td1a_cenonce_session6

October 21, 2020

1 1A.2 - Classes, héritage

L'héritage permet de réécrire certaines parties du code sans pour autant enlever les anciennes versions toujours utilisées.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.0.1 Qu'est-ce que l'héritage ?

La séance précédente a montré comment fonctionnait une classe, comment elle s'écrivait. Cette séance est à propos de l'*héritage* qui est une propriété des langages objets. Elle est utile par exemple lorsqu'on doit écrire plusieurs versions d'un même algorithme et qu'une petite partie seulement change d'une version à l'autre. Supposons que vous ayez un algorithme constitué de trois fonctions plus une dernière qui appelle les trois autres dans le bon ordre. On désire créer une version pour laquelle une des trois fonctions seulement est modifiée.

[2]:
```
class Version1:
    def __init__(self, p):
        self.p = p
    def fonction1(self):
        print("Version1.fonction1", self.p)
    def fonction2(self):
        print("Version1.fonction2", self.p)
    def fonction3(self):
        print("Version1.fonction3", self.p)
    def fonction_finale(self):
        self.fonction1()
        self.fonction2()
        self.fonction3()

v = Version1(0)
v.fonction_finale()
```

```
Version1.fonction1 0
Version1.fonction2 0
Version1.fonction3 0
```

On souhaite changer la fonction `fonction2` sans modifier la classe `Version1` et en écrivant le moins possible de code.

[3]:
```
class Version2(Version1):
    def fonction2(self):
        print("Version2.fonction2", self.p+1)

v = Version2(0)
v.fonction_finale()
```

```
Version1.fonction1 0
Version2.fonction2 1
Version1.fonction3 0
```

Le langage a compris qu'on avait changé une fonction et il s'en sert dans la seconde classe. Pour que cela fonctionne, il faut néanmoins respecter une contrainte essentielle : la fonction remplacée (ou surchargée) doit accepter les mêmes paramètres et retourner le même type de résultat. Cette contrainte n'est pas obligatoire en Python mais elle l'est dans la plupart des langages. Il est conseillé de la respecter.

1.0.2 Exercice 1 : pièce normale

On créer une classe `Piece` qui contient deux méthodes : une méthode `tirage_aleatoire` et une méthode qui appelle la précédente pour faire une moyenne sur n tirages.

[4]:
```
import random

class Piece:
    def tirage_aleatoire(self, precedent):
        return random.randint(0,1)
    def moyenne_tirage(self, n):
        # ....
        return 0  # à remplacer

p = Piece()
print(p.moyenne_tirage(100))
```

```
0
```

1.0.3 Exercice 2 : pièce truquée

Le paramètre `precedent` est inutile dans cette première version mais on suppose maintenant que le joueur qui joue est un tricheur. Lorsqu'il perd, il joue une pièce truquée le coup d'après pour laquelle la probabilité d'avoir 1 est de 0,7. On veut implémenter cela avec une classe `PieceTruquee`.

[5]:
```
import random

class PieceTruquee(Piece):
    # ......
    pass
```

Pour choisir de faire telle ou telle avec une probabilité de 0,7, on peut écrire :

[6]:
```
if random.random() <= 0.7 :
    # ... faire une chose avec la probabilité 0.7
    pass
else :
    # ... faire une autre chose avec la probabilité 0.3
    pass
```

1.0.4 Utiliser des méthodes de la classe mère

Lorsqu'on change une fonction, on a parfois juste d'un petit changement par rapport à la méthode précédente qu'il faut pouvoir appeler. Si on reprend l'exemple précédent, on modifie la méthode `tirage_aleatoire` pour retourner l'autre valeur :

[7]:
```
class PieceTruquee(Piece):
    def tirage_aleatoire(self, precedent):
        return 1 - Piece.tirage_aleatoire(self, precedent)
p = PieceTruquee()
p.tirage_aleatoire(0)
```

[7]: 1

Une autre écriture possible est la suivante avec le mot-clé super :

[8]:
```
class PieceTruquee(Piece):
    def tirage_aleatoire(self, precedent):
        return 1 - super().tirage_aleatoire(precedent)
p = PieceTruquee()
p.tirage_aleatoire(0)
```

[8]: 0

1.0.5 Exercice 3 : Pièce mixte

Ecrire une classe `PieceTruqueeMix` qui appelle aléatoiremnt soit `Piece.tirage_aleatoire` soit `PieceTruquee.tirage_aleatoire`.

[9]:
```
```

1.0.6 Autre construction avec des fonctions

La création de classe peut sembler fastidieuse. Une autre solution est l'utilisation de fonction comme paramètre d'une autre fonction :

[10]:
```
def moyenne_tirage(n, fonction):
    """
    cette fonction fait la moyenne des résultats produits par la fonction passée en␣
↪argument
    ce texte apparaît dès qu'on écrit help(moyenne_tirage) (ou moyenne_tirage? dans un␣
↪notebook)
    """
    tirage = [ ]
    for i in range (n) :
        precedent = tirage[-1] if i > 0 else None
        tirage.append( fonction (precedent) )
    s = sum(tirage)
    return s * 1.0 / len(tirage)

print (moyenne_tirage(100, lambda v : random.randint(0,1) ))

def truquee (precedent) :
    if precedent == None or precedent == 1 :
        return random.randint(0,1)
    else :
```

```
        return 1 if random.randint(0,9) >= 3 else 0

print(moyenne_tirage(100, truquee ))
```

```
0.49
0.59
```

1.0.7 Exercice 4 : pièce mixte avec des fonctions

Comment utiliser les fonctions dans le cas de la pièce `PieceTruqueeMix` ?

[11]: