More on Testing

CPEN333 – System Software Engineering
2021 W2
University of British Columbia

Introduction

- As a good programmer we want to test early and often.
  - We want to test each method fully as we implement one, before moving on to the next.
  - And we want to test that the program as a whole is correct, readable and complies with the specification.

- In this set of slides, we discuss testing and testing in Python.

Objectives

- Understand validation and testing terminology, stages and methods
- Describe the reasoning for unit testing
- Use arrange, act, assertion pattern in a unit test
- Use Python unittest framework

Software Validation

- Software validation is the process of checking that the software conforms to its specifications.

- Validation includes:
  - Testing
    - Running the program on carefully selected test cases.
  - Code review
    - Having somebody else carefully read your code to verify correctness.
  - Formal reasoning (verification):
    - Verification constructs a formal proof that a program is correct. This is beyond the scope of the discussions in this set of slides.

Code Review

- Code review is careful, systematic study of source code by others.

- This is a good practice, for example, in a group projects, where each group-mate carefully reviews some other group-mate's code.

Why Software Testing is Hard

- Exhaustive testing is infeasible
  - Space is generally too big to cover exhaustively

- Haphazard testing ("just try it and see if it works") is unreliable and doesn’t help much (maybe a bit if software is very buggy).

Testing Strategy

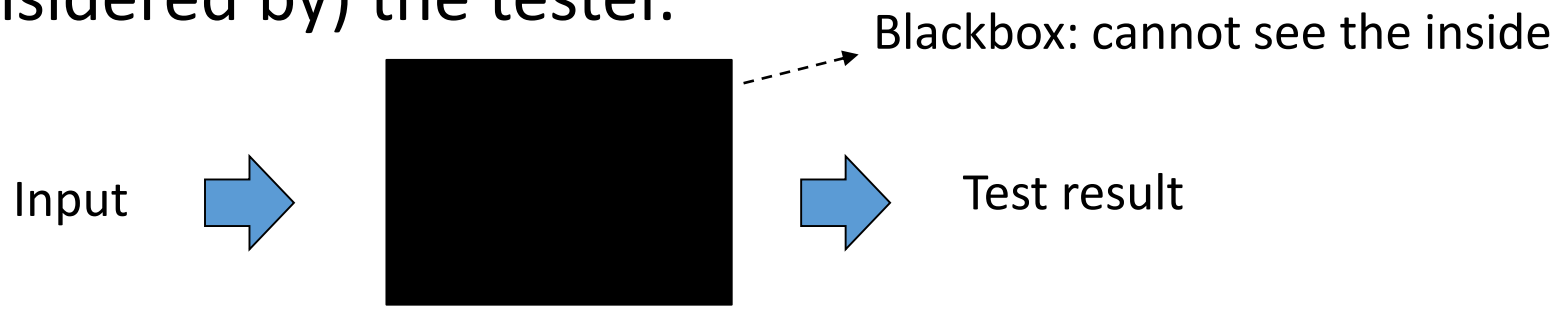
- We want to find bugs as cheaply and quickly as possible

- We use a test suite that increases the chance of finding bugs
  - A test suite is a collection of tests

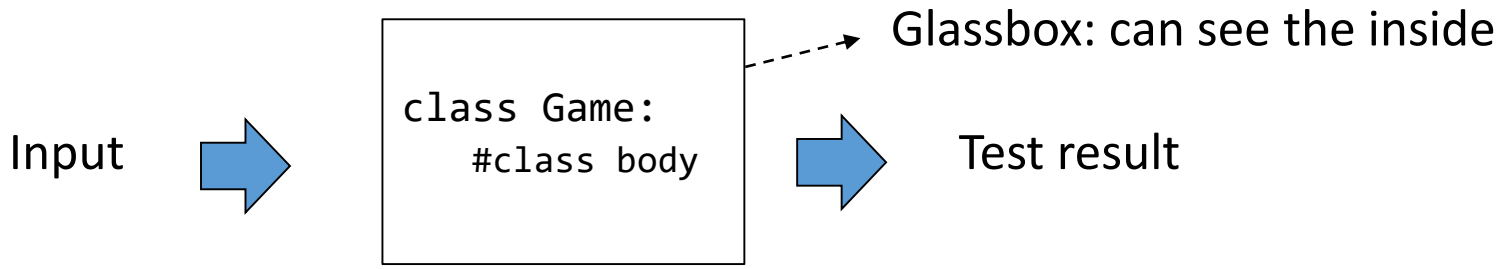
- Design testing strategy carefully
  - Select test cases carefully
  - Test early and often
  - Utilize automated tests (unit testing) to increase frequency
  - Complement with other methods: code review, reasoning,..

Testing Methods: Blackbox and Whitebox

- Blackbox testing: Choose test cases only from the specs
  - It is called blackbox since the internal implementation of the method (component/item) being tested in not known to (or not being considered by) the tester.



- Whitebox testing (glassbox testing): Choose test cases with knowledge of how the method (component/item) is implemented.



Testing stages

- Testing is most effective if done in stages

- For example, we normally consider the following stages:

  - Unit testing: testing individual units of source code such as methods, …

  - Integration testing: testing units combined as a group

  - Acceptance testing: testing to determine if user needs (based on the requirements) have been met

Unit testing

- Unit testing is the process of testing program components, such as methods or objects.
  - The most common and simplest type of components are methods.
  - Unit testing frameworks allow us to automatically test all units as we develop more.
  - Test data is usually chosen by developer(s) based on their understanding of specification and knowledge of the unit
- Unit testing:
  - can be at various levels of granularity
  - can be open box (by the developer(s)) or closed box (by special testers)

Testing Python code

- Let's start with some simple examples

- We can use `print()` to test

```
def add (a: float, b: float) -> float:
    return a + b

#test case 1: -1 + 15 = 14
a, b = -1, 15
print(f"Testing: {a} + {b} = {add(a, b)}")
```

- This is straightforward and works in any language, but maybe cumbersome, not automatic …
- We have to see the output and visually verify its correctness.

assert()

- Python's standard library has the `assert` function.
- `assert()`
  - "Raises AssertionError if the specified expression evaluates to False"
  - https://python-reference.readthedocs.io/en/latest/docs/statements/assert.html
- We can use assertion to test correctness of the code or that a condition is met.
- Let's rewrite the previous slide's code with assertion.

```
def add (a: float, b: float) -> float:
    return a + b

#test case 1: -1 + 15 = 14
a, b = -1, 15
assert a + b == add(a, b)
```

if the expression is True nothing happens, but if it is False an exception is raised

assert()(cont.)

- The general format of assert is

```
assert expression [","  expression]
```

  - which is equivalent to:

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

  - `__debug__` is a built-in variable which is `True` under normal circumstances, and `False` when optimization is requested.
  - `AssertionError` is a built-in exception

Examples

- Examples when assertion fails.

```
def add (a: float, b: float) -> float:
    return a + b + 1    #deliberately incorrect for demo

#test case 1: -1 + 15 = 14
a, b = -1, 15
assert a + b == add(a, b)
```

output:

```
Traceback (most recent call last):
  File "…", line 6, in <module>
    assert a + b == add(a, b)
AssertionError
```

```
def add (a: float, b: float) -> float:
    return a + b + 1    #deliberately incorrect for demo

#test case 1: -1 + 15 = 14
a, b = -1, 15
assert a + b == add(a, b), f"expected {a+b} got {add(a,b)}"
```

output:

```
Traceback (most recent call last):
  File "…", line 6, in <module>
    assert a + b == add(a, b), f"expected {a+b} got {add(a,b)}"
AssertionError: expected 14 got 15
```

with optional own message

More examples

- Different varieties of assert

```
a = 1
b = 2
assert a < b          #comparison assertion

num = [1, 2, 3]
assert 1 in num       #membership assertion

x = 1
y = x
assert x is y          #identity assertion
```

Python `unittest` framework

- unittest unit testing framework has a similar flavor as major unit testing frameworks in other languages.

- It is a part of Python's standard library and allows:
  - test automation (test runner)
  - aggregation of tests into collections,
  - sharing of setup and shutdown code for tests
  - …
- unittest has its own set of assertion function instead of the built-in `assert` statement.

Example

```
import unittest   #importing the test framework class

def add (a: float, b: float) -> float:  #unit to test
    return a + b

def subtract (a: float, b: float) -> float:  #unit to test
    return a - b -1  #deliberately incorrect

class Test(unittest.TestCase):
    def test_add(self):
        a, b = -1, 15
        self.assertEqual(add(a, b), a+b)

    def test_subtract(self):
        a, b = -1, 15
        self.assertEqual(subtract(a, b), a-b)

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

The units that we want to test. Here they are functions, but the could be methods of a class, …
Could be put in their own file

the unit tests: we must have many of these, one for each test case

running the tests

Writing Unit Test Methods

- A unit test involves an Arrange/Act/Assertion pattern.
  - Arrange is the setup part in which we initialize and select the test case.
  - Act is when we call the method to be tested.
  - Assertion is the part in which we compare the result of the call with the expected result.

```
    def test_add(self):
        a, b = -1, 15                       #arrange (setup)
        arithmetic = Arithmetic()
        result = arithmetic.add(a, b)    #act
        self.assertEqual(result, a+b)    #assertion
```

- For simpler cases, we may combine, for example act and assertion.
- A unit test passes if the assertion passes, and fails otherwise.

Example 2

- We normally separate the unit test code from our code to be tested.
- Assume we want to test the following class that is saved in the `demo.py` file:

```
class Arithmetic:
    def add (self, a: float, b: float) -> float:
        return a + b

    def subtract (self, a: float, b: float) -> float:
        return a - b + 1   #deliberately incorrect
```

- We can create a separate python file for the unittest and we use an import statement to import our class (for testing its methods):

```
from demo import Arithmetic
```

Example 2 (cont.)

- Here is the test code stored in a separate python file:

```
from demo import Arithmetic

import unittest   #importing the test framework class

class Test(unittest.TestCase):

    def test_add(self):
        a, b = -1, 15
        arithmetic = Arithmetic()
        self.assertEqual(arithmetic.add(a, b), a+b)

    def test_subtract(self):
        a, b = -1, 15
        arithmetic = Arithmetic()
        self.assertEqual(arithmetic.subtract(a, b), a-b)

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

class unittest.TestCase

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

class `unittest.`**`TestCase`**(*`methodName='runTest'`*)

Instances of the `TestCase` class represent the logical test units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single base method: the method named *methodName*. In most uses of `TestCase`, you will neither change the *methodName* nor reimplement the default `runTest()` method.

Changed in version 3.2: `TestCase` can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with `TestCase` from the interactive interpreter.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

unittest assertion methods

- https://docs.python.org/3/library/unittest.html

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)`

Examples: unittest assertion methods

- `assertEqual`:

assertEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

- `assertTrue` and `assertFalse`:

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

Other unit testing frameworks

- There are other unit testing frameworks for python.

- An example is `pytest`.
  - https://docs.pytest.org/en/7.1.x/
  - It uses python's own `assert` statement.
  - It is simpler but rather less capable.
  - A unit test method name starts with `test_`
  - We can run the test by using: `pytest nameOfOurPythonFile.py`

References

- assert: https://python-reference.readthedocs.io/en/latest/docs/statements/assert.html
- unittest: https://docs.python.org/3/library/unittest.html#module-unittest
- Alternative testing frameworks:
  - pytest: https://docs.pytest.org/en/7.1.x/