

# 2IN013 Groupe 1

## 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 (2019-2020)

# Plan

## Package/Module Python

## Conventions Python et documentation

## Tests unitaires

## Sérialisation en python

## Design Patterns

## Application au projet

# 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_resultat_de_myf_est",a)  
print("Toutes_ces_lignes_sont_executees_avec_import_fichier")  
if __name__=="__main__":  
    print("ceci_n'est_pas_executer_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_accessiblement_directement
```

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

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

# 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

Sérialisation en python

Design Patterns

Application au projet



# 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

## Exemple

```
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**

Sérialisation en python

Design Patterns

Application au projet

# 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
```

# Plan

Package/Module Python

Conventions Python et documentation

Tests unitaires

**Sérialisation en python**

Design Patterns

Application au projet

# Sérialisation

## Principe

- Pouvoir stocker/transférer un objet ...
  - ... et pouvoir le reconstruire possiblement dans un autre environnement (autre système d'exploitation, autres versions, ...)
  - L'objet reconstruit doit être un **clone** sémantique de l'objet initial
- ⇒ Transformer un objet en une séquence de bits (sérialisation) et pouvoir reconstruire l'objet à partir de cette séquence de bits (désérialisation).

# En python

## Module Pickle

- Méthode native de python
- Adapté pour des objets complexes (composés d'autres objets, références récursives, ...)
- différents *protocoles* :
  - 0 : format *human-readable*
  - 2 : binaire, par défaut en python 2
  - 3 : binaire compressé, par défaut en python 3, non rétro-compatible
- Avantages : simple à utiliser, sérialise beaucoup d'objets (structures de base mais aussi fonctions, classes)
- Inconvénients : parfois lourd, propre à python.

## Exemple

```
import pickle
with open('data.pkl','wb') as f:
    pickle.dump(monObjet,f)
with open('data.pkl','rb') as f:
    monObjet = pickle.load(f)
```

# Qu'est ce qu'on peut picker ?

## Les objets construits sur les types suivants :

- Booléen
- entier, réel, ...
- string, byte
- tuple, liste, dictionnaire
- fonction, classe
- objet dont le dictionnaire (les variables) est pickable

⇒ à peu près tout ...

## Pourquoi ne pas utiliser Pickle ?

- Souvent lourd et lent, surtout pour les objets très verbeux
- Pas sécurisé
- Pas transférable à d'autres langages

# JSON : JavaScript Object Notation

## Format JSON

- Format de fichiers ouvert, en texte clair, standard, très répandu
- Encodage par le biais de dictionnaires clé-valeur qui peuvent contenir les types natifs suivants :
  - `Nombre` : entier ou réel
  - `String` : séquence de caractère unicode
  - `Boolean` : `true` ou `false`
  - `Array` : une séquence ordonnée de valeurs, les types peuvent être mixés
  - `Object` (ou dictionnaire) : ensemble non ordonné de couples clé/valeur

## Exemple

```
{ "type" : "Arene",  
  "dimension" : [100, 200],  
  "objets" : {  
    "premier": { "type" : "Cube", "position" : [[0, 0],[0, 1]]},  
    "second": { "type" : "Robot", "position" : [ 0.5, 0.5 ] }  
  }  
}
```

# JSON et Python

## Module json natif mais n'encode que les types de base

Types: dict, list, string, int, long, boolean  
ne permet pas d'encoder nativement un objet !!

```
>>>import json
#json.dumps -> string, json.dump -> fichier
>>>json.dumps(['foo',{'bar':('baz', None, 1.0, 2)}])
'["foo",_{"bar":_["baz",_null,_1.0,_2]}]'
```

```
>>>json.loads('["foo",{"bar":["baz",null,1.0,2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

## Pour un objet Python

- Tous les attributs de l'objet sont dans la variable `__dict__`
- la classe d'un objet est dans la variable `__class__.__name__`

```
class A(object) :
    def __init__(self):
        self.a=1; self.b = "c'est_moi";self.c =[1,True,"dix"]
print(A().__dict__, A().__class__.__name__)
-> {'a': 1, 'b': "c'est_moi", 'c': [1, True, 'dix']}, 'A'
```



# Solution simple (mais incomplète)

```
import json
class A(object):
    def __init__(self, a=1, b="moi", c=[1, True, "dix"]):
        self.a, self.b, self.c = a, b, c
a = A()
aserial = json.dumps(a.__dict__)
-> '{"b": "moi", "c": [1, true, "dix"], "a": 1}'
## **kwargs permet de passer le dictionnaire kwargs comme argument
newa = A(**json.loads(aserial))

def myencoder(obj):
    dic = dict(obj.__dict__)
    dic.update({"__class__": obj.__class__.__name__})
    return json.dumps(dic)
def mydecoder(s):
    dic = json.loads(s)
    cls = dic.pop("__class__")
    return eval(cls)(**dic)
mydecoder(myencoder(a))
-> <__main__.A at 0x7f8ec872f668>
```

# Problème : objet composé d'autres objets ...

```
class B(object):
    def __init__(self, autre):
        self.a = A()
        self.autre = autre
```

```
b=B(a)
```

```
myencoder(b)
```

```
-> TypeError: <__main__.A object at 0x7f8ec86c4b70> is not JSON serial
```

## Solution : paramètres default/object\_hook (ou hériter de JSONEncoder et JSONDecoder)

- `default(obj)` : méthode qui encode un objet; si l'objet n'est pas natif, cette méthode est appelée, elle doit sérialiser son dictionnaire et ajouter le nom de la classe.
- `object_hook(s)` : méthode qui est appelée avec chaque dictionnaire désérialisé avant le retour.

# Solution complète

```
import json

class A(object):
    def __init__(self, a=1, b="moi", c=[1, True, "dix"], d={1:2, "a":True}):
        self.a, self.b, self.c = a, b, c

class B(object):
    def __init__(self, autre):
        self.autre = autre

def my_enc(obj):
    dic = dict(obj.__dict__)
    dic.update({"__class__": obj.__class__.__name__})
    return dic

def my_hook(dic):
    if "__class__" in dic:
        cls = dic.pop("__class__")
        return eval(cls)(**dic)
    return dic

b = B(A())
bserial = json.dumps(b, default=my_enc)
-> '{"__class__": "B", "autre": {"b": "moi", "a": 1, "c": [1, true,
        "dix"], "__class__": "A"}}'
b = json.loads(bserial, object_hook=my_hook)
```

# Plan

Package/Module Python

Conventions Python et documentation

Tests unitaires

Sérialisation en python

**Design Patterns**

Application au projet

# Design Patterns

## Someone has already solved your problems

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (C. Alexander)

## Pourquoi ?

- Solutions propres, cohérentes et saines
- Langage commun entre programmeurs
- C'est pas seulement un nom, mais une caractérisation du problème, des contraintes,...
- Pas du code/solution pratique, mais une solution générique à un problème de design.

## Un très bon livre :

Head First Design Patterns, E. Freeman, E. Freeman, K. Sierra, B. Bates, Oreilly

# Design Patterns

## Quelques Principes

- Surtout pour les langages fortement typés, structurés (**Java** par exemple)
- Identifier les aspects de votre programme qui peuvent varier/évoluer et les séparer de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible) !

En avez-vous déjà vu ?

# Design Patterns

## Quelques Principes

- Surtout pour les langages fortement typés, structurés (**Java** par exemple)
- Identifier les aspects de votre programme qui peuvent varier/évoluer et les séparer de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible) !

En avez-vous déjà vu ?

## 3 grandes classes

- *Creational* : Comment créer des objets
- *Structural* : Comment interconnecter des objets
- *Behavioral* : Comment faire une opération donnée





# Creational patterns

En python, il n'y en a pas vraiment (sauf le singleton). Pour créer un objet d'une certaine manière, il suffit de faire une fonction.

```
def get_random_vec(x,y):  
    return Vector2D.create_random(x,y)  
def from_polar(x,y):  
    return Vector2D.from_polar(0,2)  
def from_cartesien(x,y):  
    return Vector2D(x,y)  
def get_null():  
    return Vector2D()  
...
```

# Quelques caractéristiques de Python

## Dans un objet :

- `def __init__(self, *args, **kwargs)`
  - `args` : arguments non nommés (`args[0]`)
  - `kwargs` : arguments nommés (`kwargs[ 'nom' ]`)
- `__getattr__(self, name)` : appelé quand `name` n'est pas trouvé dans l'objet
- `__getattribute__(self, name)` : appelé pour toute recherche de `name`
- Propriété : pour interroger de manière dynamique
 

```
class MyClass:
    @property
    def name(self): return self._name
    @name.setter
    def name(self, value): self._name = value
    ...
a = MyClass()
print(a.name) # plutot que a.name()
a.name="toto" #plutot que a.set_name("toto")
```

En python, pas d'erreur de typage, uniquement à l'exécution !

# Python : Duck Typing

*If it looks like a duck and quacks like a duck, it's a duck!*

## Typage dynamique

- La sémantique de l'objet (son type) est déterminée par l'ensemble de ses méthodes et attributs, dans un contexte donné
- Contrairement au typage nominatif où la sémantique est définie explicitement.

## Concrètement

```

Class Duck:
    def quack(self):
        print("Quack")
Class Personne:
    def parler(self):
        print("Je parle")
donald = Duck()
moi = Personne()
autre = "un_canard"
try:
    donald.duck()
    moi.duck()
    autre.duck()
except AttributeError:
    print("c'est pas un_canard")
  
```

# Adapteur : et si je veux que ce soit un canard ?

- Il suffit d'y ajouter une méthode qui le fait se comporter comme un canard.
- Toutes les autres méthodes doivent être disponibles !

```
class PersonneAdapter:
    def __init__(self, obj):
        self._obj = obj
    def __getattr__(self, attr):
        if attr == "duck":
            return self.parler()
        return attr(self._obj, attr)
```

```
moi = PersonneAdapter(Personne())
moi.duck()
```

# Iterator

*Pouvoir parcourir une liste d'éléments sans connaître l'organisation interne des éléments*

## Un itérateur est un objet qui dispose

- d'une méthode `__iter__(self)` qui renvoie l'itérateur
- d'une méthode `next(self)` qui renvoie la prochaine valeur ou lève une exception `StopIteration`

Un itérateur peut être renvoyé par une fonction grâce à `yield`.

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high
    def __iter__(self):
        return self
    def next(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

def counter(low, high):
    current = low
    while current <= high:
        yield current
        current += 1

for c in counter(3, 8):
    print(c)
```

# Chain of responsibility

*Chaque bout de code ne doit faire qu'une et une seule chose*

Quand beaucoup d'actions complexes doivent être appliquées, il vaut mieux multiplier des petites fonctions en charge de chaque action que faire une unique grosse fonction.

```
class ContentFilter(object):
    def __init__(self, filters=None):
        self._filters = list()
        if filters is not None:
            self._filters += filters

    def filter(self, content):
        for filter in self._filters:
            content = filter(content)
        return content

filter = ContentFilter([offensive_filter, ads_filter, video_filter])
filtered_content = filter.filter(content)
```

# State (ou Proxy dans la version simple)

Changer le comportement d'une fonction en fonction de l'état interne du système.

Proxy quand il n'y a pas d'état interne.

```
class Implem1:
    def f(self):
        print("Je_suis_f")
    def g(self):
        print("Je_suis_g")
    def h(self):
        print("Je_suis_h")

    class Implem2:
        def f(self):
            print("Je_suis_toujours_f.")
        def g(self):
            print("Je_suis_toujours_g.")
        def h(self):
            print("Je_suis_toujours_h.")
```

```
class State_d:
    def __init__(self, imp):
        self._implem = imp
    def changeImp(self, newImp):
        self._implem = newImp
    def __getattr__(self, name):
        return getattr(self._implem, name)

def run(b):
    b.f()
    b.g()
    b.h()
b = State_d(Implem1())
run(b)
b.changeImp(Implem2())
run(b)
```

# Decorator : très similaire à Proxy et Adaptor

*Comment ajouter des fonctionnalités de manière dynamique à un objet*

## Exemple

```
class Decorator:
    def __init__(self, robot):
        self.state = robot
    def __getattr__(self, attr):
        return getattr(self.robot, attr)
class Avance(Decorator):
    def __init__(self, state):
        Decorator.__init__(self, robot)
    def avance(self):
        return ...
class Tourne(Decorator):
    def __init__(self, state):
        Decorator.__init__(self, robot)
    def tourne(self):
        return ...
robot = Tourne(Avance(robot)) # tout dans robot accessible
# donne acces a robot.tourne() et robot.avance()
```



# Decorator : peut changer le comportement d'une fonction

## Exemple : modifier la manière d'avancer

```
class AvancerAuPas(Decorator):  
    def aupas(self):  
        return ...  
    def avancer(self):  
        if (condition):  
            return self.aupas()  
        return self.avancer()  
robot = AvancerAuPas(Avancer(robot))
```

# Strategy

Le pattern Strategy permet de faire varier l'algorithme de manière dynamique et indépendante :

- Lorsqu'on a besoin de différentes variantes d'un algorithme.
- Lorsqu'on définit beaucoup de comportements à utiliser selon certaines situations

```
class StrategyExample:
    def __init__(self, func):
        self.update = func
        @property
        def name(self):
            if hasattr(self.func, "name"):
                return self.func.name
            return self.func.__name__
    def avanceVite(state):
        return ...
    def avanceLentement(state):
        return ...

stratVite = StrategyExample(avanceVite)
stratVite.name = "vite"
stratLent = StrategyExample(avanceLentement)
stratLent.name = "lent"

class Robot:
    def __init__(self, strat):
        self.strat = strat
        self.state = ...
    def update(self):
        return self.strat.update(
            self.state)

def strat_complexe(state):
    if ...:
        return stratLent(state)
    return stratVite(state)
```

# Plan

Package/Module Python

Conventions Python et documentation

Tests unitaires

Sérialisation en python

Design Patterns

**Application au projet**

# Interface graphique

Ce qu'il ne faut pas faire !!

```
class MaFenetre:
    ....
    def dessine(self, arene):
        for obj in arene:
            if obj == OBSTACLE: ...
            if obj == ROBOT : ...
        ....
```

Mais plutôt :

```
class MaFenetre:
    ....
    def dessine(self, arene):
        for obj in arene:
            self.draw(obj.x, obj.y, get_im
def get_image(objet):
    if objet == ... : return ...
```

# Controleur

## Controleur synchrone

- A chaque commande, attente de la fin de la commande avant le retour de la fonction
- En attendant, tout est bloqué !

Pas réactif, chaque commande prend un temps indéterminé, appel bloquant ...

## Exemple très mauvais

```
class Contrôler:  
    def __init__(self, robot): self.robot = robot  
    def update(self):  
        for i in range(100): self.robot.avance(1)  
        self.robot.tourne(90)  
        for i in range(100): self.robot.avance(1)
```

# Contrôleur

## Contrôleur asynchrone

- A chaque commande, on envoie une intention
- On ne sait pas ce qui a été exécuté !!
- Il faut contrôler l'état à chaque appel, ou possibilité de *callback* : la fonction callback est appelée à la fin de l'exécution de la commande

Réactif, plus compliqué à mettre en œuvre.

## Exemple moyen

```
class Contrôler:
    def __init__(self, robot): self.robot = robot
    def update(self):
        if (self.robot.etat.x-self.last_x)<...:
            self.robot.set_vitesse(1)
        else:
            self.robot.tourne(90)
```

Problème : devient très vite compliqué de tout coder dans `update`

# Contrôleur

## Contrôleur asynchrone

- A chaque commande, on envoie une intention
- On ne sait pas ce qui a été exécuté !!
- Il faut contrôler l'état à chaque appel, ou possibilité de *callback* : la fonction callback est appelée à la fin de l'exécution de la commande

Réactif, plus compliqué à mettre en œuvre.

## Exemple correct

```
class Contrôler:
    def __init__(self, robot): self.robot = robot
    def update(self):
        if (self.robot.etat.x-self.last_x)<...:
            avance_strategy(self.robot)
        else:
            tourne_strategy(self.robot)
```

Ou tout autre organisation mais qui fait appel à des fonctions indépendantes de petites stratégies élémentaires.

# Stratégie stateless (sans état)

Afin de ne pas compliquer de trop le code, il est nécessaire :

- de coder une classe état qui renseigne l'état du robot : temps depuis le dernier `update`, position des roues, ...
- les stratégies prennent en paramètre l'état qui leur permet d'être indépendantes de tout autre contexte
- une stratégie est un objet simple !

## Organisation générale

```
robot, arene = Robot(), Arene()
controleur = Controleur(robot)
visu = Fenetre(arene)
```

```
def run(controleur):
    while not controleur.stop():
        controleur.update()
        |---> enregistre le temps actuel, le dt par
                rapport au dernier appel, donne un ordre au robot
        time.sleep(DT)
```