

Unit Testing

Benjamin Roth
Florian Fink

CIS LMU

Unit Testing: Motivation

- ▶ It is unavoidable to have errors in code.
- ▶ Unit-testing helps you ...
 - ▶ ... to catch certain errors that are easy to automatically detect.
 - ▶ ... to be more clear about the specification of the intended functionality.
 - ▶ ... to be more stress-free when developing.
 - ▶ ... to check that functionality does not change when you re-organize or optimize code.
- ▶ Today, we will look at two frameworks for unit testing that come prepackaged with Python
 1. `doctest`: A simple testing framework, where example function calls (together with their expected output) are written into the docstring documentation, and then are automatically checked.
 2. `unittest`: A framework, where several tests can be grouped together, and that allows for more complex test cases.

Simple Tests: the doctest module

- ▶ Searches for pieces of text that look like interactive example Python sessions inside of the **documentation parts** of a module.
- ▶ These examples are run and the results are compared against the expected value.
- ▶ `example_module.py`

```
def square(x):  
    """Return the square of x.  
  
    >>> square(2)  
    4  
    >>> square(-2)  
    4  
  
    """  
    return x * x
```

Running the tests

```
$ python3 -m doctest -v example_module.py
```

```
Trying:
```

```
    square(2)
```

```
Expecting:
```

```
    4
```

```
ok
```

```
Trying:
```

```
    square(-2)
```

```
Expecting:
```

```
    4
```

```
ok
```

```
1 items had no tests:
```

```
    example_module
```

```
1 items passed all tests:
```

```
    2 tests in example_module.square
```

```
2 tests in 2 items.
```

```
2 passed and 0 failed.
```

```
Test passed.
```

```
$
```

Test-Driven Development (TDD)

- ▶ Write tests first (, implement functionality later)
- ▶ Add to each test an empty implementation of the function (use the `pass`-statement)
- ▶ The tests initially all fail
- ▶ Then implement, one by one, the desired functionality
- ▶ Advantages:
 - ▶ Define in advance what the expected input and outputs are
 - ▶ Also think about important boundary cases (e.g. empty strings, empty sets, `float(inf)`, 0, unexpected inputs, negative numbers)
 - ▶ Gives you a measure of progress (*“65% of the functionality is implemented”*) - this can be very motivating and useful!

TDD: Initial empty implementation

► example_module.py

```
def square(x):
```

```
    """Return the square of x.
```

```
    >>> square(2)
```

```
    4
```

```
    >>> square(-2)
```

```
    4
```

```
    """
```

```
    pass
```

Initially the tests fail

```
$ python3 -m doctest -v example_module.py
Trying:
    square(2)
Expecting:
    4
*****
File "/home/ben/tmp/example_module.py", line 4, in example_module.square
Failed example:
    square(2)
Expected:
    4
Got nothing
Trying:
    square(-2)
Expecting:
    4
*****
File "/home/ben/tmp/example_module.py", line 6, in example_module.square
Failed example:
    square(-2)
Expected:
    4
Got nothing
1 items had no tests:
    example_module
*****
1 items had failures:
    2 of 2 in example_module.square
2 tests in 2 items.
0 passed and 2 failed.
***Test Failed*** 2 failures.
$
```

The unittest module

- ▶ Similar to Java's *JUnit* framework.
- ▶ Most obvious difference to doctest: test cases are not defined inside of the module which has to be tested, but in a separate module just for testing.
- ▶ In that module ...
 - ▶ `import unittest`
 - ▶ import the functionality you want to test
 - ▶ define a class that inherits from `unittest.TestCase`
 - ▶ This class can be arbitrarily named, but `XYZTest` is standard, where `XYZ` is the name of the module to test.
 - ▶ In `XYZTest`, write member functions that start with the prefix `test...`
 - ▶ These member functions are automatically detected by the framework as tests.
 - ▶ The tests functions contain `assert`-statements
 - ▶ Use the `assert`-functions that are inherited from `unittest.TestCase` (do not use the Python built-in `assert` here)

Different types of asserts

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x)</code> is True	
<code>assertFalse(x)</code>	<code>bool(x)</code> is False	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Question: ... what is the difference between “`a == b`” and “`a is b`”?

Example: using unittest

► test_square.py

```
import unittest
from example_module import square

class SquareTest(unittest.TestCase):
    def testCalculation(self):
        self.assertEqual(square(0), 0)
        self.assertEqual(square(-1), 1)
        self.assertEqual(square(2), 4)
```

Example: running the tests initially

► test_square.py

```
$ python3 -m unittest -v test_square.py
testCalculation (test_square.SquareTest) ... FAIL
```

```
=====
FAIL: testCalculation (test_square.SquareTest)
```

```
-----
Traceback (most recent call last):
```

```
  File "/home/ben/tmp/test_square.py", line 6, in testCalculation
    self.assertEqual(square(0), 0)
```

```
AssertionError: None != 0
```

```
-----
Ran 1 test in 0.000s
```

```
FAILED (failures=1)
```

```
$
```

Example: running the tests with implemented functionality

```
$ python3 -m unittest -v test_square.py  
testCalculation (test_square.SquareTest) ... ok
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```

```
$
```

SetUp and Teardown

- ▶ `setUp` and `tearDown` are recognized and executed automatically before (after) the unit test are run (if they are implemented).
- ▶ `setUp`: Establish pre-conditions that hold for several tests.
Examples:
 - ▶ Prepare inputs and outputs
 - ▶ Establish network connection
 - ▶ Read in data from file
- ▶ `tearDown` (less frequently used): Code that must be executed after tests finished
Example: Close network connection

Example using setUp and tearDown

```
class SquareTest(unittest.TestCase):
    def setUp(self):
        self.inputs_outputs = [(0,0),(-1,1),(2,4)]

    def testCalculation(self):
        for i,o in self.inputs_outputs:
            self.assertEqual(square(i),o)

    def tearDown(self):
        # Just as an example.
        self.inputs_outputs = None
```

Summary

- ▶ Test-driven development
- ▶ Using doctest module
- ▶ Using unittest module
- ▶ Also have a look at the online documentation:
<https://docs.python.org/3/library/unittest.html>
<https://docs.python.org/3/library/doctest.html>
- ▶ Questions?