2I013 Groupe 1

Projet Foot

Maxime Sangnier – Nicolas Baskiotis
maxime.sangnier@lip6.fr

2019

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

Outline

Séance précédente

Bilan du TME

Vous avez :

- enrichi le `state` avec un nouvel objet `SuperState` ;
- programmé un joueur attaquant et un joueur défenseur.

```
from superstate import SuperState

class SimpleStrategy(Strategy):
    def __init__(self, action, name):
        super().__init__(name)
        self.action = action

    def compute_strategy(self, state, id_team, id_player):
        s = SuperState(state, id_team, id_player)
        return self.action(s)

def gogetter(state):
    if state.near_ball:
        return SoccerAction(shoot=state.goal - state.player)
    else:
        return SoccerAction(acceleration=state.ball - state.play

...

team2.add("Go", SimpleStrategy(gogetter, 'Go-getter'))
```

Les opposants

Comment trouver

- la liste des opposants ?
- l'opposant le plus proche ?

Les opposants

Comment trouver

- la liste des opposants ?
- l'opposant le plus proche ?

Liste des opposants

```
opponents = [self.state.player_state(id_team, id_player).position
             for (id_team, id_player) in self.state.players
             if id_team != self.id_team]
```

Les opposants

Comment trouver

- la liste des opposants ?
- l'opposant le plus proche ?

Liste des opposants

```
opponents = [self.state.player_state(id_team, id_player).position
             for (id_team, id_player) in self.state.players
             if id_team != self.id_team]
```

Opposant le plus proche

```
min([(self.player.distance(player), player) for player in opponents])
```

La position de la balle

Comment anticiper

- la position de la balle

La position de la balle

Comment anticiper

- la position de la balle

Approximation linéaire

```
self.state.ball.position + 5 * self.state.ball.vitesse
```

Un peu plus sur l'encapsulation d'objets

Encapsuler un objet

Il s'agit

- d'enrichir un objet de nouvelles fonctionnalités ;
- de cacher certaines fonctionnalités bas niveau.

Exemple

Fichier `tools.py`

```
class SuperState(object):
  def __init__(self, state, id_team, id_player):
    self.state = state
    self.id_team = id_team
    self.id_player = id_player

  @property
  def ball(self):
    return self.state.ball.position

  @property
  def player(self):
    return self.state.player_state(self.id_team, self.id_player).position

  @property
  def goal(self):
    ...
```

Encapsuler un objet

Il s'agit

- d'enrichir un objet de nouvelles fonctionnalités ;
- de cacher certaines fonctionnalités bas niveau.

Exemple

Fichier principal

```
class GoStrategy(Strategy):
def __init__(self):
  Strategy.__init__(self, "Go-getter")

def compute_strategy(self, state, id_team, id_player):
  s = SuperState(state, id_team, id_player)
  if s.player.distance(s.ball) < PLAYER_RADIUS+BALL_RADIUS:
   return SoccerAction(shoot=s.goal-s.player)
  else:
    return SoccerAction(acceleration=s.ball-s.player)
```

Simuler l'héritage

Il s'agit

- de conserver toutes les fonctionnalités de l'objet d'origine ;
- d'en redéfinir certaines ;
- d'enrichir cet objet de nouvelles fonctionnalités.

Exemple

Fichier `tools.py`

```
class SuperState(object):
  def __init__(self, state, id_team, id_player):
    self.state = state
    self.id_team = id_team
    self.id_player = id_player

  def __getattr__(self, attr):
    return getattr(self.state, attr)

  @property
  def ball(self):
    return self.state.ball.position

  ...
```

Simuler l'héritage

Il s'agit

- de conserver toutes les fonctionnalités de l'objet d'origine ;
- d'en redéfinir certaines ;
- d'enrichir cet objet de nouvelles fonctionnalités.

Exemple

Fichier principal

```
class GoStrategy(Strategy):
def __init__(self):
  Strategy.__init__(self, "Go-getter")

def compute_strategy(self, state, id_team, id_player):
  s = SuperState(state, id_team, id_player)
  speed = s.player_state(id_team, id_player).vitesse
  ...
```

Encore de l'organisation du code

Séparation des deux actions

Constat

- Les stratégies définissent deux actions : déplacement et tir.
- Est-il judicieux de mettre toutes les fonctionnalités nécessaires à ces deux actions dans `SuperState` ?
- Peut-on créer une couche d'abstraction supplémentaire ?
- On peut additionner des `SoccerAction` !

Séparation des deux actions

Exemple

Fichier `actions.py`

```
class Move(object):
    def __init__(self, superstate):
        self.superstate = superstate

    def move(self, acceleration=None):
        return SoccerAction(acceleration=acceleration)

    def to_ball(self):
        return self.move(self.superstate.ball_dir())

class Shoot(object):
    def __init__(self, superstate):
        self.superstate = superstate

    def shoot(self, direction=None):
        dist = self.superstate.player.distance(self.superstate.ball)
        if dist < PLAYER_RADIUS + BALL_RADIUS:
            return SoccerAction(shoot=direction)
        else:
            return SoccerAction()

    def to_goal(self, strength=None):
        return self.shoot(self.superstate.goal_dir)
```

Séparation des deux actions

Exemple

Fichier principal

```
class GoStrategy(Strategy):
  def __init__(self):
    Strategy.__init__(self, "Go-getter")

  def compute_strategy(self, state, id_team, id_player):
        s = SuperState(state, id_team, id_player)
        move = Move(s)
        shoot = Shoot(s)
        return move.to_ball() + shoot.to_goal()
```

Création d'un module

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()` `module.myvar`	`myf()` `myvar`	`f()` `v`	`m.myf()` `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 réalité, 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 ...
  dans un fichier script : `import module`

$\Rightarrow$ 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.
- Mais doivent pouvoir être trouvés par Python $\Rightarrow$ se trouver dans le chemin défini par `PYTHONPATH`

La variable PYTHONPATH

Par défaut, elle contient les répertoires :

- système : `/usr/lib/python3.5`
- des paquets installés au niveau du système : `/usr/lib/python3/dist-packages`
- des paquets installés localement à l'utilisateur : `/usr/local/lib/python3.5/dist-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

```
~/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

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 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 ..autresousmoudule import AutreSousOuti
```

Autre solution (absolu)

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

Remarques

Et avec Python 2 ?

- 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
$ python module/script.py => ERROR
```

Solution :

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

Structure répertoire recommandé

Exemple de strcture

```
2I013/
  Soccer/
    .git/
    examples
    slides
    soccersimulator
    tutoriels

  monprojet
    .git/
    tournament.py
    simulation.py
    module/
      __init__.py
      tools.py
      strategies.py
```

Tournois par équipe

Définir ses équipes

- Créer le fichier `tournament.py`.
- Il doit contenir une fonction `get_team` qui définit les équipes.
- Vous devez vérifier que les imports fonctionnent correctement.

Fichier `tournament.py`

```
from socceria import StrikeStrategy, RandomStrategy
from soccersimulator import SoccerTeam


def get_team(nb_players):
    team = SoccerTeam(name="Maxime's Team")
    if nb_players == 1:
        team.add("Striker", StrikeStrategy())
    if nb_players == 2:
        team.add("Striker", StrikeStrategy())
        team.add("Random", RandomStrategy())
    return team
```

Les choses essentielles pour les tournois

Définir ses équipes

- Créer le fichier `tournament.py`.
- Il doit contenir une fonction `get_team` qui définit les équipes.
- Vous devez vérifier que les imports fonctionnent correctement.

Fichier `tournament.py`

```
if __name__ == '__main__':
    from soccersimulator import Simulation, show_simu

    # Check teams with 1 player and 2 players
    team1 = get_team(1)
    team2 = get_team(2)

    # Create a match
    simu = Simulation(team1, team2)

    # Simulate and display the match
    show_simu(simu)
```

Création des équipes

Création des équipes

Comptes github connus

NasserChaker
noeger
lucasbourlette
Téo Bernier
manelfil
mehdibds

Sur github

- choisissez un compte **github** ;
- partagez votre dépôt avec l'autre ;
- renseignez le formulaire : `https://goo.gl/forms/kLOdERD8LOpEpGwu1`

Objectifs du TME

Objectifs du TME

- organiser son code à l'aide d'un **module** et de scripts ;
- décomposer ses stratégies de manière à les rendre génériques (si vous décidez à un moment de changer votre façon de courir, il ne faut rien toucher à la description ou aux stratégies) ;
- gérer les symétries ;
- créer le fichier `tournament.py`
- affiner les joueurs d'attaque et de défense (positionnement, dribble. . .) de sorte à faire face à :
  - un fonceur ;
  - un défenseur naïf.

A chaque TME

Mettre à jour le dépôt contenant le simulateur :

```
cd [DOSSIER DU SIMULATEUR]
git pull
```