Python Testing

UBCO Master of Data Science – DATA 533

Today's Class

Value of testing and different testing techniques

Levels of Software Testing

Available testing tools

`unittest` — a Unit testing framework

Test classes, test cases, test suites using `unittest`

What is Testing?

A software is only useful if they do what they are supposed to do.

Testing, in general programming terms, is the practice of writing code to help determine if there are any errors.

When you a developer

- your goal is to make the program work

When you are a tester

- you want to make it fail

Why Is Testing Important?

People make mistakes

- If they left unchecked, some of them could lead to failures
- Can be very expensive
  - Example: Ariane 5, 37 sec after launch. Cost: $1 billion

Testing code helps to catch these mistakes

Testing is an integral part of any successful software project.

Tests help to identify errors, ensure the quality of the product and to verify that the software does what it is meant to do.

Automated vs. Manual Testing

Manual testing:

- Test cases are executed manually (by a human, that is) without any support from tools or scripts.
- A user is testing a login feature in an Android app by typing an email and password, and tapping Log in button to proceed — then observe whether it logged in successfully.
- Example: Usability testing

Automated testing:

- Test cases are executed with the assistance of tools, scripts, and software.
- Writing code to create test scripts that could automatically do the above actions for you and give you a report of any error
- Example: Performance testing

Automated vs. Manual Testing Question

Question: How many of the following statements are TRUE?

1) Manual testing of an application is performed manually by a human

2) In automated testing, a tester evaluates the design, functionality, and performance of the application by clicking through various elements.

3) In automated testing, there are pre-scripted tests which run automatically

4) Automated testing is suitable when the test cases need to run repeatedly for a long duration of time.

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

Levels of Software Testing

There are several fundamental levels within software testing, each examining the software functionality from a unique vantage point:

- Unit testing
  - consist in testing individual methods of classes, components or modules used by your software
- Integration Testing
  - verify that different modules or services used by your application work well together
- Functional or End-to-End Testing
  - verify the output of an action and do not check the intermediate states of the system when performing that action
- …

Unit Testing

The foundational level of software testing is unit testing.

Unit testing specifically tests a single ***unit*** of code in isolation.

A unit is often a function or a method of a class instance.

```
def addition(num1, num2):
    return num1 + num2
```

} One Unit

```
def subtraction(num1, num2):
    return num1 - num2
```

} Another Unit

Unit Testing Tools

`unittest`

- Built-in standard library tool for testing Python code.

`pytest`

- A complete testing tool that emphasizes backward-compatibility and minimizing boilerplate code.

`nose`

- An extension to unittest that makes it easier to create and execute test cases.

`Hypothesis`

- Is a unit test-generation tool that assists developers in creating tests that exercise edge cases in code blocks.

`unittest` — Unit testing framework

`unittest` is part of the Python standard library

It supports test automation, sharing of setup and shutdown code for tests

A unit test consists of one or more assertions

- If the assert condition is true, it does nothing and a program continues to execute.
- If the assert condition is false, it raises an AssertionError exception with an optional error message

Calculation Module

The module contains two functions/methods:

- One for adding two numbers
- Another for subtracting two numbers

```
# calculator.py
def addition(num1, num2):
    return num1 + num2

def subtraction(num1, num2):
    return num1 - num2
```

`unittest` - Steps

Import `unittest: import unittest`

Import modules to test: `import calculator`

Create a test class by inheriting `unittest.TestCase`

```
class TestCalculator(unittest.TestCase):
```

The class has several methods used to design tests: `setup, teardown, setUpClass, tearDownClass` and so on

Test function naming convention: `test_[python module]`

Any method which starts with `test_` is considered as a test case.

`Assert` Methods

The TestCase class provides several assert methods to check for and report failures.

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)`

https://docs.python.org/3/library/unittest.html#unittest.TestCase

Example

```
import unittest                                    # Jupyter notebook
import calculator as cal
class TestCalculator(unittest.TestCase): # test class
    def test_addition(self):     # test case
        self.assertEqual(cal.addition(10, 10), 20)
        self.assertEqual(cal.addition(-10, 10), 0)
        self.assertEqual(cal.addition(10, -10), 0)
    def test_subtraction(self): # test case
        self.assertEqual(cal.subtraction(10, 10), 0)
        self.assertEqual(cal.subtraction(-10, 10), -20)
        self.assertEqual(cal.subtraction(-10, -10), 0)
unittest.main(argv=[''], verbosity=2, exit=False)
```

Correct

Correct

Output

```
test_addition (__main__.TestCalculator) ... ok
test_subtraction (__main__.TestCalculator) ... ok
----------------------------------------------------------------------
Ran 2 tests in 0.025s
OK
```

There are different possible test outcomes :

`OK` – all the tests are passed.

`FAIL` – the test did not pass, an `AssertionError` exception is raised.

Example

```
def addition(num1, num2):                    # Jupyter notebook
    return num1 * num2

def subtraction(num1, num2):
    return num1 - num2
```

Output

```
test_addition (__main__.TestCalculator) ... FAIL
test_subtraction (__main__.TestCalculator) ... ok

======================================================================
FAIL: test_addition (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-1-e9985e501e61>", line 5, in test_addition
    self.assertEqual(cal.addition(10, 10), 20)
AssertionError: 100 != 20

----------------------------------------------------------------------
Ran 2 tests in 0.009s

FAILED (failures=1)
```

Unit Testing Question

Question: How many of the following statements are TRUE?

1) `pytest` emphasizes minimizing boilerplate code.

2) A unit is often a method of a class instance.

3) `Assert` methods are used to check for and report failures.

4) Any method which starts with `test_` is considered as a test case.

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

Try it: Unit Testing

Question:

1. Write two functions called `multiplication` and `division` to perform manipulation and division operations on two numbers (e.g., `num1` and `num2`)

2. Write test cases to check both functions.

3. Change the `multiplication` operator from `*` to `**` and check the output.

4. Change the `division` operator from `/` to `//` and check the output.

Another Example: `Person Class`

```
# person.py
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def set_name(self, name):
        self.name = name

    def set_age(self, age):
        self.age = age

    def display(self):
        return '{} {}'.format(self.name, self.age)
```

Person Class:test_set_name

```
# Jupyter notebook
import unittest
from person import Person
class TestPerson(unittest.TestCase):
    def test_set_name(self):
        p1 = Person('Alex',20)
        p2 = Person('William',25)
        p1.set_name('Ana')
        p2.set_name('Dave')

        self.assertEqual(p1.name,'Ana')
        self.assertEqual(p2.name,'Dave')
unittest.main(argv=[''], verbosity=2, exit=False)
```

```
# person.py
def set_name(self, name):
        self.name = name
```

Person Class:test_set_age

```
def test_set_age(self):
    p1 = Person('Alex',20)
    p2 = Person('William',25)
    p1.set_age(22)
    p2.set_age(25)

    self.assertEqual(p1.age,22)
    self.assertEqual(p2.age,25)
```

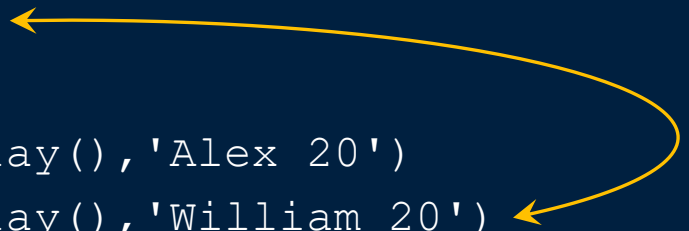
```
# person.py
def set_age(self, age):
        self.age = age
```

Person Class:test_display

```
# person.py
def display(self):
        return '{} {}'.format(self.name, self.age)

# Jupyter notebook
……
def test_display(self):
        p1 = Person('Alex',20)
        p2 = Person('William',25)

        self.assertEqual(p1.display(),'Alex 20')
        self.assertEqual(p2.display(),'William 25')
```

Person Class: Test Failure

```
def test_display(self):
        p1 = Person('Alex',20)
        p2 = Person('William',25)

        self.assertEqual(p1.display(),'Alex 20')
        self.assertEqual(p2.display(),'William 20')
```



```
test_display (__main__.TestPerson) ... FAIL
AssertionError: 'William 25' != 'William 20'
- William 25
?          ^
+ William 20
?          ^
```

A Better Approach

In the previous example, each test case loads p1 and p2 to ensure a fresh dataset

`TestCase` instances provide a better way to handle this:

`setup()` method

- is called immediately before calling the test method

`tearDown()` method

- called immediately after the test method has been called and the result recorded.

A Better Approach

```
def setUp(self): # Setting up for the test
    self.p1 = Person('Alex',20)
    self.p2 = Person('William',25)

def test_set_name(self): # test routine
    self.p1.set_name('Ana')
    self.p2.set_name('Dave')
    self.assertEqual(self.p1.name,'Ana')
    self.assertEqual(self.p2.name,'Dave')

def test_set_age(self): # test routine
    self.p1.set_age(22)
    self.p2.set_age(25)
    self.assertEqual(self.p1.age,22)
    self.assertEqual(self.p2.age,25)
```

Note: The setUp() method
fires before each test routine

A Better Approach

If you add `print` statements after each test case, you will see that the `setup()` and `teardown()` is called for every single test

```
def setUp(self):
    print('Set up')
def tearDown(self):
    print('Tear Down')
def test_set_name(self):
    print('Name')
def test_set_age(self):
    print('Age')
def test_display(self):
    print('Display')
```

Output:

```
Set up
Name
Tear Down

Set up
Age
Tear Down

Set up
Display
Tear Down
```

Calling a Method Before and After Tests

A class method called before tests in an individual class run.

```
@classmethod
def setUpClass(cls):
    print('setupClass')
```

A class method called after tests in an individual class have run

```
@classmethod
def tearDownClass(cls):
    print('teardownClass')
```

Unit Testing Question

Question: What is the output of the following program?

```
# in notebook
import unittest
from factorial import fact

class TestFactorial(unittest.TestCase):
    def test_fact(self):
        self.assertEqual(fact(0), 1)

unittest.main(argv=[''], verbosity=2, exit=False)
```

```
# factorial.py
def fact(n):
    if n == 0:
        return 1
    return n * fact(n -1)
```

A) OK **B)** FAIL **C)** Syntax Error **D)** None of the above

Unit Testing Question

Question: How many times `setup` will be called for the following code?

```
def setUp(self):
    print('Set up')
def test_A(self):
    print('A')
def test_B(self):
    print('B')
def test_C(self):
    print('C')
def test_D(self):
    print('D')
```

A) 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

Try it: Unit Testing

Question:

1. Write a module that has a function to check if two given strings are Anagram

2. Create a test class in Jupyter Notebook to test the function.

3. Add `setup()` and `tearDown()` methods and see how they change the output.

`unittest` frameworks

The unittest frameworks supports the following concepts:

Test Fixture – A fixture is what is used to setup a test so it can be run and also tears down when the test is finished.

Test Case – The test case is your actual test. It will typically check (or assert) that a specific response comes from a specific set of inputs.

Test Suite – The test suite is a collection of test cases, test suites or both.

Test Runner – A runner is what controls the running of the tests or suites. It will also provide the outcome to the user (i.e. did they pass or fail).

Creating Test Suites

A test suite is just a collection of test cases, test suites or both.

Test suites are implemented by the `TestSuite` class

This class allows individual tests and test suites to be aggregated

When the suite is executed, all tests added directly to the suite and in "child" test suites are run

Creating Test Suites

Steps to create a test suite:

- Create an instance of `TestSuite`.
- Create an instance of `TestResult`. The `TestResult` class just holds the results of the tests.
- Call `addTest` on your suite object.
- The last step is to run the suit.

Test Suite Example

```
# mod1.py
def addition(num1, num2):
    return num1 + num2


# TestModule1.py
import unittest
from mod1 import addition as md1

class TestAdd(unittest.TestCase):

    def test_addition(self):
        self.assertEqual(md1(1,1),2)
        self.assertEqual(md1(1,2),3)
```

Test Suite Example

```
# mod2.py
def subtraction(num1, num2):
    return num1 - num2


# TestModule2.py
import unittest
from mod2 import subtraction as md2

class TestSub(unittest.TestCase):

    def test_subtraction(self):
        self.assertEqual(md2(1,1),0)
        self.assertEqual(md2(1,2),-1)
```

Test Suite Example

```
import unittest
from TestModule1 import TestAdd
from TestModule2 import TestSub
def my_suite():
    suite = unittest.TestSuite()
    result = unittest.TestResult()
    suite.addTest(TestAdd('test_addition'))
    suite.addTest(TestSub('test_subtraction'))
    runner = unittest.TextTestRunner()
    print(runner.run(suite))
my_suite()
```

Test Driven Development (TDD)

Test-first development approach

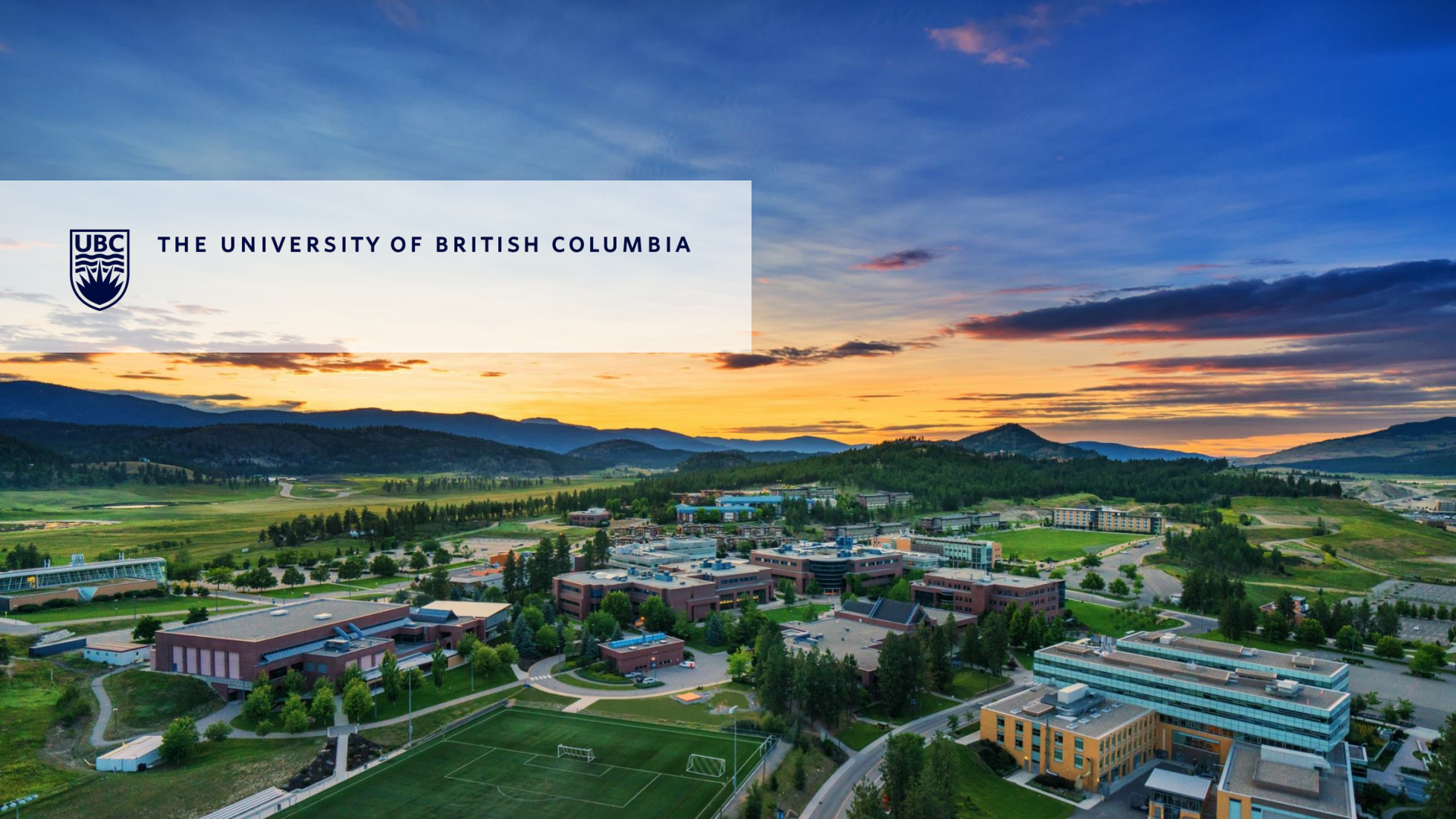
First a developer writes test cases

Then produces the minimum amount of code to pass that test

Finally refactors the new code to acceptable standards

Objectives

- Understand the value of testing and different testing techniques
- Learn about Unit testing
- Learn about available testing tools for Unit testing
- Use `unittest` — a Unit testing framework
- Be able to write test classes and test cases using `unittest`
- Be able to design a test suite


UBC
THE UNIVERSITY OF BRITISH COLUMBIA