Automated Testing

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Prerequisities:

- functions
- modules

Test your code!

- You do not know whether your code works until you test it, i.e. **until you try to use it!**

Example: `sum_digits()`

Specifications: In module `tools.py`, create function `sum_digits(string)` which return the sum of all digits in `string`.

Solution: We create the required module as follows:

In [1]:
```
%%writefile tools.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum
```

```
Writing tools.py
```

Are we finished? How do we test the code?

Option 1: Try to use it in Python shell

In [2]:
```
>>> from tools import sum_digits
>>> sum_digits('1, 2, 3, dee, dah, dee')
```

Out[2]:
```
6
```

- We have tested a single test case.
- We have to manually check the correctness of the result.
- What if we want to run the test again?

Option 2: Including the test code directly in the module

The code previously written on Python console can be stored directly with the module (or in some other module).

In [3]:
```
%%writefile tools2.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    # All the code below is executed only when the file is run as a script.
    print(sum_digits('1, 2, 3, dee, dah, dee'))
```

```
Writing tools2.py
```

In [4]:
```
import tools2   # "Nothing" happens when we import the module (desired), ...
```

In [5]:
```
%run tools2.py     # ... but the testing code is executed when we run the module!
```

```
6
```

- We still test a single test case only.
- We still have to manually check the correctness of the result.
- **But we can run the test easilly. As many times as we want!**

Option 3: Check the correctness of the result automatically

Instead of mere printing out the result, we can check its correctness!

In [6]:
```
%%writefile tools3.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    observed = sum_digits('1, 2, 3, dee, dah, dee')
    expected = 6
    if observed == expected:
        print('.')
    else:
        print('Test failed.')
        print('- Expected:', str(expected))
        print('- But got: ', str(observed))
```

```
Writing tools3.py
```

In [7]:
```
%run tools3.py
```

```
.
```

- We still test a single test case only.
- **But we do not have to manually check the correctness of the result, we can immediately see if the test passed or not.**
- **And we can run the test easilly. As many times as we want!**

Our own module for testing!

The process of checking the correctness of a result may be extracted to a function that will

- allow us to write tests using only a little code,
- be part of a module that can be reused in many projects.

Let's create module `testing` with function `test_equal()` which shall have 3 parameters:

- the `observed` and `expected` values, and
- an optional `name` of the test.

The function shall print

- `"."` if the test passes, or
- an informative message about the failure, if the test fails.

In [8]:
```
%%writefile testing.py
import sys

def quote(name):
    if name:
        name = "'" + name + "' "
    return name

def test_equal(observed, expected, name=''):
    """Compare the observed and expected results"""
    if observed == expected:
        print('.', end='')
    else:
        linenum = sys._getframe(1).f_lineno   # Get the caller's line number.
        print("\nTest {}at line {} FAILED:".format(quote(name), linenum))
        print("- Expected:", str(expected))
        print("- But got: ", str(observed))
```

```
Writing testing.py
```

With the help of our `testing` module, we can rewrite the `tools` module as follows:

In [9]:
```
%%writefile tools4.py
from testing import test_equal

def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    test_equal(sum_digits('1, 2, 3, dee, dah, dee'), 6, 'Test 1')
```

```
Writing tools4.py
```

In [10]:
```
%run tools4.py
```

```
.
```

- We still test a single test case only.
- **But we do not have to manually check the correctness of the result, we can immediately see if the test passed or failed.**
- **And we do not need to write much code to test a single case!**
- **And we can run the tests easilly. As many times as we want!**

Adding more tests

When we have more test cases, we can add them either

- to the `if __name__=="__main__"` section of the main file, or
- to a separate testing module.

Let's create a separate testing module.

In [11]:
```
%%writefile test_tools.py
from testing import test_equal
from tools4 import *

def test_sum_digits():
    test_equal(sum_digits(''), 0, 'Test empty string')
    test_equal(sum_digits('0'), 0, 'Test 0')
    test_equal(sum_digits('1'), 1, 'Test 1')
    test_equal(sum_digits('2'), 2, 'Test 2')
    test_equal(sum_digits('3'), 3, 'Test 3')
    test_equal(sum_digits('4'), 4, 'Test 4')
    test_equal(sum_digits('5'), 5, 'Test 5')
    test_equal(sum_digits('6'), 6, 'Test 6')
    test_equal(sum_digits('7'), 7, 'Test 7')
    test_equal(sum_digits('8'), 8, 'Test 8')
    test_equal(sum_digits('9'), 9, 'Test 9')
    test_equal(sum_digits('1, 2, 3, dee, dah, dee'), 6, 'Non trivial test')

# Run the test suite
test_sum_digits()
```

```
Writing test_tools.py
```

In [12]:
```
%run test_tools.py
```

```
......
Test 'Test 5' at line 11 FAILED:
- Expected: 5
- But got:  0
.....
```

Ha! We have an error in our code! Can you find it?

With the help of a testing framework:

- We can easilly build comprehensive test suites.
- We do not have to manually check the correctness of the result, we can immediately see if the test passed or failed.
- We do not need to write much code to test a single case!
- We can run the test suite easilly. As many times as we want.

Other testing frameworks

Our module `testing` is not an original idea. Python has several popular testing frameworks, e.g. modules

- `doctest` and
- `unittest`.

Testing the code using `doctest`

- Create the habit to include examples of the functions' usage in their docstrings (see below).
- Module `doctest` allows you to easilly execute the examples from the docstrings:

In [13]:
```
%%writefile modulewithdoctests.py
def average(x,y):
    """Return the average of 2 numbers.

    >>> average(10,20)
    15.0
    >>> average(1.5, 2.0)
    1.75
    """
    return (x + y) / 2

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

```
Writing modulewithdoctests.py
```

Then, if you run the module, the tests are executed automatically and compared with their expected results:

In [14]:
```
%run modulewithdoctests.py
```

```
Trying:
    average(10,20)
Expecting:
    15.0
ok
Trying:
    average(1.5, 2.0)
Expecting:
    1.75
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.average
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Summary

- Testing your own code is **extremely important!**
- You should learn several ways how to test your code.
- Using a **testing framework**, from simple ones (like our `testing`) to comprehensive ones (like `unittest`), gives you an considerable **advantage**!
- Testing frameworks like `unittest` are common to many other languages. If you learn it for one languaga, you will profit from it also in the other languages.

Notebook config

Some setup follows. Ignore it.

In [15]:
```
from notebook.services.config import ConfigManager
cm = ConfigManager()
cm.update('livereveal', {
              'theme': 'Simple',
              'transition': 'slide',
              'start_slideshow_at': 'selected',
              'width': 1268,
              'height': 768,
              'minScale': 1.0
})
```

Out[15]:
```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [16]:
```
%%HTML
<style>
.reveal #notebook-container { width: 90% !important; }
.CodeMirror { max-width: 100% !important; }
pre, code, .CodeMirror-code, .reveal pre, .reveal code {
    font-family: "Consolas", "Source Code Pro", "Courier New", Courier, monospace;
}
pre, code, .CodeMirror-code {
    font-size: inherit !important;
}
.reveal .code_cell {
    font-size: 130% !important;
    line-height: 130% !important;
}
</style>
```