description
        self.price = price
        self.ratings = list(ratings)

    def taxPayable(self):
        return self.price * 0.20

    def __str__(self):
       return "{}, £{}, {}".format(self.description, self.price, self.ratings)

```

Example test

```
1  from hamcrest import *
2  import pytest
3  from Product import Product
4
5  product = None
6
7  @pytest.fixture(autouse=True)
8  def run_around_tests():
9      global product
10     product = Product("TV", 1500, 5, 4, 3, 5, 4, 3)
11     yield
12
13 def test_product_taxPayable_correct():
14     assert_that(product.taxPayable(), close_to(300, 0.1))
15
16 def test_product_ratings_containsRating():
17     assert_that(product.ratings, has_item(3))
18
19 def test_product_ratings_doesntContainsAbsentRating():
20     assert_that(product.ratings, not(has_item(2)))
```

Defining a custom PyHamcrest matcher

The PyHamcrest "matcher" model is extensible

- You can define your own custom matcher classes

```
from hamcrest.core.base_matcher import BaseMatcher


class PriceMatcher(BaseMatcher):

    def __init__(self, maxInclusive=3_000):
        self.maxInclusive = maxInclusive

    def _matches(self, price):
        return 0 < price ⩽ self.maxInclusive

    def describe_to(self, description):
        description.append_text("0 ... " + str(self.maxInclusive))


def valid_price():
    return PriceMatcher(2_500)
```

Using a custom PyHamcrest matcher

Here's how to use a custom PyHamcrest matcher

- Exactly the same as for the standard PyHamcrest matchers ☺

```
1   from hamcrest import *
2   from priceMatcher import valid_price
3   from product import Product
4
5
6   def test_product_validPrice_priceAccepted():
7       product1 = Product("TV", 1500)
8       assert_that(product1.price, valid_price())
9
10
11  def test_product_negativePrice_priceRejected():
12      product2 = Product("TV", -1)
13      assert_that(product2.price, is_not(valid_price()))
14
15
16  def test_product_tooExpensivePrice_priceRejected():
17      product3 = Product("TV", 2501)
18      assert_that(product3.price, is_not(valid_price()))
```

Testing techniques

- Parameterized tests
- Running tests selectively
- Grouping tests into sets
- Test Driven Development (TDD)
- Refactoring

Parameterized tests (1/2)

When you start writing tests, you might notice some of the tests are quite similar and repetitive

- E.g. imagine a function that returns the grade for an exam
- How would you test it always returns the correct grade?

```
def get_grade(mark):
    if mark ⩾ 75:
        return "A*"
    elif mark ⩾ 70:
        return "A"
    elif mark ⩾ 60:
        return "B"
    elif mark ⩾ 50:
        return "C"
    elif mark ⩾ 40:
        return "D"
    elif mark ⩾ 30:
        return "E"
    else:
        return "U"
```

Parameterized tests (2/2)

You can write a parameterized test as follows:

```
import pytest
from util import get_grade

@pytest.mark.parametrize("mark,grade", [
    (99, "A*"),
    (70, "A"),
    (69, "B"),
    (60, "B"),
    (59, "C"),
    (50, "C"),
    (49, "D"),
    (40, "D"),
    (39, "E"),
    (30, "E"),
    (29, "U")])
def test_marks_and_grades(mark, grade):
    assert grade == get_grade(mark)
```

Running tests selectively

test.py lets you specify which test functions to run…

E.g. run all test functions that have 'deposit' in their name

- The -k option specifies the key (function name fragment)
- The -v option displays verbose test results

```
1   py.test -k deposit -v
```

Grouping tests into sets (1/3)

You can group tests into sets

- You can then run all the tests in a particular set

The first step is to specify your custom sets

- Define a file named pytest.ini as follows:

```
[pytest]
markers =
    numtest: mark a test as a numeric test
    strtest: mark a test as a string test
```

pytest.ini

Grouping tests into sets (2/3)

You can then mark a test function so it belongs to a set(s)

- Decorate the test function with @pytest.mark.aSetName

```
1
2   import pytest
3
4   @pytest.mark.numtest
5   def test_add_numbers():
6       assert 3 + 4 == 7
7
8   @pytest.mark.numtest
9   def test_multiple_numbers():
10      assert 3 * 4 == 12
11
12  @pytest.mark.strtest
13  def test_concatenate_strings():
14      assert "hello " + "world" == "hello world"
15
16  @pytest.mark.strtest
17  def test_uppercase_strings():
18      assert "hello world".upper() == "HELLO WORLD"
```

Grouping tests into sets (3/3)

To run all the tests in a particular set:

- Use the -m option
- Specifies which marked tests to run (i.e. the name of the set)

```
1   py.test -m numtest -v
```

Test Driven Development (TDD)

TDD is a simple concept

- You write the tests first, before your write the code
- The tests act as a specification for the new functionality you're about to implement

In TDD, you perform the following tasks repeatedly:

- Write a test
- Run the test - it must fail!
- Write the minimum amount of code, to make the test pass
- Refactor your code

Benefits of TDD

- Helps you focus on functionality rather than implementation
- Ensures every line of code is tested

Refactoring

Refactoring is an oft-overlooked aspect of TDD

- After each iteration through the test-code-pass cycle, you should refactor your code
- That is, step back and see if you can/should reorganize your code to eliminate duplication, restructure inheritance, etc.

Typical refactoring activities:

- Rename a variable / function / class / module
- Extract duplicate code into a common function
- Extract common class functionality into a superclass
- Introduce another level of inheritance