COURS DE NSI (Numérique et Sciences Informatiques)

Classe : NSIT^le

séquence 4

Mr BANANKO K.

LYCÉE INTERNATIONAL COURS LUMIÈRE

LES DIFFERENTS PARADIGMES DE PROGRAMMATION

La programmation orientée objet

Objectifs :

Contenus	*Capacités attendues*	*Commentaires*
Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.	Écrire la définition d'une classe. Accéder aux attributs et méthodes d'une classe.	On n'aborde pas ici tous les aspects de la programmation objet comme le polymorphisme et l'héritage.

Référence Manuels hachette Education :

I- Introduction : les différents paradigmes de programmation

Jusqu'à présent nous avons vu un seul paradigme de programmation (*Il consiste à créer des procédures (ou routines) qu'il est possible d'appeler pour obtenir l'exécution de courtes séquences d'instructions)* : la programmation impérative. La programmation impérative repose sur des notions qui vous sont familières :

- La séquence d'instructions (les instructions d'un programme s'exécutent l'une après l'autre)
- L'affectation (on attribue une valeur à une variable, par exemple : a = 5)
- L'instruction conditionnelle (if / else)
- La boucle (while et for)

La programmation impérative est loin d'être le seul paradigme de programmation (même si c'est sans doute le plus courant). Cette année nous allons étudier deux autres paradigmes : **le paradigmes objet et le paradigme fonctionnel**.

II- Programmation fonctionnelle

Comme dit l'année dernière dans la partie du cours consacrée aux effets de bord, le **paradigme fonctionnel** cherche à éviter au maximum les effets de bord, dit autrement, en programmation fonctionnelle on va éviter de modifier les valeurs associées à des variables.

Pour ce faire, on va chercher au maximum à utiliser les fonctions (*d'où le nom de programmation fonctionnelle*), mais ces fonctions ne devront pas modifier les variables : en **programmation**

fonctionnelle, on s'efforce de coder des fonctions qui ne modifient pas l'état courant des variables. Les fonctions utilisées en programmation fonctionnelle sont parfois appelées "**fonction pure**" : le résultat renvoyé par une fonction pure doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction (*elle ne doit pas non plus engendrer d'effet de bord*):

Activité 1

Analysez le programme suivant :

```
i = 5
def fct():
  if i > 5:
    return True
  else :
    return False
fct()
```

La fonction ci-dessus n'est pas une fonction pure, car la valeur renvoyée par la fonction fct (***True ou False***) dépend d'une valeur extérieure à la fonction.

Activité 2

Analysez le programme suivant :

```
def fct(i):
  if i > 5:
    return True
  else :
    return False
fct(5)
```

La fonction ci-dessus est une fonction pure, car la valeur renvoyée par la fonction fct (***True ou False***) dépend uniquement du paramètre passé à la fonction.

Même si certains langages de programmation ont été conçus pour "**imposer**" au programmeur le paradigme fonctionnel (*Lisp, Scheme, Haskell*...), il est tout à fait possible d'utiliser le **paradigme fonctionnel** avec des langages de programmation plus "**généralistes**" (Python par exemple).

Nous allons maintenant travailler sur un exemple de programme Python utilisant le paradigme fonctionnel :

Activité 3

Analysez le programme suivant :

```
l = [4,7,3]
def ajout(i):
  l.append(i)
```

Selon vous, le programme ci-dessus respecte-t-il le paradigme fonctionnel ?

La réponse à la question posée ci-dessus est non, car nous avons un effet de bord (*la fonction ajout modifie le tableau l*)

À faire vous-même 4

Analysez le programme suivant :

```
def ajout(i,l):
  tab = l + [i]
  return tab
```

Selon vous, le programme ci-dessus respecte-t-il le paradigme fonctionnel ?

La fonction ajout ne modifie aucune variable, elle crée un nouveau tableau (*tab*) à partir du tableau l et du paramètre i (*le signe + permet de créer un nouveau tableau, ce nouveau tableau*

est constitué des éléments contenus dans le tableau l auxquels on ajoute la valeur i), la fonction renvoie le tableau ainsi créé.

D'une façon plus générale, la méthode **append** de Python ne respecte pas le paradigme fonctionnel puisque **append** modifie une donnée existante. Le paradigme fonctionnel va amener le programmeur non pas à modifier une valeur existante, mais plutôt à créer une nouvelle grandeur à partir de la grandeur existante : ***une grandeur existante n'est jamais modifiée, donc aucun risque d'effet de bord.***

Utiliser le bon paradigme de programmation

Il est important de bien comprendre qu'un programmeur doit maitriser plusieurs paradigmes de programmation (***impératif, objet ou encore fonctionnelle).*** En effet, il sera plus facile d'utiliser le paradigme objet dans certains cas alors que dans d'autres situations, l'utilisation du paradigme fonctionnelle sera préférable. Être capable de Choisir le "**bon**" paradigme en fonction des situations fait partie du bagage de tout bon programmeur.

III- LE PARADIGME OBJET

La programmation orientée objet repose, comme son nom l'indique, sur le concept d'objet.

Un objet dans la vie de tous les jours, vous connaissez, mais en informatique, *qu'est ce que c'est ? Une variable ? Une fonction ? Ni l'un ni l'autre,* c'est un nouveau concept.

Imaginez un objet (*de la vie de tous les jours*) très complexe (par exemple un moteur de voiture) : il est évident qu'en regardant cet objet, on est frappé par sa complexité (*pour un non spécialiste*). Imaginez que l'on enferme cet objet dans une caisse et que l'utilisateur de l'objet n'ait pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser. L'utilisateur a, à sa disposition, des boutons, des manettes et des écrans de contrôle pour faire fonctionner l'objet, ce qui rend son utilisation relativement simple. La mise au point de l'objet (*par des ingénieurs*) a été très complexe, en revanche son utilisation est relativement simple. Programmer de manière orientée objet, c'est un peu reprendre cette idée : utiliser des objets sans se soucier de leur complexité interne. Pour utiliser ces objets, nous n'avons pas à notre disposition des boutons, des manettes ou encore des écrans de contrôle, mais des attributs et des méthodes (*nous aurons l'occasion de revenir longuement sur ces 2 concepts*). Un des nombreux avantages de la programmation orientée objet (***POO***), est qu'il existe des milliers d'objets (*on parle plutôt de classes, mais là aussi nous reviendrons sur ce terme de classe est peu plus loin*) prêts à être utilisés (*vous en avez déjà utilisé de nombreuses fois sans le savoir*). On peut réaliser des programmes extrêmement complexes uniquement en utilisant **des classes préexistantes**.

Les idées sous-tendant le paradigme objet datent des années 60. Mais il faudra attendre le début des années 70 et la mise au point du **langage Smalltalk**[1] pour que le paradigme objet gagne en popularité chez les informaticiens. Aujourd'hui de nombreux langages permettent d'utiliser le paradigme objet : ***C++, Java,...***

Pour nous initier à la programmation orientée objet nous allons utiliser un langage que vous connaissez bien : Python. Python permet d'utiliser le paradigme impératif (comme nous l'avons fait jusqu'à présent), mais il permet aussi d'utiliser le paradigme objet. Il est même possible, comme nous le verrons plus loin, d'utiliser les 2 paradigmes dans un même programme.

La création d'une classe en python commence toujours par le mot class. Ensuite toutes les instructions de la classe seront indentées :

```
class LeNomDeMaClasse:
  #instructions de la classe
#La définition de la classe est terminée.
```

La classe est une espèce de moule (*nous reviendrons plus tard sur cette analogie qui a ses limites*), à partir de ce moule nous allons créer des objets (*plus exactement nous parlerons d'instances*). Par exemple, nous pouvons créer une classe voiture, puis créer différentes instances de cette classe (Peugeot407, Renault Espace,...). Pour créer une de ces instances, la procédure est relativement simple :

```
peugeot407 = Voiture()
```

Cette ligne veut tout simplement dire : "***crée un objet (une instance) de la classe Voiture que l'on nommera peugeot407***."

Ensuite, rien ne nous empêche de créer une deuxième instance de la classe Voiture :

```
renaultEspace = Voiture()
```

[1] *Smalltalk est un langage de programmation orienté objet, réflexif et dynamiquement typé. Il fut l'un des premiers langages de programmation à disposer d'un environnement de développement intégré complètement graphique.*

Nous rencontrons ici la limite de notre analogie avec le moule. En effet 2 objets fabriqués avec le même moule seront (*définitivement*) identiques, alors qu'ici nos 2 instances pourront évoluer différemment.

Pour développer toutes ces notions (*et d'autres*), nous allons écrire un premier programme :

Nous allons commencer par écrire une classe Personnage (*qui sera dans un premier temps une coquille vide*) et, à partir de cette classe créer 2 instances : **bilbo et gollum.**

Activité 1

Saisissez, analysez et testez ce code

```
class Personnage:
  pass
gollum = Personnage()
bilbo = Personnage()
```

Pour l'instant, notre classe ne sert à rien et nos instances d'objet ne peuvent rien faire. Comme il n'est pas possible de créer une classe totalement vide, nous avons utilisé l'instruction ***pass*** qui ne fait rien. Ensuite nous avons créé 2 instances de la classe Personnage : ***gollum et bilbo***.

Comme expliqué précédemment, une instance de classe possède des attributs et des méthodes.

Commençons par les attributs :

Un attribut possède une valeur (*un peu comme une variable*). Nous allons associer un attribut vie à notre classe Personnage (*chaque instance aura un attribut vie, quand la valeur de vie deviendra nulle, le personnage sera mort !)*

Ces attributs s'utilisent comme des variables, l'attribut vie pour **bilbo** sera noté

```
bilbo.vie
```

de la même façon l'attribut vie de l'instance **gollum** sera noté

```

```

```
gollum.vie
```

Activité 2

Saisissez, analysez et testez ce code

```
class Personnage:
  pass
gollum=Personnage()
gollum.vie=20
bilbo=Personnage()
bilbo.vie=20
```

Comme pour une variable il est possible d'utiliser la console Python pour afficher la valeur référencée par un attribut. Il suffit de taper dans la console **gollum.vie ou bilbo.vie** (*sans bien sûr avoir oublié d'exécuter le programme au préalable*.)

Cette façon de faire n'est pas très "**propre**" et n'est pas une bonne pratique

En effet, nous ne respectons pas un principe de base de la POO : l'encapsulation

Il ne faut pas oublier que notre classe doit être "**enfermée dans une caisse**" pour que l'utilisateur puisse l'utiliser facilement sans se préoccuper de ce qui se passe à l'intérieur, or, ici, ce n'est pas vraiment le cas.

En effet, les attributs (**gollum.vie et bilbo.vie**), font partie de la classe et devraient donc être enfermés dans la "caisse" !

Pour résoudre ce problème, nous allons définir les attributs, dans la classe, à l'aide d'une méthode (*une méthode est une fonction définie dans une classe*) d'initialisation des attributs.

Cette méthode est définie dans le code source par la ligne :

```
def __init__ (self)
```

La méthode **__init__** est automatiquement exécutée au moment de la création d'une instance. Le mot self est obligatoirement le premier argument d'une méthode (***nous reviendrons ci-dessous sur ce mot self)***

Nous retrouvons ce mot self lors de la définition des attributs. La définition des attributs sera de la forme :

```
self.vie=20
```

Le mot self représente l'instance. Quand vous définissez une instance de classe (***bilbo ou gollum***) le nom de votre instance va remplacer le mot self.

Dans le code source, nous allons avoir :

```
class Personnage:
  def __init__ (self):
    self.vie=20
```

Ensuite lors de la création de l'instance **gollum**, python va automatiquement remplacer self par **gollum** et ainsi créer un attribut gollum.vie qui aura pour valeur de départ la valeur donnée à **self.vie** dans la méthode **__init__**

Il se passera exactement la même chose au moment de la création de l'instance **bilbo,** on aura automatiquement la création de l'attribut **bilbo.vie**.

Activité 3

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self):
    self.vie=20
gollum=Personnage()
bilbo=Personnage()
```

Utilisez la console Python comme dans le "***Activité 2***"

Le résultat du "***Activité 3***" est identique au résultat de l'exemple du "***Activité 2***". Mais cette fois nous n'avons pas défini l'attribut **gollum.vie=20 et bilbo.vie=20** en dehors de la classe, nous avons utilisé une méthode **__init__.**

Imaginons que nos 2 personnages n'aient pas au départ les mêmes points de vie ! Pour l'instant, impossible d'introduire cette contrainte (**self.vie=20**)

Une méthode, comme une fonction, peut prendre des paramètres.

Le passage de paramètres se fait au moment de la création de l'instance :

Activité 4

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self,nbreDeVie):
    self.vie=nbreDeVie
gollum=Personnage(20)
bilbo=Personnage(15)
```

Utilisez la console Python pour vérifier que **gollum.vie** est égal à 20 et **bilbo.vie** est égal à 15

Au moment de la création de l'instance **gollum**, on passe comme argument le nombre de vies (***gollum=Personnage (20)***). Ce nombre de vies est attribué au premier argument de la méthode **__init__** , la variable **nbreDeVie** *(nbreDeVie n'est pas tout à fait le premier argument de la méthode **__init__** puisque devant il y a self, mais bon, self étant obligatoire, nous pouvons dire que nbreDeVie est le premier argument non obligatoire*).

N.B. Je parle bien de variable pour **nbreDeVie** (car ce n'est pas un attribut de la classe personnage puisqu'elle ne commence pas par self).

Nous pouvons passer plusieurs arguments à la méthode **__init__** (*comme pour n'importe quelle fonction*).

Nous allons créer 2 nouvelles méthodes :

- Une méthode qui enlèvera un point de vie au personnage blessé
- Une méthode qui renverra le nombre de vies restantes

Activité 5

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    self.vie=self.vie-1
gollum = Personnage(20)
bilbo = Personnage(15)
```

Pour tester ce programme, dans la console, tapez successivement les instructions suivantes :

- **gollum.donneEtat()**
- **bilbo.donneEtat()**
- **gollum.perdVie()**

- **gollum.donneEtat()**
- **bilbo.perdVie()**
- **bilbo.donneEtat()**

Vous avez sans doute remarqué que lors de "**l'utilisation**" des instances **biblo et gollum**, nous avons uniquement utilisé des méthodes et nous n'avons plus directement utilisé des attributs (**plus de "gollum.vie**"). Il est important de savoir qu'en dehors de la classe l'utilisation des attributs est une mauvaise pratique en programmation orientée objet : les attributs doivent rester "à l'intérieur" de la classe, l'utilisateur de la classe ne doit pas les utiliser directement. Il peut les manipuler, mais uniquement par l'intermédiaire d'une méthode (*la méthode **self.perdVie**() permet de manipuler l'attribut **self.vie***)

Pour l'instant nous avons utilisé les méthodes uniquement en tapant des instructions dans la console, il est évidemment possible d'utiliser ces méthodes directement dans votre programme :

Activité 6

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    self.vie=self.vie-1


bilbo = Personnage(15)
bilbo.perdVie()
point=bilbo.donneEtat()
```

Évaluez la variable point à l'aide de la console.

Activité 7

Nos personnages peuvent boire une potion qui leur ajoute un point de vie. Modifiez le programme du "**Activité 5"** en ajoutant une méthode **boirePotion**. Testez ensuite cette modification à l'aide de la console.

Selon le type d'attaque subit, le personnage peut perdre plus ou moins de points de vie. Pour tenir compte de cet élément, il est possible d'ajouter un paramètre à la méthode **perdVie** :

Activité 8

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self,nbPoint):
    self.vie=self.vie-nbPoint
bilbo = Personnage(15)
bilbo.perdVie(2)
point=bilbo.donneEtat()
```

Évaluez la variable point à l'aide de la console.

Il est possible d'ajouter une part d'aléatoire dans la méthode **perdVie** :

Activité 9

Saisissez, analysez et testez ce code

```
import random
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    if random.random()>0.5:
      nbPoint = 1
    else :
      nbPoint = 2
    self.vie=self.vie-nbPoint
bilbo = Personnage(15)
bilbo.perdVie()
bilbo.perdVie()
bilbo.perdVie()
point=bilbo.donneEtat()
```

Évaluez la variable point à l'aide de la console.

N.B : **random.random()** renvoie une valeur aléatoire comprise entre 0 et 1

Expliquez le fonctionnement de la méthode **perdVie**

Comme vous l'avez remarqué, il est possible d'utiliser une instruction conditionnelle (**if / else**) dans une méthode. Il est donc possible d'utiliser dans le même programme le paradigme objet et le paradigme impératif.

Nous allons maintenant organiser un combat virtuel entre nos 2 personnages :

Activité 10

Saisissez, analysez ce code

```
import random
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    if random.random()>0.5:
      nbPoint = 1
    else :
      nbPoint = 2
    self.vie=self.vie-nbPoint
def game():
  bilbo = Personnage(20)
  gollum = Personnage(20)
  while bilbo.donneEtat()>0 and gollum.donneEtat()>0 :
    bilbo.perdVie()
    gollum.perdVie()
  if bilbo.donneEtat()<=0 and gollum.donneEtat()>0:
    msg = f"Gollum est vainqueur, il lui reste encore {gollum.donneEtat()} points alors que Bilbo est mo
rt"
  elif gollum.donneEtat()<=0 and bilbo.donneEtat()>0:
    msg = f"Bilbo est vainqueur, il lui reste encore {bilbo.donneEtat()} points alors que Gollum est mor
t"
  else :
    msg = "Les deux combattants sont morts en même temps"
  return msg
```

Pour tester le programme, exécutez la fonction **game** dans une console. Vérifiez que l'on peut obtenir des résultats différents en exécutant plusieurs fois la fonction **game**.

Nous avons encore ici la démonstration qu'il est possible d'utiliser le paradigme objet et le paradigme impératif dans un même programme.

Activité 11

Améliorez le programme développé au "***Activité 10***" en modifiant des méthodes ou en implémentant vos propres méthodes.

Exercice du BAC

- Sujet 3 2021 Exercice 1
- Sujet 4 2021 Exercice 2
- Sujet 5 2021 Exercice 3
- Sujet 6 2021 Exercice 1
- Sujet 10 2021 Exercice 4
- Sujet 2 2022 Exercice 5
- Sujet 3 2022 Exercice 1
- Sujet 5 2022 Exercice 1
- Sujet 6 2022 Exercice 2
- Sujet 7 2022 Exercice 4
- Sujet 8 2022 Exercice 4
- Sujet 10 2022 Exercice 2
- Sujet 12 2022 Exercice 3
- Sujet 14 2022 Exercice 2

Ce qu'il faut savoir
paradigme fonctionnel

- la programmation fonctionnelle est un paradigme de programmation comme la programmation impérative ou la programmation orientée objet.
- le paradigme fonctionnel repose sur l'utilisation de fonction

- le paradigme fonctionnel cherche à éviter au maximum les effets de bord => en programmation fonctionnelle, on s'efforce de coder des fonctions qui ne modifient pas l'état courant des variables globales.
- Les fonctions utilisées en programmation fonctionnelle sont parfois appelées "fonction pure" : le résultat renvoyé par une fonction pure doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction

paradigme objet

La programmation orientée objet (poo) est un paradigme de programmation qui repose sur la notion de classe, la notion d'attribut et la notion de méthode (la poo repose aussi sur les notions d'héritage et de polymorphisme, mais ces 2 notions ne sont pas au programme de NSI). En poo on travaille sur des objets (des instances plus exactement), chaque instance est créée à partir d'un "moule" : la classe. Les attributs représentent l'état d'un objet alors que les méthodes représentent le comportement d'un objet.

Ce qu'il faut savoir faire

paradigme fonctionnel

être capable d'écrire un programme simple en Python qui respecte le paradigme fonctionnel

paradigme objet

- vous devez être capable d'analyser et comprendre un programme Python simple qui utilise le paradigme objet
- vous devez être capable d'écrire un programme Python simple qui utilise le paradigme objet