

# Projet Foot 2I013

Nicolas Baskiotis

`nicolas.baskiotis@lip6.fr`

`http://webia.lip6.fr/~baskiotisn`

`http:`

`//github.com/baskiotisn/SoccerSimulator-2017`

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

S2 (2017-2018)

# Plan

**Organisation de vos fichiers**

Design Patterns

# Comment organiser vos fichiers ?

Garder votre code séparé du module `soccersimulator` !!!

## Exemple d'organisation

```
2I013/  
    SoccerSimulator-2017/                # git du prof  
        .git  
        setup.py  
        soccersimulator/  
        ...  
    MonGit/                               # votre git  
        mes fichiers
```

## Pourquoi ?

# Comment organiser vos fichiers ?

Garder votre code séparé du module `soccersimulator` !!!

## Exemple d'organisation

```
2I013/  
    SoccerSimulator-2017/                # git du prof  
        .git  
        setup.py  
        soccersimulator/  
        ...  
    MonGit/                               # votre git  
        mes fichiers
```

## Pourquoi ?

- Vous ne devez pas changer le code du module !
- Votre code doit pouvoir être distribué !
- Votre code est indépendant du module.
- Le module change (souvent), ne pas mélanger les versions.

# Module en python : un objet comme un autre

## Utilisation de `import`

<code>import module</code>	<code>from module \</code> <code>import myf,myvar</code>	<code>from module import\</code> <code>myf as f,myv as v</code>	<code>import module</code> <code>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, un répertoire contenant un `__init__.py` est importable (python 3 : même sans `init`, le répertoire est importable).
- Mais doivent pouvoir être trouvés 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 dev.

# Organisation des fichiers pour du script

## Exemple de répertoire

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

```
-----  
fichier outils.py :  
import sys  
class Util(object):  
    def __init__(self):  
        self.moi = self.__class__  
        self.path = sys.path  
-----
```

```
fichier script.py:  
from outils import Util  
    outil = Util()  
    print("outil",outil.moi,  
        outil.path)
```

## Exemples d'utilisation (python2 ou python3)

```
~/monprojet$ python script.py  
⇒ OK  
outil <class 'outils.Util'> ~/monprojet
```

```
~$ python ~/monprojet/script.py  
⇒ OK  
outil <class 'outils.Util'> ~/monprojet
```

## Différents types d'import

```
#Import relatif implicite  
from outils import Util  
#Import relatif explicite  
from .outils import Util  
# 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 Util
    outil = Util()
    print("outil",outil.moi,outil.path)
```

```
-----
fichier __init__.py:
from outils import Util
## import relatif implicite
```

```
-----
fichier script.py :
from module import Util
## import absolu
outil = Util()
print("outil",outil.moi,outil.path)
```

## Exemples

```
~/monprojet/module$ python
scriptoutils.py
    => OK (import relatif implicite)
    outil <class 'outils.Util'>
~/monprojet
```

```
~$ python ~/monprojet/script.py
    => OK
    outil <class 'outils.Util'>
~/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 XxxUtil(object):
    ...
fichiers script.py :
from module import Util,
    SousUtil, AutreSousUtil
```

## Fichiers \_\_init\_\_.py

```
.../module/autresousmodule/__init__.py:
from .autresousoutils import AutreSousUtil
.../module/sousmodule/__init__.py:
from .sousoutils import SousUtil
from ..autresousmodule import AutreSousUtil
.../module/__init__.py:
from .outils import Util
from .autre_outils import AutreUtil
from .sousmodule import SousUtil
from ..autresousmodule import AutreSousUtil
```

## Autre solution (absolu)

```
.../module/autresousmodule/__init__.py:
from module.autresousmodule.autresousoutils
    import AutreSousUtil
.../module/sousmodule/__init__.py:
from module.sousmodule.sousoutils
    import SousUtil
from module.autresousmodule import AutreSousUtil
```

# 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

## Dans script.py :

```
~/monprojet/module/  
  __init__.py  
  script.py  
  outils.py  
  autre_outils.py  
  sousmodule/  
    __init__.py  
    sousoutils.py  
  autresousmodule/  
    __init__.py  
    autresousoutils.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`  
 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
print(os.path.dirname(
    os.path.dirname(os.path.abspath(__file__))))
# -> ~/monprojet/
## Ajout a Python Path du repertoire contenant script.py
sys.path.append(os.path.dirname(
    os.path.dirname(os.path.abspath(__file__))))

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

A utiliser avec parcimonie !!

# Structure répertoire recommandé

```
projet/  
  module/__init__.py      : get_team, get_challenge, ...  
  module/team.py  
  module/sousmodules/ ...  
  script_essai.py        # from module import get_team  
  script2_essai.py       # from module import get_team ...
```

## Compatibilité Python 2

Rajouter : `from __future__ import absolute_import` au début de tous les fichiers.

## Pour importer d'autres joueurs :

- copier le répertoire du module dans votre répertoire
- `import binome; binome.get_team(1)`

# Plan

Organisation de vos fichiers

**Design Patterns**

# 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

# Une liste non exhaustive

## Creational Patterns

Abstract Factory

Builder

Factory Method

Prototype

Singleton

## Structural Patterns

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

## Behavioural Patterns

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

Visitor

# 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 ...
    ...
a = MyClass()
a.name # plutot que a.name()
```

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 = DuckAdapter(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.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 : tirer au but

```
class Decorator:
    def __init__(self, state):
        self.state = state
    def __getattr__(self, attr):
        return getattr(self.state, attr)
class Shoot(Decorator):
    def __init__(self, state):
        Decorator.__init__(self, state)
    def shoot(self, p):
        return SoccerAction(Vector2D(...))
class Passe(Decorator):
    def __init__(self, state):
        Decorator.__init__(self, state)
    def passe(self, p):
        return SoccerAction(Vector2D(...))
```

```
mystate = Shoot(Passe(state))
```

# Decorator : peut changer le comportement d'une fonction

## Exemple : modifier la passe

```
class MeilleurPasse(Decorator):  
    def petite_passe(self,p):  
        return SoccerAction(...)   
    def passe(self,p):  
        if (condition):  
            return self.petite_passe(p)  
        return self.state.passe(p)  
mystate = MeilleurPasse(Passe(state))
```

# Strategy

Le pattern Strategy définit une famille d'algorithmes, les encapsule et les rend interchangeables. Il 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.compute_strategy = func
    @property
    def name(self):
        if hasattr(self.func, "name"):
            return self.func.name
        return self.func.__name__
def passe(state, id_team, id_player):
    return fait_une_passe()
def cours(state, id_team, id_player):
    return cours_versr()
```

```
stratCours = StrategyExample(cours)
stratCours.name = "cours"
stratPasse = StrategyExample(passe)
stratPasse.name = "passe"
```

# Vos difficultés pour l'instant

- Décomposer et préciser vos stratégies
- Extraire de l'information des états
- Faire des stratégies génériques
- Réagir en fonction de situations

## Vos difficultés pour l'instant

- Décomposer et préciser vos stratégies
- Extraire de l'information des états
- Faire des stratégies génériques
- Réagir en fonction de situations

### Quelques conseils

- Ne mélangez pas les outils, les actions et les stratégies !
  - Une classe (ou plusieurs) pour enrichir vos objets
  - Des classes (ou fonctions) pour agir
  - Des classes (ou fonctions) pour chaque stratégie
- Tout doit être générique ! Si vous décidez à un moment de changer votre façon de courir, il ne faut rien toucher à la description ou aux stratégies !
- Que faire pour les symétries ?
- N'oubliez pas d'identifier de façon unique chaque petite stratégie (par son nom).
- Vous pouvez additionner deux `SoccerAction`

# Observer : un design pattern de plus

## SoccerEvents

- Une simulation possède une liste d'observateurs (`listeners`)
- A chaque évènement marquant, tous les observateurs sont avertis par l'appel d'une fonction
- il est possible d'ajouter à la volée ou de supprimer des observateurs de la simulation.

## Actions déclenchées lors d'une simulation

- `begin_match(team1, team2, state)` : au début de la simulation
- `end_match(team1, team2, state)` : à la fin
- `begin_round(team1, team2, state)` : au début de chaque engagement
- `end_round(team1, team2, state)` : à chaque but marqué
- `update_round(team1, team2, state)` : à chaque fin de tour



# Comment utiliser l'observeur ?

## Pour simuler !

```
class Observer(object):
    MAX_STEP=40
    def __init__(self,simu):
        self.simu = simu
        self.simu.listeners+=self #ajout de l'observer
    def begin_match(self,team1,team2,state):
        #initialisation des parametres ...
        self.last, self.cpt, self.cpt_tot = 0, 0, 0
    def begin_round(self,team1,team2,state):
        self.simu.state.states[(1,0)].position = ...
        #ou self.simu.set_state(state)
        self.strat.shoot = ...
        self.last = self.simu.step
    def update_round(self,team1,team2,state):
        if state.step>self.last+self.MAX_STEP: self.simu.end_round()
    def end_round(self,team1,team2,state):
        if state.goal>0: self.cpt+=1
        self.cpt_tot+=1
        self.res[...] = self.cpt*1./self.cpt_tot.
        if ... : #fin de la simu
            self.simu.end_match()
```

## Ou hériter de Simulation

```

class ChallengeFonceurButeur(Simulation):
    def __init__(self, team1, max_but=20, max_steps=5*settings.MAX_GAME_S
        super(ChallengeFonceurButeur, self).__init__(team1, None, max_ste
        ...
    def begin_round(self):
        self._last_step = self.state.step
        super(ChallengeFonceurButeur, self).begin_round()
    def update_round(self):
        super(ChallengeFonceurButeur, self).update_round()
    def end_round(self):
        if self.state.goal==1: self.resultats.append(self.state.step-s
        super(ChallengeFonceurButeur, self).end_round()
    def stop(self):
        return super(ChallengeFonceurButeur, self).stop() or self.state
    def end_match(self):
        if len(self.resultats)>0: self.stats_score = sum(self.resultat
        super(ChallengeFonceurButeur, self).end_match()
    def get_initial_state(self):
        state = SoccerState()
        state.ball = Ball(Vector2D(0,0))
        state.states[(1,0)] = PlayerState(position = ..., vitesse = ...
        state.states[(2,0)] = PlayerState(position = ..., vitesse = ..
        ...
        return state

```