

# 2I013 Groupe 5

## Module Python

## Organisation du code

## Tests unitaires

Nicolas Baskiotis

`nicolas.baskiotis@lip6.fr`

équipe MLIA, Laboratoire d'Informatique de Paris 6 (LIP6)  
Sorbonne Université

S2 (2018-2019)

# Plan

**Package/Module Python**

Conventions Python et documentation

Tests unitaires

# 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)
- débbugger/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`

# Installer un package en Python

## 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.

# Module en python : un objet comme un autre

## Utilisation de `import`

<code>import module</code>	<code>from module \ import myf,myvar</code>	<code>from module import\ myf as f,myv as v</code>	<code>import module as m</code>
<code>module.myf()</code> <code>module.myvar</code>	<code>myf()</code> <code>myvar</code>	<code>f()</code> <code>v</code>	<code>m.myf()</code> <code>m.myvar</code>

```
>>> 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 (module)
- un répertoire contenant un `__init__.py` est importable (package)
- `__init__.py` définit le comportement de l'import
- 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 ...

# Importation de module

## Lorsqu'on importe un module :

- Tout le fichier est exécuté : les variables, les fonctions et les classes définies dans le module sont donc disponibles, mais les lignes de scripts sont également exécutées !
- Sauf si : `if __name__=="__main__":`

## Exemple

```
def myf():  
    return 1  
a = myf()  
print("Le_résultat_de_myf_est",a)  
print("Toutes_ces_lignes_sont_exécutées_avec_import_fichier")  
if __name__=="__main__":  
    print("ceci_n'est_pas_exécuter_par_import")  
    print("mais_uniquement_par_python_fichier.py")
```



# Le fichier `__init__.py`

## Sert à initialiser le package lors de l'import

- C'est un fichier python comme les autres
- Il peut contenir des variables, des fonctions, du code ...
- Lors de l'import du package, c'est ce fichier qui est exécuté !

⇒ Tout ce qui est disponible après l'import est spécifié par ce fichier

- Une variable `__all__` peut être définie pour spécifier le comportement de

```
from package import *
~/projet2I013
__init__.py
from .robot import Robot
def projet():
    return "c'est_mon_projet"
__all__=["Robot"]
projet()
robot.py
-> class Robot
vieuxrobot.py
-> class VieuxRobot
```

```
#import execute __init__.py
import projet2I013
    c'est_mon_projet
#Robot, _projet_accessiblent_directemen
projet2I013.Robot()
projet()
#module_vieuxrobot_aussi
projet2I013.vieuxrobot.VieuxRobot()
-----
#par_contre, _que_Robot_dans_ce_cas
from _projet2I013 import *
__c'est mon projet
Robot() #OK
projet() #KO
```

# 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

```
~/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
```

# Organisation des fichiers pour des paquets

## Import relatif implicite impossible !!

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

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

```
~/monprojet/arene      ## Ne marche plus !!!
__init__.py           python ~/monprojet/script.py
arene.py               ## mais ceci marche !
obstacle.py           python -c "from _projet2I013_import_script"
~/monprojet/
    script.py
-----
script.py:
    #import arene NE MARCHE PAS
    import .arene
```

# 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 `__init__.py`

```
.../module/autresousmodule/__init__.py:  
    from .autresousoutils import AutreSousOutils  
.../module/sousmodule/__init__.py:  
    from .sousoutils import SousOutil  
    from ..autresousmodule import AutreSousOutils  
.../module/__init__.py:  
    from .outils import Outil  
    from .autre_outils import AutreOutil  
    from .sousmodule import SousOutil  
    from ..autresousmodule import AutreSousOutils
```

## Autre solution (absolu)

```
-----  
fichiers xxxoutils.py :  
    class XxxOutil(object):  
        ...  
fichiers script.py :  
    from module import Outil,  
        SousOutil, AutreSousOutil
```

```
.../module/autresousmodule/__init__.py:  
    from module.autresousmodule.autresousoutils  
        import AutreSousOutils  
.../module/sousmodule/__init__.py:  
    from module.sousmodule.sousoutils  
        import SousOutil  
    from module.autresousmodule import AutreSousOutils
```

# 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
```

# La variable PYTHONPATH

## Par défaut, il contient les répertoires :

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

## Autre solution : tricher

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

```
~/monprojet/
module/
  __init__.py
  script.py
  outils.py
  autre_outils.py
sousmodule/
  __init__.py
  sousoutils.py
autresousmodule/
  __init__.py
  autresousoutils.py
script/
  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
# -> ~/monprojet/
sys.path.insert(0, os.path.abspath(
    os.path.join(
        os.path.dirname(__file__), '..')))

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

A utiliser avec parcimonie !!

# Plan

Package/Module Python

**Conventions Python et documentation**

Tests unitaires



# Quelques conventions

## Convention syntaxe

- Nom des paquets/modules en miniscule !
- Nom des classes : Camel Case (NomDeLaClasse)
- nom des variables en minuscule
- séparation par des \_

⇒ Pas de confusion entre modules/paquets et classes !

Par ailleurs, plusieurs classes dans un même fichier (contrairement à Java)

Peu de méthodes statiques ! En général, ça ne sert à rien, autant faire une fonction.

# Docstring

- Docstring : manière d'écrire de la doc en python : `""" DOC """`
- Pas de commentaire entre `"""`, utiliser plutôt `#`
- Générateur automatique de documentation à partir de ce format : Pydoc, Sphinx

## Example

```
class Arene:
    """ L'arene contient un robot et des obstacles
        :param x: longueur de l'arene
        :param y: largeur de l'arene
        ...
    """
    def __init__(self,x,y): pass
    def add_obstacle(self,o):
        """ On peut ajouter un obstacle a l'arene
            :param o: l'objet a ajouter
            :returns: rien, changement inplace
        """
```

# Plan

Package/Module Python

Conventions Python et documentation

**Tests unitaires**

## 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.assertTrue(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) != 0
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.TestCase):  
    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 !**

# Idée de l'API Robot

```
class Robot2I013(object):
    def set_led(self, led, red = 0, green = 0, blue = 0):
        """ Allume une led. """
    def get_voltage(self):
        """ get the battery voltage """
    def set_motor_dps(self, port, dps):
        """ Fixe la vitesse d'un moteur en nombre de degres par second
        :port: une constante moteur, MOTOR_LEFT ou MOTOR_RIGHT (ou le
        :dps: la vitesse cible en nombre de degres par seconde"""
    def get_motor_position(self):
        """ Lit les etats des moteurs en degre.
        :return: couple du degre de rotation des moteurs"""
    def offset_motor_encoder(self, port, offset):
        """ Fixe l'offset des moteurs (en degres) (permet par exemple
        du moteur gauche avec offset_motor_encode(self.MOTOR_LEFT, self
        :port: un des deux moteurs MOTOR_LEFT ou MOTOR_RIGHT (ou les d
        :offset: l'offset de decalage en degre.
        Zero the encoder by offsetting it by the current position
        """
    def get_distance(self):
        """ Lit le capteur de distance (en mm).
        :returns: entier distance en millimetre.
            1. L'intervalle est de **5-8,000** millimeters.
            2. Lorsque la valeur est en dehors de l'intervalle, le ret
```