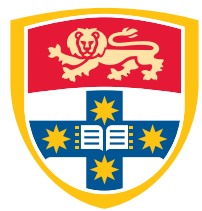
# INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



# Copyright Warning

# Lecture 19: More testing

*Types of testing, what needs testing, assertions, unit tests*

# Precondition and postcondition

Preconditions and postconditions are *assertions* placed before and after some piece of code

```
1 # Pre : x >= 0
2 y = math.sqrt(x)
3 # Post : y >= 0
```

The precondition asserts that x is non-negative at that point, which is needed to ensure that the sqrt function will work properly;

The postcondition asserts that y must equal $\sqrt{x}$ afterwards, which is also true since otherwise something must be wrong with the *sqrt* function.

These communicate the idea that: this code will *always* produce a true postcondition if the precondition is true

# Testing a returned `String`

In many cases, e.g., “Hello World”, you return a `String` object. It’s easy to test whether a function returns the right string: we can just use compare the output string with what expect:

```
1  def greet():
2        return "Hello, World!"
3
4  def test_greet():
5        if greet() == "Hello, World!":
6           return True
7        return False
```

which is a little clunky but does the job. Here we are testing to see whether

the answer is correct and returning `True` if (and only if) it is.

# Testing a returned `String` (cont.)

clean this up!

```
1  def greet():
2      return "Hello, World!"
3
4  def test_greet():
5      # actual running of real code, storing the "output"
6      output = greet()
7
8      # what we expected running of real code
9      expected = "Hello, World!"
10
11     # figure out how to measure output vs expected
12     # this could be very complicated, or very simple
13     passed = (output == expected)
14
15     # final outcome
16     if passed:
17         return True
18     return False
```

# Testing a returned `String` (cont.)

Another way is to throw an exception if the test is not passed.

```
    ...
    # final outcome
    if not passed:
        raise SomeSpecificException("Greet is not working!")
```

An Error is like an Exception in that it is throwable and it is handled and dealt with in a similar way

```
import traceback
try:
    test_greet()
except SomeSpecificException as e:
    print(e)
    traceback.print_exc()
```

# Testing a returned `String` (cont.)

Use assertion as the exception

```
    ...
    # final outcome
    assert passed
```

```
import traceback
try:
    test_greet()
except AssertionError as e:
    print(e)
    traceback.print_exc()
```

# Loop invariant

An *assertion of the loop*

```
while condition1:
    ...
    if condition2:
        break
    ...
    if condition3:
        break
    ...
```

It is most useful with more complex loops, where you have identified conditions that should always be true.

```
import random
index = random.randrange(100,500)
# loop invariant is index > 0. That is, at any time during the
    loop, index is guaranteed to be positive.
while index > 0:
    print(index)
    index -= random.randrange(0,5)
```

# Loop invariant (cont.)

```
1  import random
2  index = random.randrange(100,500)
3  # loop invariant is index > 0 for statements "print" only. After
       these statements, it is possible for index <= 0.
4  while index > 0:
5      print(index)
6      index -= random.randrange(0,5)
7      if index < 0:
8          index += 1
```

In the above case, you can start to define boundaries of what values are expected at any time during the loop. They can be very complex!

# Class invariant

An *assertion of the class*

Height of person: `size > 0`

Matryoshka doll: `size > 0`

Can test a class invariant anywhere[1] in the class and it is assumed to be true.

Most useful for identifying where a problem is, both from external or internal modifications of the object data

Not as helpful if instance variables are public: values of the object can change without using methods, from anywhere in the program at anytime

[1] except in constructor before values are initialised

# Precondition and postcondition with objects

Recall: Preconditions and postconditions are *assertions* placed before and after some piece of code

The conditions can be tied to the state of an object.

# Precondition and postcondition with objects (cont.)

```
1  class Location:
2      # pre: none
3      # post: new point with default coords and name length >= 1
4      def __init__(self, name):
5          reset()
6          if not len(name) >= 1:
7              raise ValueError("name provided is too short")
8          self.name = name
9
10     # pre: none
11     # post: coords always set to x,y
12     def set(self, x, y):
13         self.x = x
14         self.y = y
15
16     # pre: none
17     # post: coords always set to zero
18     def reset(self):
19         self.x = 0
20         self.y = 0
21
```

# Precondition and postcondition with objects (cont.)

```
22      # pre: name length > 0
23      # post: name length > 0, name contains no vowels
24      def shorten_name(self):
25          '''compresses the name by removing the vowels and spaces.
26          if contains only vowels, name becomes an random string
                integer'''
27          assert len(name) >= 1:
28
29          # ways to shorten name here
30          # ...
31
32
33  # pre: none
34  # post: new point with default coords and name is length >= 1
35  p1 = Location("Station")
36  p2 = Location("Hotel")
37
38  # pre: none
39  # post: coords always set to zero
40  p1.reset()
41
```

# Precondition and postcondition with objects (cont.)

```
42 # pre: none
43 # post: coords always set to 3,4
44 p2.set(3,4)
45
46 # pre: name is length >= 1
47 # post: no vowels in name
48 # post: name is length >= 1
49 p2.shorten_name()
```

These communicate the idea that: this code will *always* produce a true postcondition if the precondition is true

# Testing a method that returns an int

Here's another simple method, that will return the area of a square with integer-valued sides, and a class with a method to test it:

```
1  class Square:
2  # ... lots of code in here to construct the Square instances etc.
3      def get_area(self):
4          return self.size * self.size
```

```
1  class SquareTester:
2      def is_area_ok() {
3          # assume constructor sets size
4          s = Square(3)
5          if 9 != s.get_area():
6              return False
7          return True
```

# Testing a method that returns a floating point

Take care with methods that return floating points numbers. Should not compare floating point numbers with ==:

```
1  import math
2  class Circle:
3      # ...
4      def get_area(self):
5          return math.pi * radius * radius
```

This time I'll throw an exception if the value isn't correct:

```
1  class CircleTester:
2      def test_circle_area1():
3          # sets the radius to 1.0
4          c = Circle(1.0)
5          if c.get_area() != 3.1415926:
6                  raise AssertionError("The area is not correct"+
7                        " for a circle of radius 1")
```

# Testing a method that returns a floating point (cont.)

running this code fails even though my method is correct

*rounding error* is affecting the comparison

Instead of using == directly we test that the *difference* between the expected answer and the returned value is *small*:

```
1  def test_circle_area2():
2      # sets the radius to 1.0
3      c = Circle(1.0)
4      if abs(c.get_area() - 3.1415926) > 0.000001:
5          raise AssertionError("The area is not even"
6                               + " close for a circle of radius 1");
```

This code will only make the `Circle.getArea()` method fail if the area is more than 0.000001 different from what I expect.

# Testing a method that returns a floating point (cont.)

The arbitrary number used is 0.000001, this is a magic number

```
if abs(c.get_area() - 3.1415926) > 0.000001:
```

The problem domain may demand a certain amount of precision

- e.g. mechanical arm must compute movements with **at least** 0.1mm precision

The computer hardware, or software implementation may support a certain amount of precision

- e.g. `sys.float_info.epsilon` - Python machine floating point number epsilon

# Pre : $x \geq 0$
$y = \sqrt{x}$
# Post : $\left|y - \sqrt{x}\right| \leq 10^{-6}$

In most cases, the hardware/software has greater precision than most applications require.

# Idiom with Person Objects

Final all elements in the array matching criteria

A Person object has a date for their birthday

```
class Person:
    # instance variables ...
    # constructor ...
    def get_name(self):
        return name
    def getBirthday(self):
        return self.birthday.copy()
```

```
class SimpleDate:
    # instance variables ...
    # constructor ...
    def copy(self):
        ''' returns a copy of this object '''

    def equals(self,other):
        ''' returns True if this date is equal to the other date'''
```

Given an array of `Person` objects and one object that is `SimpleDate`, with todays date. Print a message if a person's their birthday is `today`.

# Built in mechanisms for testing in Python: unittest

Sample problem. Write a function `max_repeat()` that finds the longest number of *consecutively repeating* characters in a string. If there is no string, it will return -1

FIRST make some test cases

| input | expected |
|---|---|
| not a string | -1 |
| "" | 0 |
| "a" | 1 |
| "abc" | 1 |
| "aa" | 2 |
| "aba" | 1 |
| "abb" | 2 |
| "abbaa" | 2 |
| "ababaccc" | 3 |

# Built in mechanisms for testing in Python: unittest (cont.)

SECOND write the test case code, with an empty shell of the function

Module test_max_repeat_function.py

```
1  # this may be imported from somewhere else
2  def max_repeat(s):
3      pass # does not even return
4
5  import unittest
6  class TestMaxRepeat(unittest.TestCase):
7      def test_not_string(self):
8          self.assertEqual( max_repeat(123), -1 )
9      def test_empty(self):
10         self.assertEqual( max_repeat(""), 0 )
11     def test_single(self):
12         self.assertEqual( max_repeat("a"), 1 )
13
14 if __name__ == '__main__':
15      unittest.main()
```

Run the test code!

```
python3 -m unittest test_max_repeat_function.py
```

# Built in mechanisms for testing in Python: unittest (cont.)

THIRD write code incrementally. Test incrementally.

```
1  def max_repeat(s):
2      if not isinstance(s,str):
3          return -1
4
5      return 0
```

Run the test code again!

```
python3 -m unittest test_max_repeat_function.py
```

# doctest

Documenting is necessary for communication to another reader about what the code does

Placing a string in the right place is treated as a docstring. This becomes a runtime *attribute* of the class/module. Accessible via `__doc__`

```
1 value = float.__doc__
2 print(value)
3
4 value = str.upper.__doc__
5 print(value)
```

```
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.

S.upper() -> str

Return a copy of S converted to uppercase.
```

# doctest (cont.)

## Make your own

```
1  class Person:
2      '''This describes a person class'''
3
4      def __init__(self):
5          ''This is the person class constructor '''
6
7  print(Person.__doc__)
8  print(Person.__init__.__doc__)
```

```
This describes a person class
This is the person class constructor
```

# doctest (cont.)

A complex function requires example usage that could be placed in the docstring.

These examples could be converted into runnable test code

The docstring will contain >>> which invokes the python interpreter and runs code as if it were a unit test.

**Question: I want to know if Meichen has some of the same hobbies as Jason and what they are.**

Our complex problem is: actually finding an intersection of two sets and there are various details about how the data is organised and the question is asked.

Documentation is very subjective and it may take a lot of time to describe. One solid example can make it clearer.

# doctest (cont.)

```
1  class Person:
2      def __init__(self, hobbies):
3          '''This is the person class constructor'''
4          self.hobbies = hobbies
5
6      def get_matching_hobbies(self,a,b):
7          '''Will append to a list of matching hobbies from this
               person to the input list b, and return the results in
                the output list a in sorted order.
8
9          >>> meichen = Person( [ 'party', 'tennis', 'running', '
               singing', 'info1110' ] )
10         >>> result = [ ]
11         >>> meichen.get_matching_hobbies( result, [ 'singing', '
               tennis', 'violin' ] )
12         >>> print(result)
13         ['singing', 'tennis']
14         '''
15         for h_in in b:
16             if h_in in self.hobbies:
17                 a.append(h_in)
```

# doctest (cont.)

```
18          sorted(a)
19
20  if __name__ == "__main__":
21      import doctest
22      doctest.testmod()
```

```
$ python -m doctest -v doctest_example.py
Trying:
    meichen = Person( [ 'party', 'tennis', 'running', 'singing', 'info1110
Expecting nothing
ok
Trying:
    result = [ ]
Expecting nothing
ok
Trying:
    meichen.get_matching_hobbies( result, [ 'singing', 'tennis', 'violin'
Expecting nothing
ok
```

# doctest (cont.)

```
Trying:
    print(result)
Expecting:
    ['singing', 'tennis']
ok
3 items had no tests:
    doctest_example
    doctest_example.Person
    doctest_example.Person.__init__
1 items passed all tests:
   4 tests in doctest_example.Person.get_matching_hobbies
4 tests in 4 items.
4 passed and 0 failed.
Test passed.
```