Programmation avec python

(2e partie : la POO)

Version 0.2 du 26/02/12

David Roche (lycée G Fichet Bonneville (Haute Savoie))

Notes de l'auteur :

- Ce document constitue la 2e partie du « cours » programmation avec python. Bien que hors programme, ce document pourra éventuellement interesser les élèves de la spécialité de terminale S ISN, notamment, lors de la réalisation des projets.

- A plusieurs reprises, j'ai copié-collé quelques lignes de l'excellent livre de G. Swinnen «Apprendre à programmer avec Python ». Cet ouvrage, publié sous licence GNU, est disponible gratuitement sur le net au format pdf ou en version papier pour une somme modique. Merci à G Swinnen pour son travail.
- Je me suis aussi inspiré du tutoriel sur le C++ de M@teo21 du « site du zéro ». Merci à M@teo21 pour son site (www.siteduzero.com) et pour tous les excellents tutoriaux qui s'y trouvent.
- Ce texte est en cours d'élaboration, merci à tous ceux qui l'ont fait évoluer et qui le feront évoluer dans le bon sens (merci à Nicolas Rougier pour ses remarques et ses conseils)

La programmation orientée objet (POO)

La programmation orientée objet est une méthode moderne de programmation extrêmement puissante.
Dans la vie de tous les jours, nous sommes entourés d'objet (par objet j'entends toutes choses inanimées, mais aussi les êtres vivants !). L'idée de la programmation orientée objet, c'est de manipuler des éléments que l'on appelle des "objets" dans son code source.
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 nouvel élément en programmation.
Pour être plus précis, un objet c'est un mélange de plusieurs variables et fonctions !
Imaginez un objet (de la vie de tous les jours) très complexe (par exemple un moteur de voiture pour un profane) :
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'a pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser. L'utilisateur a, à sa disposition, 2 boutons et une manette 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 créer du code source (qui peut-être complexe), mais que l'on masque en le plaçant à l'intérieur d'un cube (un objet) à travers lequel on ne voit rien. Pour le programmeur qui va utiliser une classe codée par un autre programmeur , travailler avec un objet est donc beaucoup plus simple qu'avant : il a juste à appuyer sur des boutons et n'a pas besoin d'être ingénieur pour s'en servir.

Un des nombreux avantages de la POO, c'est qu'au jour d'aujourd'hui, il existe des milliers d'objets (on les appelle des classes) prêts à être utilisés. On peut réaliser des programmes extrêmement complexes uniquement en utilisant des classes (objets) pré-existantes.
Bien évidemment, nous allons aussi créer nos propres classes en python.

La POO et python

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

```
class leNomDeMaClasse(object):
        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, RenaultEspace,.....). 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()
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 avoir « un destin » très différent.
Pour développer toutes ces notions (et d'autres), nous allons écrire un petit programme (sans aucune prétention) :
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.
ex1:

```
# Défini une classe personnage
class personnage(object):
        #Pour l'instant cette classe est une coquille vide, mais comme elle doit contenir une
        #instruction nous mettons l'instruction pass qui ne fait rien !
        pass
        # on crée une instance de la classe personnage nommée Gollum (on revient à la ligne car la
        #définition de la classe est terminée)
gollum=personnage()
# on crée une autre instance de la classe personnage nommée Bilbo
bilbo=personnage()
```

Comme expliqué plus haut une instance de classe possède des attributs (variables) et des méthodes (fonctions).Commençons par les attributs :
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.
ex2:

```
# Défini une classe personnage
class personnage(object):
        pass
# on crée une instance de la classe personnage nommée Gollum
gollum=personnage()
# On donne 20 points de vie à Gollum en créant un attribut (variable dans une classe) vie pour
#l'instance Gollum
gollum.vie=20
# on créé une autre instance de la classe personnage nommée Bilbo
bilbo=personnage()
# On donne 20 points de vie à Bilbo
bilbo.vie=20
#affichage du nombre de points de vie pour Bilbo
print ('Bilbo a',bilbo.vie,'points de vie')
#affichage du nombre de points de vie pour Gollum
print ('Gollum a',gollum.vie,'points de vie')
print ('Bilbo est blessé par Gollum')
#Bilbo est blessé, il perd un point de vie
bilbo.vie=bilbo.vie-1
#on affiche de nouveau les points de vie
print ('Bilbo a',bilbo.vie,'points de vie')
print ('Gollum a',gollum.vie,'points de vie')
```

A2

```
Bilbo a 20 points de vie
Gollum a 20 points de vie
Bilbo est blessé par Gollum
Bilbo a 19 points de vie
Gollum a 20 points de vie
```

Tout ceci est bien beau, mais pas très « propre », et cette façon de programmer ferait dresser les cheveux sur la tête de n'importe quel programmeur chevronné !
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, hors, 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 ! Que font-ils en dehors de la définition de la classe alors ?
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) » (le mot self est obligatoirement le premier argument quelque soit la méthode).
Nous retrouvons ce mot self lors de la définition des attributs (la définition des attributs sera de la forme « self.vie=20 »)
La présence de ce mot self est assez compliquée à expliquer. En gros, 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(object):
        def __init__ (self):
                self.vie=20
#fin de la classe
```

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__ (dans notre exemple 20)
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.

ex3

```
# Défini une classe personnage
class personnage(object):
        # on définit la méthode qui va initialiser les attributs
        def __init__(self):
                self.vie=20
# on crée une instance de la classe personnage nommée Gollum
gollum=personnage()
# on crée une autre instance de la classe personnage nommée Bilbo
bilbo=personnage()
#affichage du nombre de points de vie pour Bilbo
print ('Bilbo a',bilbo.vie,'points de vie')
#affichage du nombre de points de vie pour Gollum
print ('Gollum a',gollum.vie,'points de vie')
print ('Bilbo est blessé par Gollum')
#Bilbo est blessé, il perd un point de vie
bilbo.vie=bilbo.vie-1
#on affiche de nouveau les points de vie
print ('Bilbo a',bilbo.vie,'points de vie')
print ('Gollum a',gollum.vie,'points de vie')
```

A3

```
Bilbo a 20 points de vie
Gollum a 20 points de vie
Bilbo est blessé par Gollum
Bilbo a 19 points de vie
Gollum a 20 points de vie
```

Le résultat de l'exemple 3 est évidemment identique au résultat de l'exemple 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__.
La méthode __init__ est une fonction (plus exactement une méthode, mais nous avons vu que c'était la même chose), comme toutes fonctions on peut lui passer des paramètres sous forme de variable (comme déjà vu précédemment avec les fonctions).
Le passage de paramètres se fait au moment de la création de l'instance, encore un petit exemple pour éclaircir tout cela :
ex4

```
class personnage(object):
        #on définit la méthode qui va initialiser les attributs, elle possède maintenant un
#paramètre supplémentaire.
        def __init__(self,nbreDeVie):
                #l'attribut self.vie est initialisé avec la valeur de nbreDeVie
                self.vie=nbreDeVie
# on crée une instance de la classe personnage nommée Gollum et on initialise la variable
#nbreDeVie avec la valeur 20
gollum=personnage(20)
# on crée une autre instance de la classe personnage nommée Bilbo et on initialise la variable
#nbreDeVie avec la valeur 20
bilbo=personnage(20)
#le reste du code reste identique
print ('Bilbo a',bilbo.vie,'points de vie')
print ('Gollum a',gollum.vie,'points de vie')
print ('Bilbo est blessé par Gollum')
bilbo.vie=bilbo.vie-1
print ('Bilbo a',bilbo.vie,'points de vie')
print ('Gollum a',gollum.vie,'points de vie')
```

A4

```
Bilbo a 20 points de vie
Gollum a 20 points de vie
Bilbo est blessé par Gollum
Bilbo a 19 points de vie
Gollum a 20 points de vie
```

Quelques explications s'imposent :
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).
Ensuite, on attribue la valeur contenue dans la variable nbreDeVie à l'attribut self.vie (ligne self.vie=nbreDeVie)
NB : 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).
Bien évidemment, nous pouvons passer plusieurs arguments à la méthode __init__ (comme pour n'importe quelle fonction), nous verrons des exemples dans un instant.
Cela va déjà mieux, mais tout n'est pas encore parfait. En effet, nous voyons encore se « balader » des « gollum.vie » en dehors de la classe (notamment dans gollum.vie= gollum.vie-1 ou encore dans les lignes qui permettent d'afficher le nombre de vies restantes), ce qui pose encore un problème au niveau de l'encapsulation.

Pour résoudre ce problème, nous allons créer 2 nouvelles méthodes :
- Une méthode qui enlèvera un point de vie au personnage blessé
- Une méthode qui affichera le nombre de vies restantes

Voyons avec l'exemple 5 comment arriver à nos fins (attention, il va y avoir pas mal de nouveautés), nous allons en profiter pour faire passer plusieurs arguments à la méthode __init__

ex5

```
# Définit une classe
class personnage(object):
        #méthode __init__ avec 2 arguments (sans compter self !) le nombre de vies et le nom du
        #personnage
        def __init__(self, nbreDeVie, nomDuPerso):
                self.vie=nbreDeVie
                self.nom=nomDuPerso
        #voici la méthode qui affiche l'état du personnage (attention de bien mettre l'argument
        #self, il est obligatoire.
        def afficheEtat (self):
                print ('Il reste',self.vie,'points de vie à',self.nom)
        #voici la méthode qui fait perdre 1 point de vie au personnage qui a subit une attaque
        def perdVie (self):
                print (self.nom,'subit une attaque, il perd une vie')
                self.vie=self.vie-1
#la « fabrication » de la classe est terminée, la suite du programme est uniquement composée de
#création d'instances et d'appel de méthodes
#on crée l'instance gollum sans oublier de passer le nombre de points de vie et le nom du
#personnage
gollum=personnage(20,'Gollum')
#on appelle la méthode afficheEtat pour l'instance gollum
gollum.afficheEtat()
bilbo=personnage(20,'Bilbo')
#on appelle la méthode afficheEtat pour l'instance bilbo
bilbo.afficheEtat()
#on appelle la méthode perdVie pour l'instance bilbo
bilbo.perdVie()
#on appelle de nouveau la méthode afficheEtat pour l'instance gollum
gollum.afficheEtat()
#et pour l'instance bilbo
bilbo.afficheEtat()
```

A5

```
Il reste 20 points de vie à Gollum
Il reste 20 points de vie à Bilbo
Bilbo subit une attaque, il perd une vie
Il reste 20 points de vie à Gollum
Il reste 19 points de vie à Bilbo
```

Voilà, nous arrivons enfin à un programme qui respecte l'encapsulation. En dehors de la création des classes, un bon programme orienté objet, ne sera qu'une suite de création d'instances et d'appel de méthode pour ces instances (bien évidemment l'appel de telle ou telle méthode pourra se faire en fonction de certaines conditions, nous verrons cet aspect des choses un peu plus loin.

NB: Dans l'exemple 5 lors de la création de Bilbo et Gollum, on passe 20 comme argument nbreDeVie, bien évidemment, vous pouvez choisir n'importe quelle valeur (si par exemple au départ vous décidez que Gollum est en moins bon état que Bilbo, vous pouvez lui donner seulement 15 points de vie : gollum=personnage(15,'Gollum')

Mais que se passe-t-il si on oublie de passer des arguments au moment de la création d'une instance ?(ex: Gandalf=personnage()). Et bien, cela vous sort une belle erreur, du style : « lors de la création d'une instance de personnage, j'attends des arguments, or, ici il n'y en a pas ! »

Nous pouvons très facilement résoudre le problème en donnant des valeurs par défaut (si j'oublie les arguments se sont les valeurs données par défaut qui seront données aux attributs). Les valeurs par défaut sont données au moment de la création de la méthode __init__

ex6

```
class personnage(object):
        #méthode __init__ avec les 2 arguments et des valeurs données par défaut
        #si vous oubliez les arguments votre instance aura pour nom Sans Nom et 20 points de vie
        def __init__(self, nbreDeVie=20, nomDuPerso='Sans Nom'):
                self.vie=nbreDeVie
                self.nom=nomDuPerso
        def afficheEtat (self):
                print ('Il reste',self.vie,'point de vie à',self.nom)
        def perdVie (self):
                print (self.nom,'subit une attaque, il perd une vie')
                self.vie=self.vie-1
#on crée une nouvelle instance en "oubliant" les arguments.
gandalf=personnage()
#Et pourtant.......tout fonctionne avec les valeurs par défaut
gandalf.afficheEtat()
```

A6

```
Il reste 20 points de vie à Sans Nom
```

Notion d'héritage en POO

Pour terminer avec les objets, il nous reste une notion importante à aborder : l'héritage.
Nous allons créer une nouvelle classe (la classe magicien) qui héritera de la classe personnage (on dira que la classe magicien sera la classe fille et personnage sera la classe parente).
Pour une fois, ce terme d'héritage est bien approprié à la situation. En effet la classe magicien va hériter de toutes les méthodes et de tous les attributs de la classe personnage (un attribut de magicien aura un nom et des points de vie, comme un attribut de personnage, et pourra utiliser les méthodes de la classe personnage). Alors quel est l'intérêt de créer une nouvelle classe ? Eh bien, l'instance de magicien aura un attribut supplémentaire (pointDeMagie) et une méthode supplémentaire (lancerDesSorts). En tout, une instance de magicien aura 3 attributs (2 hérités de la classe personnage (vie et nom) et 1 qui lui est propre (pointDeMagie)) et 3 méthodes (2 hérités de la classe personnage et 1 propre)
La mise en place de l'héritage est très simple, puisqu'il suffit de remplacer object par le nom de la classe parente, dans notre exemple on définira la classe magicien comme suit : class magicien (personnage).

ex7

```
class personnage(object):
        #méthode __init__ avec 2 arguments (sans compter self !) le nombre de vies et le nom du
        #personnage
        def __init__(self, nbreDeVie, nomDuPerso):
                self.vie=nbreDeVie
                self.nom=nomDuPerso
        #voici la méthode qui affiche le nombre de vies du personnage.
        def affichePointVie (self):
                print ('Il reste',self.vie, 'points de vie à',self.nom)
        #voici la méthode qui fait perdre 1 point de vie au personnage qui a subi une attaque
        def perdVie (self):
                print (self.nom,'subit une attaque, il perd une vie')
                self.vie=self.vie-1
#la classe magicien hérite de la classe personnage
class magicien(personnage):
        #dans def_init_ on retrouve nbreDeVie et nomDuPerso comme dans le def __init__ de la
        #classe personnage
        def __init__(self, nbreDeVie, nomDuPerso, pointMagie):
                #la ligne suivante est très importante dans le cas d'héritage, il faut
                #systématiquement faire ce genre d'appel :
                #classeparente.__init__(self,arg1,arg2.....)
                personnage.__init__(self, nbreDeVie, nomDuPerso)
```

```
                #le seul nouvel attribut est self.magie, tous les autres sont hérités de la classe
                #personnage
                self.magie=pointMagie
        #une classe uniquement disponible pour les instances de magiciens
        def faireMagie (self):
                print (self.nom,'fait de la magie')
                self.magie=self.magie-1
        def affichePointMagie (self):
                print (self.nom,'a',self.magie, 'points de magie')
        #les autres méthodes des instances de magicien sont héritées de la classe personnage
#on crée une instance de magicien
gandalf=magicien(20,'Gandalf',15)
#applique la méthode affichePointVie à gandalf, cette méthode est héritée de la classe personnage
gandalf.affichePointVie()
gandalf.affichePointMagie()
#applique la méthode faireMagie à gandalf, cette méthode est uniquement applicable aux instances
#de la classe magicien
gandalf.faireMagie()
gandalf.affichePointMagie()
```

A7

```
Il reste 20 points de vie à Gandalf
Gandalf a 15 points de magie
Gandalf fait de la magie
Gandalf a 14 points de magie
```

Voici deux exemples avec des « if » et des « while » qui vont nous permettre de terminer le « combat ».
Nous utilisons la fonction random() (tirage au sort)

ex8

```
from random import *
class personnage(object):
        def __init__(self, nbreDeVie=20, nomDuPerso='Sans Nom'):
                self.vie=nbreDeVie
                self.nom=nomDuPerso
        def afficheEtat (self):
                print ('Il reste',self.vie,'point de vie à',self.nom)
        #Voici une nouvelle méthode qui a pour paramètre le nom de l'attaquant
        def subitAttaque (self,attaquant):
                print (attaquant,'attaque',self.nom)
                #tauxReussite va prendre une valeur aléatoire comprise entre 0 et 1 grâce à la
                #fonction random()
                tauxReussite=random()
                #Si tauxReussite est supérieur à 0,6 alors l'attaque réussit,on enlève 1 pt de vie
                #à celui qui subit l'attaque
                if tauxReussite>0.6:
                        print ("L'attaque de",attaquant,"a réussi,",self.nom,"perd une vie")
                        self.vie=self.vie-1
                #Sinon l'attaque échoue et celui qui subit l'attaque ne perd pas de pts de vie
                else:
                        print ("L'attaque de",attaquant,"a échoué")
bilbo=personnage(20,'Bilbo')
gollum=personnage(20,'Gollum')
bilbo.subitAttaque('Gollum') #Gollum est l'attaquant
bilbo.afficheEtat()
gollum.subitAttaque('Bilbo') #Bilbo est l'attaquant
gollum.afficheEtat()
```

A8

```
Gollum attaque Bilbo
L'attaque de Gollum a échoué
Il reste 20 points de vie à Bilbo
Bilbo attaque Gollum
L'attaque de Bilbo a réussi, Gollum perd une vie
Il reste 19 points de vie à Gollum
```

ex9

```
from random import *
class personnage(object):
        def __init__(self, nbreDeVie=20, nomDuPerso='Sans Nom'):
                self.vie=nbreDeVie
                self.nom=nomDuPerso
        def afficheEtat (self):
                print ('Il reste',self.vie,'point de vie à ',self.nom)
        def subitAttaque (self,attaquant):
                print (attaquant,'attaque',self.nom)
                tauxReussite=random()
                if tauxReussite>0.6:
                        print ("L'attaque de",attaquant,"a réussi,",self.nom,"perd une vie")
                        self.vie=self.vie-1
                else:
                        print ("L'attaque de",attaquant,"a échoué")
bilbo=personnage(2,'Bilbo')
gollum=personnage(2,'Gollum')
# tant que les points de vie des deux adversaires restent au-dessus de 0, les 4 méthodes suivantes
#sont appelées
while (bilbo.vie>0) & (gollum.vie>0) :
        #on vérifie si gollum a encore des points de vie
        if gollum.vie !=0 :
                bilbo.subitAttaque('Gollum') #Gollum est l'attaquant
                bilbo.afficheEtat()
        #on vérifie si Bilbo a encore des points de vie
        if bilbo.vie !=0 :
                gollum.subitAttaque('Bilbo') #Bilbo est l'attaquant
                gollum.afficheEtat()
#Un des 2 adversaires a 0 point de vie, nous sommes sortis de while et nous testons si les points
#de vie de Gollum sont à 0
if gollum.vie==0 :
        print ('Gollum est mort, Bilbo est vainqueur')
#sinon, c'est forcément Bilbo qui est mort
else :
        print ('Bilbo est mort, Gollum est vainqueur')
```

A9

```
Gollum  attaque  Bilbo
L'attaque de  Gollum  a réussi,  Bilbo  perd une vie
Il reste  1 point de vie à  Bilbo
Bilbo  attaque  Gollum
L'attaque de  Bilbo  a échoué
Il reste  2 points de vie à  Gollum
Gollum  attaque  Bilbo
L'attaque de  Gollum  a échoué
Il reste  1 point de vie à  Bilbo
Bilbo  attaque  Gollum
L'attaque de  Bilbo  a réussi,  Gollum  perd une vie
Il reste  1 point de vie à  Gollum
Gollum  attaque  Bilbo
L'attaque de  Gollum  a échoué
Il reste  1 point de vie à  Bilbo
Bilbo  attaque  Gollum
L'attaque de  Bilbo  a réussi,  Gollum  perd une vie
Il reste  0 point de vie à  Gollum
Gollum est mort, Bilbo est vainqueur
```

Voilà, maintenant, le programme se termine par la défaite d'un des deux personnages !