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

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

$\Rightarrow$ 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

$\Rightarrow$ 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]}]')
[u'foo', {u'bar': [u'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

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

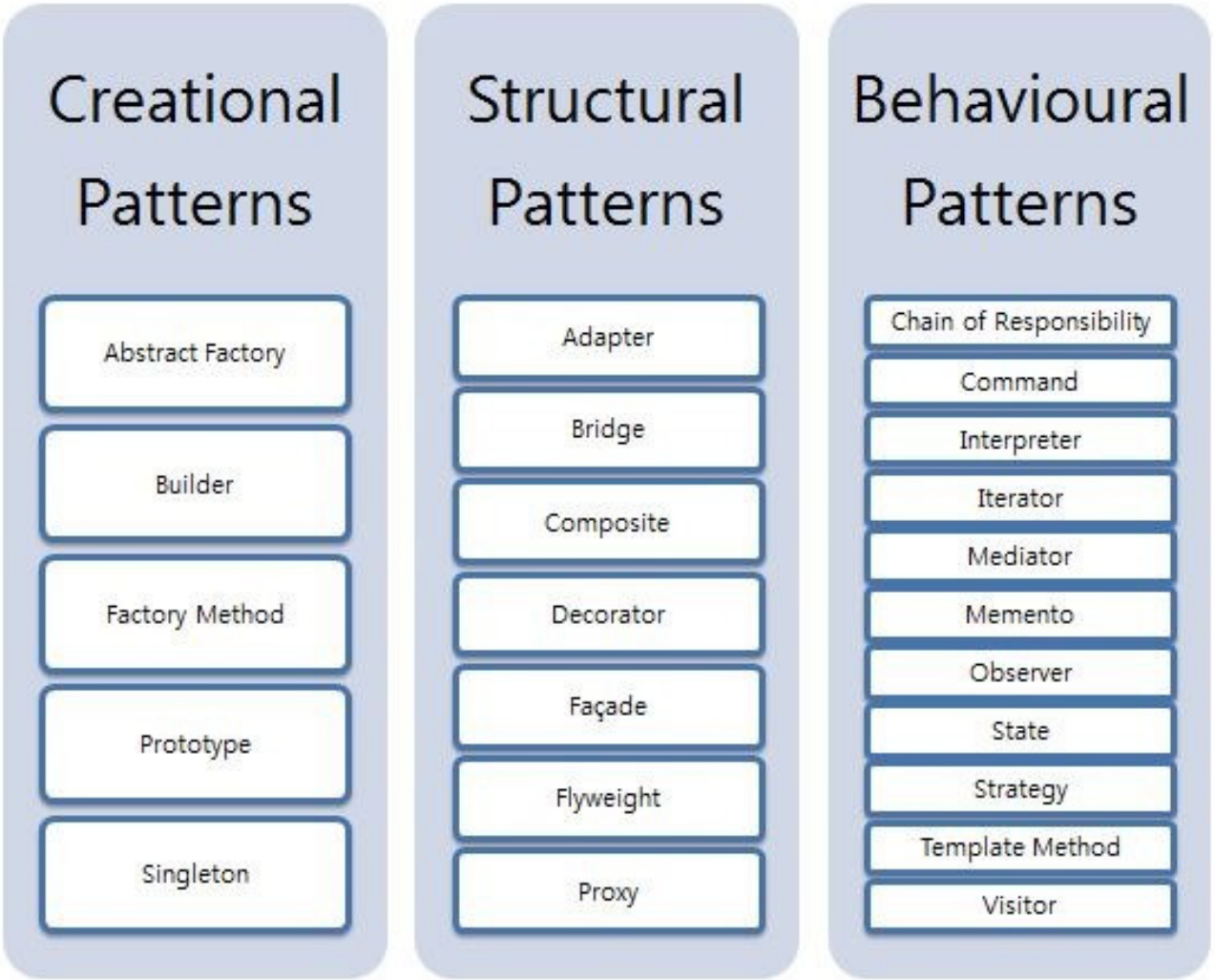
- Surtout pour les langages fortement typés, structurés (**Java** par exemple)
- Identifier les aspects de votre programme qui peuvent varier/évoluer et les séparer de ce qui reste identique
- Penser de manière générique et non pas en termes d'implémentations
- Composer plutôt qu'hériter (plus flexible)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

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

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 rercherche 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.

Concrétement

```
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 responsability

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

Quand beaucoup d'actions complexes doivent être appliquer, 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 définie 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

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 Controler:
    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)
```

Controleur

Controleur asynchrone

- A chaque commande, on envoie une intention
- On ne sait pas ce qui a été exécuté !!
- Il faut controler 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 Controler:
    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`

Controleur

Controleur asynchrone

- A chaque commande, on envoie une intention
- On ne sait pas ce qui a été exécuté !!
- Il faut controler 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 Controler:
    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)
```