2I013 Groupe 5

Organisation python
Tests unitaires
Programmation Hardware

Nicolas Baskiotis
nicolas.baskiotis@lip6.fr

équipe MLIA, Laboratoire d'Informatique de Paris 6 (LIP6)
Université Pierre et Marie Curie (UPMC) - Sorbonne Universités

S2 (2017-2018)

Plan

Objectifs d'un module

Pourquoi :

- ne pas faire un gros fichier avec tout le code ?
- ne pas faire un répertoire avec différents fichiers contenant tout le code ? (mode script)

Objectifs d'un module

Pourquoi :

- ne pas faire un gros fichier avec tout le code ?
- ne pas faire un répertoire avec différents fichiers contenant tout le code ? (mode script)

Afin de

- organiser son code (un module responsable d'un aspect logiciel)
- débugger/tester plus facilement
- retrouver les fonctionnalités plus facilement
- travailler à plusieurs sans conflits
- distribuer son code
- factoriser le développement, ...

Terminologie

- Package : répertoire
- Module : fichier `.py`

Module en python : un objet comme un autre

Utilisation de `import`

`import module`	`from module \ import myf,myvar`	`from module import\ myf as f,myv as v`	`import module as m`
`module.myf()`	`myf()`	`f()`	`m.myf()`
`module.myvar`	`myvar`	`v`	`m.myvar`

```
>>> import math
>>> type(math) -> <type 'module'>
>>> mm = __import__('math') # autre facon d'importer
    Out[6]: <module 'math' (built-in)>
>>> mm.acos(1.) # utilisation comme import math as mm
>>> print(math.__dict__)
    {'radians': <built-in function radians>,
    'cos': <built-in function cos>,
    'frexp': <built-in function frexp>, ... }
>>>dir(mm)
    ['__doc__', '__loader__', '__name__',
    '__package__','__spec__', 'acos',
    'acosh', 'asin', 'asinh', ...]
>>> print(mm.__name__)
    'math'
```

(en fait, tout en python est objet)

Les importations en python

Deux manières de programmer en python :

- Script : pour du développement rapide, pour tester des fonctionnalités, pour utiliser principalement du code déjà existant, pour prototyper, ...
  commande : `python monscript.py` (ou shell interactif)
- Package/Module : pour du *vrai* développement, pour partager/diffuser son code, pour coder proprement ...
  commande : `python -m module/script.py` ou dans un fichier script `import module`

⇒ La seule grande différence : la gestion des `import`

Fonctionnement des `import` en python

- Un fichier `.py` est importable, un répertoire contenant un `__init__.py` est importable (python 3 : même sans `init`, le répertoire est importable).
- Mais doivent pouvoir être trouvé par python ⇒ se trouver dans le chemin défini par `PYTHONPATH`
- Et ce n'est pas le même en fonction de l'exécution script ou de l'importation du module ...

La variable PYTHONPATH

Par défaut, il contient les répertoires :

- système : `/usr/lib/python2.7` (ou python3.6), etc
- des paquets installés au niveau du système : `/usr/lib/python2.7/sites-package`
- des paquets installés localement à l'utilisateur : `~/.local/lib/python2.7/sites-packages`
- le répertoire courant du script lancé (placé en tout premier)

Pour l'installation des paquets : commande `pip`

- `pip install monpackage` : installation système du paquet `monpackage` (à partir du python repository)
- `pip install monpackage --user` : installation sur le compte utilisateur
- `pip install .`, `pip install repertoire` : installation d'un paquet local qui se trouve soit dans le répertoire courant, soit dans le répertoire passé en paramètre
- `pip install -e repertoire` : installation en lien symbolique. Le paquet n'est pas copié dans le répertoire `site-packages`, un lien symbolique est simplement créé : très utile en dév.

Organisation des fichiers pour du script

Exemple de répertoire

```
~/monprojet/
    outils.py
    script.py
```

---

fichier outils.py :

```
import sys
class Outil(object):
  def __init__(self):
    self.moi = self.__class__
    self.path = sys.path
```

---

fichier script.py:

```
from outils import Outil
  outil = Outil()
  print("outil",outil.moi,
        outil.path)
```

Exemples d'utilisation (python2 ou python3)

```
~/monprojet$ python script.py
```

⇒ OK

```
outil <class 'outils.Outil'> ~/monprojet
```

```
~$ python ~/monprojet/script.py
```

⇒ OK

```
outil <class 'outils.Outil'> ~/monprojet
```

Différents types d'import

```
#Import relatif implicite
from outils import Outil
#Import relatif explicite
from .outils import Outil
# Import absolu
# (module dans le python path)
from module import fonction
```

Paquets et Modules : Python 2

Exemple de répertoire

```
~/monprojet/
          script.py
          module/
            __init__.py
            outils.py
            scriptoutils.py
```

---

fichier scriptoutils.py:

```
from outils import Outil
  outil = Outil()
  print("outil",outil.moi,outil.path)
```

---

fichier __init__.py:

```
from outils import Outil
## import relatif implicite
```

---

fichier script.py :

```
from module import Outil
## import absolu
outil = Outil()
print("outil",outil.moi,outil.path)
```

Exemples

```
~/monprojet/module$ python
scriptoutils.py
```

⇒ OK (import relatif implicite)

```
 outil <class 'outils.Outil'>
~/monprojet
```

```
~$ python ~/monprojet/script.py
```

⇒ OK

```
 outil <class 'outils.Outil'>
~/monprojet
```

En python 3 : Ne marche plus !!!

Utilisation obligatoire des import relatifs explicites ou des imports absolus dans les modules !

⇒ plus possible d'avoir des scripts de tests dans les modules.

Paquets et Modules : Python 3

Exemple de répertoire

```
~/monprojet/
  script.py
  module/
    __init__.py
    outils.py
    autre_outils.py
    sousmodule/
        __init__.py
        sousoutils.py
    autresousmodule/
          __init__.py
          autresousoutils.py
-----------------------------
fichiers xxxoutils.py :
 class XxxOutil(object):
    ...
fichiers script.py :
  from module import Outil,
    SousOutil, AutreSousOutil
```

Fichiers `__init__.py`

```
.../module/autresousmodule/__init__.py:
  from .autresousoutils import AutreSousOut
.../module/sousmodule/__init__.py:
  from .sousoutils import SousOutil
  from ..autresousmodule import AutreSousOu
.../module/__init__.py:
  from .outils import Outil
  from .autre_outils import AutreOutil
  from .sousmodule import SousOutil
  from ..autresousmoudule import AutreSous
```

Autre solution (absolu)

```
.../module/autresousmodule/__init__.py:
 from module.autresousmodule.autresousouti
    import AutreSousOutil
.../module/sousmodule/__init__.py:
 from module.sousmodule.sousoutils
    import SousOutil
  from module.autresousmodule import Autre
```

Résumé

Différence entre Python 2 et 3

- Import chemin absolu : la même chose
- Import chemin relatif : différent !
  - Python 2 : import relatif *par défaut*
  - Python 3 : import relatif doit être explicite

  ⇒ `from .mafonction import fonction`
  - ne fonctionne que pour l'import de package, pas le script . . .
- Toujours mettre `from __future__ import absolute_import` au début de tous vos fichiers (en python 2)
  ⇒ assure la compatibilité de Python 2 vers Python 3.

Attention !

- Dans tout fichier importé, le fichier est exécuté en totalité sauf la partie `if __name__==__main__`

⇒ Jamais de code exécutable dans un fichier importé par `__init__.py`

Problème : les fichiers scripts dans les modules

```
~/monprojet/module/
    __init__.py
    script.py
    outils.py
    autre_outils.py
    sousmodule/
        __init__.py
        sousoutils.py
    autresousmodule/
          __init__.py
          autresousoutils.py
```

Dans `script.py` :

```
from module.outils import Outil
#ou
from .outils import Outil
# Marche en execution module :
$ python -m module.script => OK
# Mais ne marche pas en execution script !
$ python module/script.py => ERROR
```

Solution :

Tous les fichiers scripts à la racine du répertoire (`~/monprojet`)!

Autre solution : tricher

Possible de changer dynamiquement le Python Path : `sys.path`

```
~/monprojet/
  module/
    __init__.py
    script.py
    outils.py
    autre_outils.py
    sousmodule/
      __init__.py
      sousoutils.py
    autresousmodule/
      __init__.py
      autresousoutils.py
  script/
    script.py
```

Dans script.py :

```
import sys
import os
# Chemin du module
print(os.path.abspath(__file__))
# -> ~/monprojet/script/script.py
# Repertoire contenant le module script.py
print(os.path.dirname(os.path.abspath(__file_
# -> ~/monprojet/script/
#Repertoire parent du repertoire de script.py
print(os.path.dirname(
  os.path.dirname(os.path.abspath(__file__)))
# -> ~/monprojet/
## Ajout a Python Path du repertoire contenar
sys.path.append(os.path.dirname(
  os.path.dirname(os.path.abspath(__file__)))

$ python script/script.py -> ok
```

A utiliser avec parcimonie !!

Plan

Objectifs

S'assurer que

- les bouts de code développés fonctionnent
- il n'y a pas d'introduction de bug au cours du développement
- l'intégration n'a pas de conflit, rétrocompatibilité
- les refactorisations/optimisations n'ont pas d'impact sur le code

Ne permet pas de

- Débusquer tous les bugs !

Un test unitaire

- doit être unitaire ! (une méthode à la fois)
- si trop complexe à tester, le code est mal fait !
- être codé tout de suite après le code de la fonctionnalités (ou avant !)

En python : `unittest`

Framework de test unitaire : il permet

- de façon simple de réaliser des tests unitaires (comparable a `JUnit`)
- d'automatiser un certain nombre de tâches
- lever et détecter les tests échoués
- de tester un nombre réduit de sous-modules
- de séparer le test du code du paquet

Exemple simple :

```
unit_test.py :
import unittest
class SimplisticTest(unittest.TestCase):
  def test(self):
    self.asserTrue(True)

if __name__ == '__main__':
  unittest.main()
----------------------
$ python unit_test.py  -> Ran 1 test in 0.000s OK
```

Méthodes utiles (et plus)

```
assertEqual(a, b)  a == b
assertNotEqual(a, b)  a != b
assertTrue(x)  bool(x) is True
assertFalse(x)  bool(x) is False
assertIs(a, b)  a is b
assertIsNot(a, b)  a is not b
assertIsNone(x)  x is None
assertIsNotNone(x)  x is not None
assertIn(a, b)  a in b
assertNotIn(a, b)  a not in b
assertIsInstance(a, b)  isinstance(a, b)
assertNotIsInstance(a, b)
assertAlmostEqual(a, b)  round(a-b, 7) == 0
assertNotAlmostEqual(a, b)  round(a-b, 7) !=
assertGreater(a, b)  a > b
assertGreaterEqual(a, b)  a >= b
assertLess(a, b)  a < b
assertLessEqual(a, b)  a <= b
assertRegex(s, r)  r.search(s)
assertCountEqual(a, b)  a and b have the same elements in the same num
assertSequenceEqual(a, b)  sequences
assertListEqual(a, b)  lists
assertTupleEqual(a, b)  tuples
assertDictEqual(a, b)  dicts
```

Exemple complet

```
projet/
    geo/
      geo2d.py
      geo3d.py
      __init__.py
```

```
geo3d.py :
from .geo2d import Vec2D
  class Vec3D(Vec2D):
    def __init__(self,x,y,z):
      super(Vec3D,self).__init__(x,y)
      self.z = x,y,z
    def add(self,v):
      super(Vec3D,self).add(v)
      self.z+=v.z
    def mul(self,v):
      super(Vec3D,self).mul(v)
      self.z *= v.z
```

```
geo2d.py :
  class Vec2D(object):
    def __init__(self, x, y):
      self.x, self.y = x, y
    def add(self, v):
      self.x += v.x
      self.y += v.y
    def mul(self, v):
      self.x *= v.x
      self.y *= v.y

  class PolyLine2D(object):
    def __init__(self):
      self.sommets = []
    def add(self,p):
      self.sommets.append(p)
    def len(self):
      return len(self.sommets)
```

Que tester ?

Exemple complet

```
projet/
.git
geo/
  geo2d.py
  geo3d.py
test/
  __init__.py
  test_geo2d.py
  test_geo3d.py
-----------------------
test_geo2d.py :
import unittest
from geo import Vec2D,PolyLine2D
class TestVec2D(unittest.TestCase):
    def setUp(self):
        self.p = Vec2D(1,1)
        self.deux = Vec2D(2,2)
    def test_point(self):
        self.assertEqual(self.p.x,1)
        self.assertEqual(self.p.y,1)
    def test_add(self):
        self.p.add(self.p)
        self.assertEqual(self.p.x,2)
        self.assertEqual(self.p.y,2)
    def test_mul(self):
        self.p.mul(self.deux)
        self.assertEqual(self.p.x, 2)
        self.assertEqual(self.p.y, 2)

class TestPolyLine2D(unittest.TestCas
    def setUp(self):
        self.line = PolyLine2D()
        self.line.add(Vec2D(1,1))
        self.line.add(Vec2D(2,2))
    def test_len(self):
        self.assertEqual(
          self.line.len(),2)

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

Exemple complet

```
projet/
.git
geo/
  geo2d.py
  geo3d.py
test/
  __init__.py
  test_geo2d.py
  test_geo3d.py
------------------------
```

```
import unittest
from geo import Vec3D
class TestVec3D(unittest.TestCase):
    def setUp(self):
        self.p = Vec3D(1, 1,2 )
        self.deux = Vec3D(2, 2,1)
    def test_point(self):
        ...
if __name__ == '__main__':
  unittest.main()
```

```
#Pour tout tester :
~/projet$ python -m unittest discover test -v
#Pour tester un fichier :
~/projet$ python -m unittest test.test_geo2d -v
#Pour tester une classe de test :
~/projet$ python -m unittest test.test_geo2d.TestVec2D -v
```

Pas besoin de modifier le Python Path, `unittest` se charge de tout mettre comme il faut !