UVA | SCHOOL *of* DATA SCIENCE

Module 05 Testing And Debugging

WHEN ARE YOU READY TO TEST?

ensure **code runs**

- remove syntax errors
- remove static semantic errors
- Python interpreter can usually find these for you

have a **set of expected results**

- an input set
- for each input, the expected output

CLASSES OF TESTS

Unit testing

- validate each piece of program
- **testing each function** separately

Regression testing

- add test for bugs as you find them
- **catch reintroduced** errors that were previously fixed

Integration testing

- does **overall program** work?
- tend to rush to do this

TESTING APPROACHES

Intuition about natural boundaries to the problem

if no natural partitions, might do **random testing**

- probability that code is correct increases with more tests
- better options below

black box testing

- explore paths through specification

glass box testing

- explore paths through code

BLACK BOX TESTING

```
def sqrt(x, eps):
""" Assumes x, eps floats, x >= 0, eps > 0
Returns res such that x-eps <= res*res <= x+eps"""
```

designed **without looking** at the code

can be done by someone other than the implementer to avoid some implementer **biases**

testing can be **reused** if implementation changes

Paths through specification

- build test cases in different natural space partitions
- also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

BLACK BOX TESTING

```
def sqrt(x, eps):
""" Assumes x, eps floats, x >= 0, eps > 0
Returns res such that x-eps <= res*res <= x+eps"""
```

Case	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

GLASS BOX TESTING

use code directly to guide design of test cases

called **path-complete** if every potential path through code is tested at least once

what are some **drawbacks** of this type of testing?

- can go through loops arbitrarily many times
- missing paths

guidelines

- branches
- for loops
- while loops

GLASS BOX TESTING

```
def abs(x):
""" Assumes x is an int
Returns x if x>=0 and -x otherwise """
  if x < -1:
    return -x
  else:
    return x
```

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test boundary cases

UVA | SCHOOL *of* DATA SCIENCE

Unit Testing

- Developers can work in a predictable way of developing code
- Programmers write their own unit tests
- Get rapid response for testing small changes
- Encourages programmers to build many **highly-cohesive loosely- coupled modules** to make unit testing easier

① Assert statement

```
# pyScript21_B.py

class AStudent:
  def __init__ (self, name, id, grades=None):
    self.name = name
    self.id = id
    if grades is None:
      grades = []
    self.grades = grades

  def addGrade (self, grade):
    self.grades.append(grade)
```

We will test this function

Google Colab

```
from pyScript21_B import *

student1 = AStudent('Clayton', '5010', 0)
student1.grades = []
student1.addGrade(90)

assert student1.grades == [80]  # AssertionError Error
assert student1.grades == [90]  # Correct
```

① Assert statement

The `assert` keyword is used when debugging code.

The `assert` keyword lets you test if a condition in your code returns True, if not, the program will raise an **AssertionError**.

Writing tests in this way is okay for a simple check, but what if more than one fails?
Test runner is a special application designed for running tests, checking the output, and giving you tools for debugging and diagnosing tests and applications.

Test Runner

There are many test runners available for Python. The one built into the Python standard library is called unittest. The three most popular test runners are:

- unittest
- nose or nose2
- pytest

unittest requires that:

- You put your tests into classes as methods
- You use a series of special assertion methods in the unittest.TestCase class instead of the built-in assert statement

② unittest basic structure

```
import unittest
from pyScript21_B import *

class AddGradeTest(unittest.TestCase):

  def test_1(self):
    student1 = AStudent('Clayton', '5010', 90)
    self.assertEqual(student1.grades,90)
  def test_2(self):
    student1 = AStudent('Clayton', '5010', 100)
    self.assertEqual(student1.grades,90)

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

Google Colab

Import `unittest`

Create a `class` that inherits from the `TestCase` class

Convert the test functions into methods

Change the assertions to use the `self.assertEqual` method on the TestCase class

Change the command-line entry point to call **unittest.main()** (use `unittest.main(argv=[''], verbosity=2)` in **google colab**)

② Understanding Test Output

```
.F
==========================================
FAIL: test_2 (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\admin\Desktop\t\test_sum.py", line 10, in
test_sum2
    self.assertEqual(student1.grades, 90)
AssertionError: 100 != 90


----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

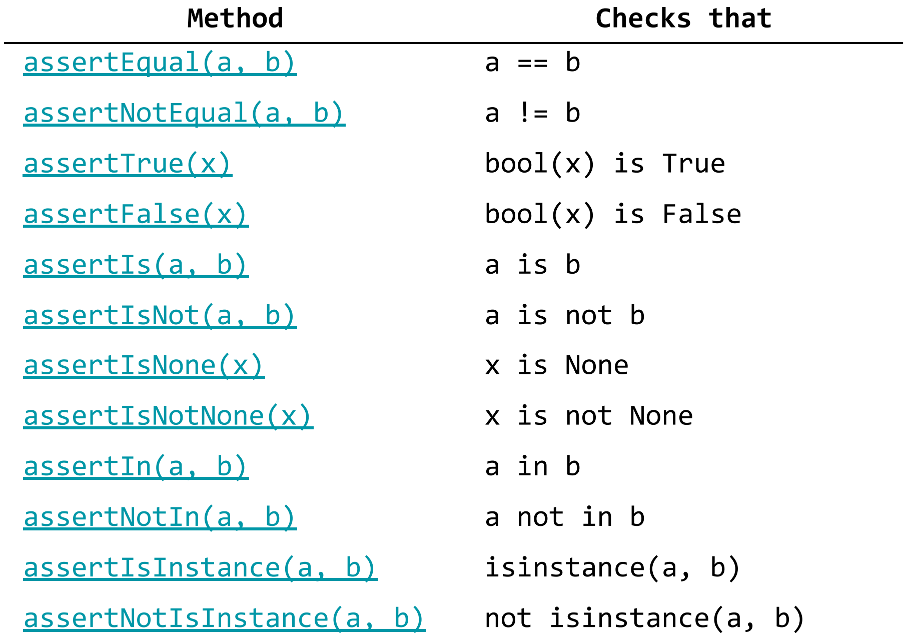
The first line shows the execution results of all the tests, one failed (F) and one passed (.)

The FAIL entry shows some details about the failed test:

- The test module and the test case
- A traceback to the failing line
- The details of the assertion with the expected result and the actual result

② assert methods in unittest.TestCase

`unittest` comes with lots of methods to assert on the values, types, and existence of variables. Here are some of the most commonly used methods:



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)`

All the assert methods (except **assertRaises**(), **assertRaisesRegexp**()) accept a **msg** argument that, if specified, is used as the error message on failure

Parameterised testing

When there are very small differences among your tests, for instance some parameters, `unittest` allows you to distinguish them inside the body of a test method using the `subTest()` context manager.

③ Parameterised testing

When multiple tests are required ❌

Define a method separately for each test

```
class AddGradeTest(TestCase):
  def test_1(self):
    self.assertEqual(student1.grades,1)
  def test_2(self):
    self.assertEqual(student1.grades,2)
  def test_3(self):
    self.assertEqual(student1.grades,3)
…
…
```

A huge amount of repetition, and a maintenance headache whenever things change

Use loop: ❌

```
class GradeTest(TestCase):
  def test_1(self):
    for i in range(1000):
      self.assertEqual(student1.grades, i )
```

Execution would stop after the first failure

③ Parameterised testing using subTest()

i = 0 pass
i = 1 fail
i = 2 fail

```
import unittest
from pyScript21_B import *

grades=[90,80,100]
class AddGradeTest(unittest.TestCase):
  def test_1(self):
    for i in range (0,3):
      with self.subTest(i=i):
        student1 = AStudent('Clayton', '5010', 90)
        self.assertEqual(student1.grades,grades[i])

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

```
FAIL: test_1 (__main__.AddGradeTestCase) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\admin\Desktop\t\test_subtest.py", line 11, in test_sum
    self.assertEqual(student1.grades,grades[i])
AssertionError: 90 != 80
======================================================================
FAIL: test_1 (__main__.AddGradeTestCase) (i=2)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\admin\Desktop\t\test_subtest.py", line 11, in test_sum
    self.assertEqual(student1.grades,grades[i])
AssertionError: 90 != 100
```

Without using a subtest, execution would **stop after the first failure**, and the error would be less easy to diagnose because the value of i wouldn't be displayed

Test Fixtures

Test **fixtures** are methods and functions that run before and after a test.

The intent is to provide developers hooks to set up preconditions needed for the test, and cleanup after the test.

④ Software Test Fixtures

Test **fixtures** are methods and functions that run before and after a test.

The intent is to provide developers hooks to set up preconditions needed for the test, and cleanup after the test.

The most common fixture methods are `setUp` and `tearDown`.

The `setUp()` method runs before every test.
The `tearDown()` method runs after every test.

④ Software Test Fixtures

```
import unittest
from pyScript21_B import *

student1 = AStudent('Clayton', '5010', [])

class AddGradeTest(unittest.TestCase):

  def setUp(self):
    student1.addGrade(90)

  def test_1(self):
    self.assertEqual(student1.grades,[90,90])

  def test_2(self):
    self.assertEqual(student1.grades,[90,90])

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

# addGrade() before each testing

# test_1 fail ;test_2 pass

```
F.
======================================================================
=
FAIL: test_1 (__main__.AddGradeTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\admin\Desktop\t\test_fixture.py", line 12, in test_1
    self.assertEqual(student1.grades,[90,90])
AssertionError: Lists differ: [90] != [90, 90]

Second list contains 1 additional elements.
First extra element 1:
90

- [90]
+ [90, 90]

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Log Unittest output to a text file

```
import unittest
# Testing Code
# ...

if __name__ == '__main__':
   log_file = 'log_file.txt'
   with open(log_file, "w") as f:
       runner = unittest.TextTestRunner(f)
       unittest.main(testRunner=runner)
```

Review

- Assert statement
- Unittest library
- Parameterised testing
- Test Fixtures
- Log Unittest output to a text file

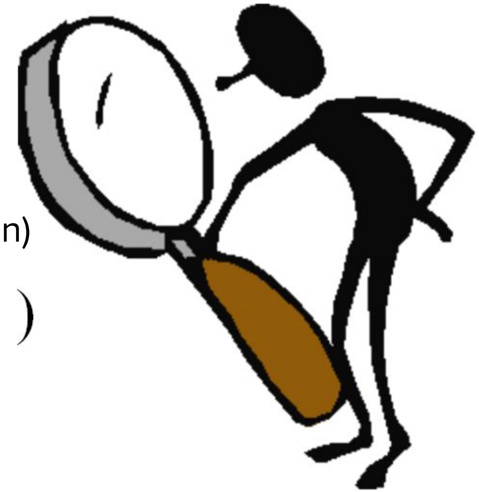
UVA | SCHOOL *of* DATA SCIENCE

Design of Unit Testing

Test Coverage

- How do you know that your set of tests really exercise all of your code?

  Term: **code coverage**
- You could (of course) have at least one test-method per method
  - Not sufficient!
  - if/else statements, loops, catch blocks, etc.
- Coverage tool
  - Runs your code, gives a report of what is covered
  - Example tools: Clover (used by Web-CAT), Cobertura (Eclipse plug-in)



Coverage:

- % of lines covered
- % of flow paths covered (often 0%: due to infinite options)
- % of input sequences covered (often 0%)

Unit Testing Rule of Thumb

- If your test can fail in more than one way, make a separate test for each of those ways!
- Do not mix many tests in one unit test – better to separate them
- Useful to use the optional “msg” (message) parameter to clearly describe what when wrong when running tests that failed

TDD – Unit Testing

- **Test-first, Test-driven design**
  - Write a “stub” (just the signature and dummy return value)
  - Write many test cases
  - Implement the method until the test cases pass
- **Three benefits**:
  1. Less temptation to skip on testing, forces you to be meticulous
  2. More likely to think about every part of spec in coding
  3. Less likely to test only what works

Principles of Unit Test Design

- Input space, output space, and internal space (i.e. Input domain)

- **Input**: the set of possible arguments, input streams, and events
  - Argument: "did we test Math.sin(float('Infinitiy'))?"
  - Input: "did we test when the file is empty?"
  - Events: "did we test when the user double-clicks here?"

- **Output**: the set of possible returns, output streams, and side effects
  - Returns: "did we test where the answer was zero (0)?
  - Output streams: "did we test where it prints out this message?"
  - Side effects: "did we test where it is supposed to open a new window?"
- **Internal**: the input/output spaces of the operation used inside the method

[Most people focus on input space]

Principles of Unit Test Design

For each space identify **equivalence classes**

- Goal: either all or none in class work
- For each equivalence class
  - Test the boundaries of the class
  - Test at least one thing in the middle of the class
- Identify corner cases
  - Exceptions – including “no input left”, “website unavailable”, etc
  - Weird behaviors – e.g. Math.atan2: “if the first argument is negative zero and the second argument is positive then the result is negative zero.” -0.0?

Principles of Unit Test Design

Remember the one-off inputs:

- 0 (zero)
- null
- “” (empty string)
- Integer.MAX_VALUE
- Float(‘-Infinity’)
- Etc...