PYTHON
in the Enterprise

Plan

- Zajęcia – organizacja pracy…
- Wykłady – przegląd
- Organizacja Lab.

Tomasz Szumlak AGH-UST

Wydział Fizyki i Informatyki Stosowanej

06/03/2015

PLAN WYKŁADÓW

- ~~Wstęp organizacja pracy~~
- Używanie systemów kontroli wersji – GIT (wykład gościnny A. Dendek)
- **Unit Tests**
- Programowanie równoległe w Pythonie
- Programowanie sieciowe & „cloud computing”
- Elementy programowania GUI/MS Office/DB (**do wyboru**)
- Extending Python
- Django

Setting the scene

- Let's not get too **particular** about that
- If it works it works – leave it!
- Linux, Mac and Windows should be **equally fine** (at least for what we are going to do…)

- There are plenty of tools to **aid** the installation and setup process

- A package manager for Python – **Pip**
  - connects to PyPi repo
  - easy-peasy-japaneasey installation
  - usage
    - If not present in your system first do

      $> easy_install pip

      $> pip install XXX

Setting the scene

- **virtualenv** – really nice piece of software
  - Helps managing multiple medium/large projects
  - Imagine – setting up dependencies from scratch…
  - www.virtualenv.org

- **SVN**, **Git** – source control
  - For **medium/large projects** and **teams** of programmers working on them **concurrently**
  - We will have a dedicated lecture on that!

- Last but not least… - IDE (Interactive/Integrated Development Environment)
- It was primarily invented for RAD programming technique
- Now I will teach you otherwise…
- But the IDE is going to be useful nonetheless

Testing

- Each software application makes use of a numer of smaller components
- It combines their strength and shows that „**the whole is greater than the sum of its parts**”
- However if you provide shoddy db interface you will not mask it with the most beautiful front!

- Thus, great application is made with great components
    - If one of them fails the whole application is crap!

Unit Tests

- Yeah, give me an **abridged** version please… I'm busy…

- So, you want to make tests, this is what **you do**:
  - learn the application's **functionality**
  - and… **test it** on the **most basic level** by looking at each individual piece of code
  - this usually means **methods**

- Make this tests in isolated and controlled environment
  - I know exactly what the input is
  - check if the code response is **as expected**

- By testing the application at this granularity gives one a confidence that each piece of code will behave itself…
- Also, one can identify the potential problems and deal with them!

Unit Tests (2)

- Why should I break down the code I test…?
- What does it do for me?

- Can I learn something from that (too many/few lines)
- Any action after testing?

- One should treat testing like telling a story of one's code
  - be **nice** for yourself and others
  - develop a **naming convention** and stick to it!
  - use concise names for variables and classes, filenames

- Thinking about tests when writing the code can really help
  - **Test Driven Development** (TDD)

- Tests help in maintenance and upgrades

Unit Tests (3)

- Ok, yeah, but… **what do I test**…
- This is a fair question and there is no good answer to it…
- It is like trying to answer: „**What should I wear**?"
- It depends…

- **What tests to run depends completely on your application**!
- What you need to take care of is:
  - chose the granularity properly
  - test your functionality
  - do not test data types etc…, this may be a dead end and cause a lot of problems!

- Not too much help here, yay…?

Go and create!

calc.py

```
class MyCalc(object):
  def plus(self, x, y):
    return ( x + y )

if __name__ == '__main__':
  computer = MyCalc()
  result = computer.plus(2, 4)
  print result
```

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
[szumlat@localhost ~]$ python calc.py
6
[szumlat@localhost ~]$
```

Go and create!

calc.py | test_1.py

```python
import unittest

from calc import MyCalc

class TestMyCalc(unittest.TestCase):
  def setUp(self):
    self.computer = MyCalc()

  def test_add_method_return_correct_result(self):
    self.assertEqual( 6, self.computer.plus(2, 4) )

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

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
[szumlat@localhost ~]$ python test_1.py
F
======================================================================
FAIL: test_add_method_return_correct_result (__main__.TestMyCalc)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_1.py", line 10, in test_add_method_return_correct_result
    self.assertEqual( 6, self.computer.plus(2, 3) )
AssertionError: 6 != 5

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

FAILED (failures=1)
[szumlat@localhost ~]$
```

Go and create!

calc.py | test_1.py

```python
import unittest

from calc import MyCalc

class TestMyCalc(unittest.TestCase):
  def setUp(self):
    self.computer = MyCalc()

  def test_add_method_return_correct_result(self):
    self.assertEqual( 6, self.computer.plus(2, 4) )

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

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
[szumlat@localhost ~]$ python test_1.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
[szumlat@localhost ~]$
```

Go and create!

calc.py | test_1.py | calc_up.py

```
class MyCalc(object):
  def plus(self, x, y):
    return ( x + y )

if __name__ == '__main__':
  computer = MyCalc()
  result = computer.plus(2, 4)
  print result
```

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
>>> from calc import MyCalc
>>> computer = MyCalc()
>>> result = computer.plus( 'Aga','Gusia' )
>>> print result
AgaGusia
>>>
```

Dynamic binding in Python… this is how the **rocket fell**…

Hm! Upgrades...

calc.py | test_1.py | calc_up.py

```
class MyCalcXP(object):
  def plus(self, x, y):
    if type( x ) == int and type( y ) == int:
      return ( x + y )
    else:
      raise TypeError( " Invalid type: { } and { } ".format( type( x ), type
( y ) ) )
```

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
>>> from calc_up import MyCalcXP
>>> computer = MyCalcXP()
>>> result = computer.plus(2, 4)
>>> print result
6
>>> result = computer.plus('2', '4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "calc_up.py", line 6, in plus
    raise TypeError( " Invalid type: { } and { } ".format( type( x ), type(
y ) ) )
KeyError: ' '
>>>
```

```python
import unittest

from calc_up import MyCalcXP

class TestMyCalcXP(unittest.TestCase):
  def setUp(self):
    self.computer = MyCalcXP()

  def test_add_method_returns_correct_result(self):
    self.assertEqual("AgaGusia", self.computer.plus( "Aga", "Gusia" ) )

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

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
[szumlat@localhost ~]$ python test_2.py
E
======================================================================
ERROR: test_add_method_returns_correct_result (__main__.TestMyCalcXP)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_2.py", line 10, in test_add_method_returns_correct_result
    self.assertEqual("AgaGusia", self.computer.plus( "Aga", "Gusia" ) )
  File "/home/szumlat/calc_up.py", line 6, in plus
    raise TypeError( " Invalid type: { } and { } ".format( type( x ), type(
y ) ) )
KeyError: ' '

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

FAILED (errors=1)
[szumlat@localhost ~]$
```

```python
import unittest

from calc_up import MyCalcXP

class TestMyCalcXP(unittest.TestCase):
  def setUp(self):
    self.computer = MyCalcXP()

  def test_add_method_raises_typeerror_if_not_integer(self):
    self.assertRaises(TypeError, self.computer.plus, ( "Aga", "Gusia" ) )

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

Terminal - szumlat@localhost:~

File Edit View Terminal Go Help

```
[szumlat@localhost ~]$ python test_3.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
[szumlat@localhost ~]$
```