Notes on Unit Testing

Abridged from Notes by Dr. Burton Ma

September 26, 2021

1 Unit Testing

Unit testing is the process of testing units of source code. A unit of code is the smallest testable part of a piece of software. For our purposes, the units are Python functions.

The textbook does not discuss this topic in any detail, and for most students, testing in previous courses amounts to writing a small program and manually inspecting the output of the program. This notebook discusses writing tests using the `pytest` framework. Using a testing framework has many advantages compared to manual testing:

- a test framework typically produces a summary of the test results indicating which tests passed and which tests failed; the programmer does not need to manually inspect the output of a program to determine the test results
- tools exist that allow tests to be run automatically
  - for example, many professional workplaces require their programmers to check their code into a centralized version control system; such systems can be configured to automatically run test suites whenever code is checked into the system
- test frameworks typically allow the programmer or tester to choose which tests should run; there is no easy way to select tests to run when manually inspecting the output of a program

1.1 `pytest`

`pytest` home page

The unit test framework that is part of the Python standard is called `unittest`. Unfortunately, `unittest` is not suitable for the purposes of CISC121 because it requires knowledge of object-oriented Python.

`pytest` does not require object-oriented Python and is easily accessible for CISC121 students. Because it is a third-party framework, students will need to install it.

Installing `pytest`

1. In Windows, open a Command Prompt (the `cmd` program). On Mac, open a terminal (the `Terminal` program).
2. In the Command Prompt or terminal type the following command:

```
pip3 install -U pytest
```

3. Check that the correct version has been installed correctly by typing the following command:

```
pytest --version
```

You should see the output `pytest 6.2.2` (you might see a version number greater than 6.2.2 depending on when you are reading this document).

2 Unit testing basics

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra

Programmers try to write correct code, but it is very challenging to prove that a piece of code is correct (the field of study that tries to do is called formal methods). Instead, most programmers rely on testing to try to find errors in their code.

When testing a function, we test that the contract of the function is upheld. In other words, we use test cases where the input argument values satisfy the preconditions of the function and then test that the postconditions of the function are satisfied.

Writing a unit test for a Python function is simple in principle:

1. Choose a set of input arguments for the function.
2. If the function returns one or more values, determine what the expected return values are for the chosen set of input arguments.
3. If the function has a side effect, determine what the expected side effect is.
4. Call the function with the chosen set of input arguments and determine if:
    - the actual return values are equal to the expected return values
    - the actual side effects are equal to the expected side effects

A set of input arguments, the expected return values, and the expected side effects make up a *test case*. Testing one function typically requires using multiple test cases. Determining how many test cases are needed is beyond the scope of this notebook, but we can explore this problem somewhat with a few exercises.

Exercise 1 Consider a function having one parameter where the parameter type is a boolean value. How many possible test cases exist for the function?

Exercise 2 Consider a function having two parameters where both parameter types are boolean values. How many possible test cases exist for the function?

Exercise 3 Consider a function having three parameters where all the parameter types are boolean values. How many possible test cases exist for the function?

Exercise 4 Most Python implementations use a 64-bit representation for float values; this means that there are 2^{64} distinct float values. Consider a function having two parameters where both parameter types are float values. How many possible test cases exist for the function?

2.1 Example: Testing a function having one return value

Consider the `is_prime` function from the course textbook (renamed to use PEP8 function naming convention):

```
def is_prime(n):
    if n < 2:
        return False
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True
```

To test the function, we first define a test case by choosing an input argument value and corresponding expected return value. A quick inspection of the function indicates that any input argument value less than 2 results in `False` being returned by the method, which is the correct result. The return value for arguments greater than or equal 2 is less clear. Let's begin with the test case where the input argument value is 2 and the expected return value is `True`.

To write a unit test using `pytest` we import the `pytest` module and then define one or more test functions whose names begins with `test` or ends with `test`. For example, our test case can be implemented as shown below;

```
import pytest

def is_prime(n):
     if n < 2:
         return False
     i = 2
     while i * i <= n:
         if n % i == 0:
             return False
         i += 1
     return True

# test function names need to start or end with test

def test_is_prime_2_true():
     assert is_prime(2) == True


   # run pytest
   pytest.main(["--capture=sys"])
```

The output should have a line containing `1 passed` indicating that the test case has passed.

The Python `assert` statement has the form:

`assert` *expr*

where *expr* is any Python expression whose value can be converted to `True` or `False`. If *expr* evaluates to `True` then the `assert` statement does nothing. If *expr* evaluates to `False` then an `AssertionError` is raised. In a unit test, `pytest` handles the `AssertionError` and notes that the test has failed.

Let's intentionally introduce an error into `is_prime` to see what a failed test looks like;

```
import pytest

def is_prime(n):
    # intentional error on next line
    if n <= 2:
        return False
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True


def test_is_prime_2_true():
    n = 2
    assert is_prime(n) == True

# run pytest
pytest.main(["--capture=sys"])
```

The `pytest` output now indicates that the test has failed. In particular, it shows the input arguments to the function (`is_prime(2)`), the expected return value (`True`), and the actual return value (`False`). For the programmer, knowing what input values cause a function to fail is valuable information when debugging the function.

We have one test case where the expected return value is `True`. We should also create a test case where the expected return value is `False`. The smallest non-prime number is 4, so we can add a second test case where the input argument value is 4 and the expected return value is `False`

```
import pytest

def is_prime(n):
    if n < 2:
        return False
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True


def test_is_prime_2_true():
    n = 2
    assert is_prime(n) == True

def test_is_prime_4_false():
    n = 4
    assert is_prime(n) == False

# run pytest
pytest.main(["--capture=sys"])
```

At this point the reader should realize that tests for the `is_prime` function fall into one of two categories: Those where the expected return value is `False` and those where the expected return value is `True`. Instead of testing a single value in each test function, we can test multiple different input arguments using a single function:

```
import pytest

def is_prime(n):
    if n < 2:
        return False
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1
    return True


def test_is_prime_true():
    N = [2, 3, 5, 7, 11, 13, 17]
    for n in N:
        assert is_prime(n) == True

def test_is_prime_false():
    N = [4, 6, 8, 9, 10, 12, 14]
    for n in N:
        assert is_prime(n) == False

# run pytest
pytest.main(["--capture=sys"])
```

The disadvantage with this approach is that a test written this way will stop on the first test case that fails; the remaining test cases will not be run. A better way to write such tests is to use a *parameterized test* but this is beyond the scope of CISC121.

With a bit of reflection, it should be clear to the reader why unit testing cannot prove that the `is_prime` function is correct: There are an infinite number of possible input arguments to test! Even worse, for very large input values, we have no good way to determine what the expected return value should be.

Exercise 5 Write a small number of tests for the function in the following box. The function is supposed to compute the length of the hypotenuse of a right triangle with sides of lengths `a` and `b`.

```
import pytest
import math

def hypot(a, b):
    return math.sqrt(a * a + b * b)

# write your tests here

# run pytest
pytest.main(["--capture=sys"])
```

2.2 Example: Testing a function that has a side effect

A function has a side effect if it has some observable effect besides returning a value. Printing a value is an example of a side effect. Changing the state of a mutable input argument (such as a list) is another example of a side effect.

Consider the function `exchange` that swaps two elements of a list:

```python
def exchange(a, i, j):
    """
    Swaps two elements of a list.

    Swaps the elements a[i] and a[j] of the list a.

    Parameters
    ----------
    a : list
        A list
    i : int
        Index of an element to swap
    j : int
        Index of a second element to swap

    Raises
    ------
    IndexError
        If `i` or `j` is out of range.
    """

    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

After calling `exchange` every element of `a` should be unchanged except for the elements at indexes `i` and `j` which should be swapped.

(It may seem strange to call the exchange of the two elements a **side effect** when that is the entire purpose of the function! However, this is the terminology we use.)

Testing a function with side effects is similar to testing a function with a return value. One difference is that we probably have to call the function that is being tested before the `assert` statement to cause the side effect to occur.

Let's write a test for `exchange` where we swap the elements of a list of length 2:

```
import pytest

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp

def test_exchange_len2():
    '''
    Test for list of length 2.
    '''

    # input arguments
    a = ['x', 'y']
    i = 0
    j = 1

    # expected list
    exp = ['y', 'x']

    # call the method to cause the side effect
    exchange(a, i, j)

    assert a == exp
```

For CISC121, the above test is a good attempt at writing a unit test for the `exchange` function. It sets up the input arguments and the expected side effect correctly, calls the `exchange` function using the test case arguments, and then tests if the modified list `a` is equal to the expected list `exp`.

A second test case that would be useful to include is a case where the indexes `i` and `j` are equal (corresponding to the case where no exchange actually occurs); given the existing test, it is easy to write such a test case:

```
import pytest

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp

def test_exchange_len2():
    '''
    Test for list of length 2.
    '''

    # input arguments
    a = ['x', 'y']
    i = 0
    j = 1

    # expected list
    exp = ['y', 'x']

    # call the method to cause the side effect
    exchange(a, i, j)

    assert a == exp

def test_exchange_len2_same_indexes():
    '''
    Test for list of length 2 with same indexes.
    '''

    # input arguments
    a = ['x', 'y']
    i = 1
    j = 1

    # expected list
    exp = ['x', 'y']

    # call the method to cause the side effect
    exchange(a, i, j)

    assert a == exp
```

```
# run pytest
pytest.main(["--capture=sys"])
```

The above tests are useful, but they are technically asserting the wrong condition. See the sections near the end of this note starting with the box titled **Extra material: How the exchange tests could be better** for details.

Exercise 6 Write a small number of tests for the function `min_to_front` in the following box. The purpose of the function is that it moves the smallest element of the list to the front of the list and puts the element that was at the front of the list somewhere else in the list.

[ ]:
```

def min_to_front(a):
    if len(a) == 0:
        raise ValueError('min_to_front(): no minimum element in an empty list')

    if len(a) == 1:
        # list of length 1 already has its minimum element at the front of the␣
 ↪list
        return

    min_elem = a[0]
    min_index = 0
    for i in range(1, len(a)):
        ai = a[i]
        if ai < min_elem:
            min_elem = ai
            min_index = i
    front = a[0]
    a.pop(min_index)
    a[0] = min_elem
    a.append(front)
```

2.3 Example: Testing a slightly more complicated function

The above examples are useful for introducing the concept of unit testing using `pyttest` but the tested functions are already correctly implemented. Consider the following function that is supposed to return the second largest element in a list having at least 2 elements:

```python
def second_largest(a):
    '''
    Return the second largest element in a list.

    The list `a` is assumed to have at least two elements that can be compared
    using the >= operator.
    '''

    # look at the first two elements
    largest = max(a[0], a[1])
    second = min(a[0], a[1])

    # look at the remaining elements
    for i in range(2, len(a)):
        elem = a[i]
        if elem >= largest:
            second = largest
            largest = elem
    return second
```

Following the previous examples, we can easily write a unit test by creating a list with two or more elements and setting the expected return value to be equal to the second largest element of the input list. The following box shows three different tests:

```
import pytest

def second_largest(a):
    '''
    Return the second largest element in a list.

    The list `a` is assumed to have at least two elements that can be compared
    using the >= operator.
    '''

    # look at the first two elements
    largest = max(a[0], a[1])
    second = min(a[0], a[1])

    # look at the remaining elements
    for i in range(2, len(a)):
        elem = a[i]
        if elem >= largest:
            second = largest
            largest = elem
    return second


def test_list_len2_1():
    a = [1, 2]
    exp = 1
    assert second_largest(a) == exp

def test_list_len2_2():
    a = [2, 1]
    exp = 1
    assert second_largest(a) == exp

def test_list_len5():
    a = [1, 8, 10, 5, 12]
    exp = 10
    assert second_largest(a) == exp

# run pytest
pytest.main(["--capture=sys"])
```

The function passes all three test cases but there is an error in the function.

Exercise 7 Can you find the error in the function `second_largest`? (Don't go to the next page until you have tried this exercise.)

The error in the function `second_largest` is that it assigns a value to `second` only when a new largest value is found in the list; this means that the function will fail to find the correct second largest value if it comes after the largest value in the list.

Note that the second test case uses a list where the second largest element comes after the largest element, but the function is written in such a way that it works correctly for lists of length 2.

Adding a fourth test case will reveal the error:

```
import pytest

def second_largest(a):
    '''
    Return the second largest element in a list.

    The list `a` is assumed to have at least two elements that can be compared
    using the >= operator.
    '''

    # look at the first two elements
    largest = max(a[0], a[1])
    second = min(a[0], a[1])

    # look at the remaining elements
    for i in range(2, len(a)):
        elem = a[i]
        if elem >= largest:
            second = largest
            largest = elem
    return second


def test_list_len2_1():
    a = [1, 2]
    exp = 1
    assert second_largest(a) == exp

def test_list_len2_2():
    a = [2, 1]
    exp = 1
    assert second_largest(a) == exp

def test_list_len5_1():
    a = [1, 8, 10, 5, 12]
    exp = 10
    assert second_largest(a) == exp

def test_list_len5_2():
    a = [1, 8, 10, 5, 9]
    exp = 9
    assert second_largest(a) == exp

# run pytest
pytest.main(["--capture=sys"])
```

This example again illustrates why it is impossible to use testing to prove that a method is correct: There are an infinite number of input lists and we cannot possibly test them all. If we fail to use a test case that reveals an error then we may erroneously conclude that the function is correct.

2.4 Extra material: How the `exchange` tests could be better

The `exchange` test shown earlier in this notebook does not actually correctly test the behavior of `exchange`. `exchange` is supposed to swap two elements of a list; in other words, after `exchange(a, i, j)` finishes running:

- the element at `a[i]` should refer to the same object that `a[j]` referred to immediately before `exchange` was called
- the element at `a[j]` should refer to the same object that `a[i]` referred to immediately before `exchange` was called
- every element in `a` not at index `i` or `j` should refer to the same object that it referred to immediately before `exchange` was called

The test written for `exchange` uses == to test if the list `a` is equal to a list where the elements have been correctly swapped. Unfortunately, == tests if two objects are equal. For a list, == tests if all of the corresponding elements of two lists are equal. Here is an example of two lists that are equal, but do not contain references to the same objects:

[ ]:
```
a = ['hello world']
b = ['hello world']
print('equal lists?:', a == b)
print('same elements?:', a[0] is b[0])
```

To test if two references refer to the same object we can use the Python operator `is`. Re-writing the previous version of the test leads to the following:

```
import pytest

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp

def test_exchange_len2():
    # input arguments
    a = ['x', 'y']
    i = 0
    j = 1

    # copy of a
    copy = a[:]

    # call the method to cause the side effect
    exchange(a, i, j)

    assert a[i] is copy[j]
    assert a[j] is copy[i]

# run pytest
pytest.main(["--capture=sys"])
```

In the above test, the list `copy` is a copy of the list `a` before the swap occurs: `copy[0]` and `a[0]` are references to the same object, and `copy[1]` and `a[1]` are references to the same object. Calling `exchange` modifies the list `a` but does not modify the list `copy`. After calling `exchange`, `a[i]` and `copy[j]` should refer to the same object, and `a[j]` and `copy[i]` should refer to the same object. Notice the use of `is` in the test to test that the references refer to the same object instead of using == to test if the objects are equal.

Now let's write a test using a slightly longer list:

```python
import pytest

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp


def test_exchange_len2():
    # input arguments
    a = ['x', 'y']
    i = 0
    j = 1

    # copy of a
    copy = a[:]

    # call the method to cause the side effect
    exchange(a, i, j)

    assert a[i] is copy[j]
    assert a[j] is copy[i]


def test_exchange_len3():
    # input arguments
    a = ['x', 'y', 'z']
    i = 1
    j = 2

    # copy of a
    copy = a[:]

    # call the method to cause the side effect
    exchange(a, i, j)

    for k in range(0, len(a)):
        if k == i:
            assert a[k] is copy[j]
        elif k == j:
            assert a[k] is copy[i]
        else:
            assert a[k] is copy[k]

# run pytest
```

```
pytest.main(["--capture=sys"])
```

For lists longer than length 2 we should test that the elements whose indexes are not equal to `i` or `j` are unchanged by the `exchange` function, so there is now a third `assert` statement.

The observant reader will recognize that both of the tests above differ only in the input arguments that are used. We should move the common code into a separate function like this (see next page):

```
import pytest

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp


def exchange_test_impl(a, i, j):
    # copy of a
    copy = a[:]

    # call the method to cause the side effect
    exchange(a, i, j)

    for k in range(0, len(a)):
        if k == i:
            assert a[k] is copy[j]
        elif k == j:
            assert a[k] is copy[i]
        else:
            assert a[k] is copy[k]

def test_exchange_len2():
    # input arguments
    a = ['x', 'y']
    i = 0
    j = 1
    # call the test implementation
    exchange_test_impl(a, i, j)


def test_exchange_len3():
    # input arguments
    a = ['x', 'y', 'z']
    i = 1
    j = 2
    # call the test implementation
    exchange_test_impl(a, i, j)

# run pytest
pytest.main(["--capture=sys"])
```

The function `exchange_test_impl` is the function that actually implements the testing logic (the `_impl` is an abbreviation of *implementation*). We have to be careful when naming this function not to start or end the name with `test` otherwise `pytest` will think that it is a separate test function.