Automated Unit Testing

Dr. Carol Alexandru-Funakoshi

University of Zurich, Department of Informatics

Four-Phase Unit Testing

Each programmed unit should be tested. This should be done in isolation to avoid side effects introduced through mistakes in other units.

Such a unit test follows four phases:

- Setup (Prepare the environment)
- **Exercise** (Interact with the *system-under-test*, SUT)
- **Verify** (assert that the output/state is correct)
- Teardown (Bring everything back to the original state)

Please note that exercise and verify are the crucial phases. Setup and teardown are **optional** and are only needed for certain kinds of tests.

Excursion: Using Different Source-Code Files

mathutils.py

```
def square(x):
    return x * x
```

my_program.py

```
from mathutils import square

print ("3^2 = %d" % square(3))
```

We will talk more about how to use modules in Python in the upcoming lecture "Organizing Code: Modules and Version Control"

Defining Unit Tests in `unittest`

(Python's built-in unit testing framework)

mathutils.py

```
def square(x):
    return x * x
```

A good unit test should only test one thing and, therefore, should have exactly one assert!

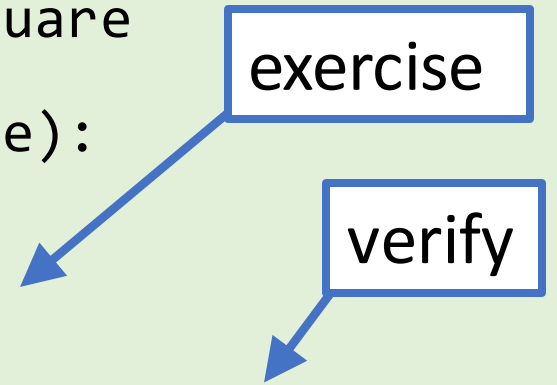
square_test.py

```
from unittest import TestCase
from mathutils import square

class SquareTest(TestCase):

    def test_zero(self):
        actual = square(0)
        expected = 0
        self.assertEqual(expected, actual)

    def test_something_else(self):
        ...
```



Pick the right assert method

- Equality:
  - assertEqual
  - assertAlmostEqual
  - assertIs
- Relations:
  - assertGreaterThan
  - assert…
- Exceptions:
  - assertRaises
- …

Keep the test short and readable by using the right assertion method!

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

Exmple 1 – Where is Waldo?

- Function `where_is_waldo`, which takes a list of strings as a parameter `names` and returns the index of the string `"Waldo"` or `None` if `"Waldo"` is not present.
- Implement a test suite covering this specification
  1. Think about each individual failure condition independently, as if asking "What if the implementation works perfectly except for this one specific thing?"
  2. Write a test that checks for that particular error

Exmple 1 – Where is Waldo? – Solution

```
def where_is_waldo(n):
  if "Waldo" not in n: return None
  return n.index("Waldo")
```

```
from unittest import TestCase

from waldo import where_is_waldo

class WaldoTest(TestCase):
    def test_empty(self):
        expected = None
        actual = where_is_waldo([])
        self.assertEqual(actual, expected)
    def test_not_in_list(self):
        expected = None
        actual = where_is_waldo(["foo", "bar"])
        self.assertEqual(actual, expected)
    def test_in_list(self):
        expected = 2
        actual = where_is_waldo(["foo", "bar", "Waldo", "baz"])
        self.assertEqual(actual, expected)
```

Why should I bother writing unit tests?

- Unit tests can be automated, it is cheap to test "everything at once"
- It is very easy to repeat a test
- Unit tests can serve as documentation
- A solid test suite provides a safety net and encourages refactorings

Refactoring: Changing the source code of a program to improve its quality, without changing its behavior.

Exmple 2 – Refactoring

- Function average, which takes a list of numbers as a parameter `values` and returns their average. Returns None if `values` is empty.
- Implement a test suite covering this specification

```
def average(values):
  if values == []: return None
  s = 0
  for v in values: s += v
  return s / len(values)
```

- What if non-numbers are in the list?

Exmple 2 – Refactoring – Solution

```
def average(values):
    if values == []: return None
    s = 0
    for v in values:
      try:
        s += v
      except TypeError:
        raise TypeError(
"values contains non-numeric element")
    return s / len(values)
```

```
from unittest import TestCase

from avg import average

class AvgTest(TestCase):
    def test_empty(self):
        self.assertEqual(average([]), None)
    def test_integer_avg(self):
        self.assertEqual(average([1,2,3]), 2)
    def test_float_avg(self):
        self.assertAlmostEqual(
             average([1,2,4]), 2.33, 2)
    def test_non_numbers(self):
        with self.assertRaises(
             TypeError,
             msg="values contains non-numeric element"):
            average([1,2,"hello",3])
```

Black-Box Testing

Interesting test cases are solely identified by studying the specification of a program, e.g., by testing relevant boundaries between value domains. This can be natural domains (e.g., positive and negative numbers) or problem-specific domains (e.g., age > 18).

White-box Testing

```
def foo(x, y):
  if x == 13:
    …
  elif x < y:
    …
  else
    for i in range(x, y):
      …
```

Interesting test cases are identified by studying the implementation of a program, e.g., by testing relevant conditions or case distinctions. White-box tests often try to maximize the count of tested lines in the program (test coverage)

Regression Testing

- Often, it is hard to understand and find a bug
- You can use debugging to explore erroneous executions
- Once understood, **replicate the problem** in a unit test
- The resulting regression test resembles this specific use case
- Regression tests **prevent** the same bug from being reintroduced

Test Generation

- Often, it is possible to write test code that automatically checks many different inputs for specific properties.

```
def square(x):
  …

for i in range(-100, 100):
  assert square(i) >= 0
```

Fuzz Testing

- A "Fuzzer" generates random values over a specified range, e.g.
  - Random characters between a-z, A-Z, 0-9
  - Random strings of characters (exercise 08)
  - Random numbers in a given range, e.g.

```
def fuzz(x, y):
    return random.randint(x, y)

def square(x):
  …

for i in range(100):
  assert square(fuzz(-100000, 100000)) >= 0
```

Manual Testing

- What you probably started with **– but please don't make it a habit!**

Manual Testing

- What you probably started with **– but please don't make it a habit!**
- Sometimes it is very hard to define automated test suites. For example, when network accesses, UI interactions, or system events are involved. Instead of testing individual parts with automated unit tests, it can be easier to manually test the whole system at once.
- It is important to execute this testing strategy in a structured fashion to make sure that important use cases are covered and that no corner cases are forgotten.
- It is crucial to formulate test plans that reflect which actions should be tested in which environment and to define the expected output.

Important Points to Define in a Test Plan

- What is the initial state of my application? How do I get there?
- Which exact steps do I need to perform for the test?
- What do I need to check to assert correctness?

Test-driven development (TDD)

- This is the idea of "write the tests first" and then the program
  1. Determine what the program should do (requirements engineering)
  2. Consider all edge cases and functional requirements, then implement them as unit tests
  3. Start implementing to see more and more tests "go green"
  4. When all tests pass, you're probably done

Exmple 3 – Hot-Dog Stand

```
menu = {
   "Hot Dog":       3.50,
   "Spicy Dog":     4.00,
   "Vegan Dog":     3.50,
   "Water":         1.50,
   "Fizzy Drink":   2.50,
   "Beer":          4.00
}
```

- You are programming the cashier's system for a hot-dog stand with a menu as shown below.

- However, you offer some special discounts if certain conditions apply:
  - When ordering a water with a Spicy-Dog, the water is free. This applies for every pair of these products ordered even if ordering multiple pairs.
  - Every 6th beer in an order is free.

- Implement a function `bill` which takes two parameters, a dictionary `order`, which is mapping product names to the number of each item ordered, and a dictionary `menu` which maps product names to the price of each product. The function should return the total sum of the order.

You should be able to answer the following:

- What is a call stack? A Frame?
- What are exceptions? How do you raise and handle them?
- How can I stop a program execution at at specific point?
- Which tools do I have to interact with a running program?
- When should I debug, when should I write unit tests?
- What is the dis-/advantage of unit testing, compared to debugging?
- What is the difference between white-box and black-box testing?
- What is the relation between unit tests and regression tests?
- How can I use `unittest` to write unit tests in Python?