Introduction to Python and its unit testing framework

Lab Instructions

There is nothing to hand in with this lab. You will be orally marked by the lab assistant. You will show code that have produced, and the lab assistants will ask you questions about the code. The lab can be done in pairs, but both members of the group are required to understand the code that you have written. You will have 4 occasions when the lab can be marked (and only 4):

- During the lab.
- On Friday 15th of November by the lab assistant in his office. Times should be scheduled by email in advance with the assistant.
- By appointment with the lab assistant during the week 16/12 – 21/12.
- A yet to be announced time during the re-exam period in the middle of Aug. 2012.

Lab Goals

There are two goals in this lab: The first goal is to get a little experience programming Python, and the second goal is to learn how to use Python's unit testing framework. This lab should be easy and is designed as preparation for the next lab. If you get stuck you should look at the Python tutorial at `http://docs.python.org/tutorial/` or at the extensive documentation at `http://docs.python.org/`. If you have had some experience with Python then this lab should be very easy. Hopefully if you have had no experience with Python, then this lab will help to get you started for the next lab.

Note on Python versions

Python is a changing language. Python 3.0 is in places very different from early versions of Python. For various reasons, older versions of Python are in use on the Unix terminals. But all of the code presented in this lab should work for any version of Python later than 2.4. Note therefore that on the terminal servers, writing just 'python' will invoke Python 2.3, while 'python2' will invoke the latest version, Python 2.7.

Python Unit Testing Mechanics

We will split our code up into two files `code.py` and `testcode.py`. In your favourite text editor, create the following skeleton for `testcode.py`

```
import code
import unittest

class TestCode(unittest.TestCase):
    def test_simple(self):
        self.assertEqual(code.return_zero(),0)
if __name__ == '__main__':
    unittest.main()
```

and the following for `code.py`

```
def return_zero():
    return 0
```

Note that in Python, *indentations matter*. So for example the `def test_simple(self)` function is a part of the TestCode class *because it has a deeper indentation*. Likewise, the `self.assertEqual` test is part of the `test_simple` function. In Python, levels of indentations are used instead of the enclosing brackets you may be used to from many other languages. Take care to use *either* spaces *or* tabs for indentations. Never use both in the same file.

Now at the command line, or using your favourite IDE, do

```
hamberg> python2 testcode.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

What happened here? We have defined a rather silly function that returns 0. We assert that the code also returns 0 using the `self.assertEqual` function in `testcode.py`

1 Hand in

Python like C++ and Java uses == for comparison. So for example 1 == 2 is `False` while 1 == 1 is `True`. Rewrite the above test to use `self.assertTrue`. Note that you need a version of Python greater than or equal to 2.4 to use `assertTrue`. As mentioned, on the department Unix terminals you can use the command line:

```
hamberg> python2 testcode.py
```

to invoke the latest installed version of Python 2.

Functions and arguments

Python functions can take arguments. For example we can write the function `double(x)` (insert this into `code.py`):

```
  def double(x):
    return 2*x
```

we can then write some unit tests (in `testcode.py`):

```
import code
import unittest

class TestCode(unittest.TestCase):
    def test_simple(self):
        self.assertEqual(code.return_zero(),0)
    def test_double(self):
        self.assertEqual(code.double(2),4)
        self.assertEqual(code.double(4),8)
        self.assertEqual(code.double(0),0)
        self.assertEqual(code.double(-2),-4)

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

and run them as usual with `python2 testcode.py`. Note that each test function starts with `test`. Note again that indentation is important. From now on only the new test functions will be given. It is up to you to add them to `testcode.py` and make sure that you have the correct indentation.

2 Hand In

Now write a new function `triple` in `code.py` and write a similar set of tests as seen above. You function `triple` should triple its argument.

Making Decisions

Python like any other language has `if` statements. For example the function:

```
  def if1(x):
    if x==0:
        return 'zero'
    if x>0:
        return 'positive'
    if x<0:
        return 'negative'
```

returns a string. This can be tested using the following test cases.

```
def test_if1(self):
    self.assertEqual(code.if1(0),'zero')
    self.assertEqual(code.if1(1),'positive')
    self.assertEqual(code.if1(-1),'negative')
```

3 Hand In

Type the code in and run the tests to make sure that it all works. Python uses `elif` as an abbreviation of `else if`. Write a new version of `if1` called `if2` that uses two `elif` instead of the corresponding `if` statements. You have to work out which ones you change. It should pass the tests:

```
def test_if2(self):
  self.assertEqual(code.if2(0),'zero')
  self.assertEqual(code.if2(1),'positive')
  self.assertEqual(code.if2(-1),'negative')
```

For loops

We met the range statement in the lecture. This rather boring function just returns the same integer that it is given by a rather roundabout route.

```
def loop1(x):
  result = 0
  for n in range(1,x+1):
      result = result + 1
  return(result)
```

as witnessed by the test cases:

```
def test_loop1(self):
  self.assertEqual(code.loop1(0),0)
  self.assertEqual(code.loop1(19),19)
```

4 Hand In

Write a function `loop_sum(x)` which sums the integers 1,2, up to `x`. It should pass the test cases:

```
    def test_loop_sum(self):
        self.assertEqual(code.loop_sum(0),0)
        self.assertEqual(code.loop_sum(1),1)
        self.assertEqual(code.loop_sum(2),3)
        self.assertEqual(code.loop_sum(100),5050)
```

Use a `for` loop!.

Python Strings

Python has a rich set of string handling methods (look at `http://docs.python.org/library/string.html`). In your second lab you will use the method `split`. For example:

```
>>> '1,2,3,4'.split(',')
['1', '2', '3', '4']
>>>
```

To concatenate strings you can use +

```
>>> 'hello ' + ' goodbye'
'hello  goodbye'
```

5 Hand In

As you can see `split` returns a list strings. Python has a built in `join` method. You job is to write your own. It should pass the tests:

```
    def test_myjoin(self):
        split1 = '1,2,3,4'.split(',')
        self.assertEqual(code.myjoin(split1,','),'1,2,3,4')
        self.assertEqual(
            code.myjoin('Hello and goodbye'.split('and'),'and'),
            'Hello and goodbye')
```

The following code will not work:

```
def myjoin(lst,sep):
  rstring = ''

  for i in lst:
      rstring = rstring + i + sep
  return rstring
```

Run the test and find out. The code nearly works. The loop `for i in l` iterates over every item in the list. The following hints might help:

- Given a list `lst` to get the list minus the last item use:

```
        lstminus_last = lst[0:len(lst)-1]
```

- To access the last item in a list you can use `lst[len(lst)-1]`.

Conclusion

This lab has only briefly scratched the surface of Python. Before the next lab you are strongly encouraged to go through one of the many Python tutorials on the web.