PROGRAMMATION ORIENTEE OBJET
Classes, Méthodes

Les **classes** que nous avons définies dans le chapitre précédent peuvent être considérées comme des espaces de noms particuliers, dans lesquels nous n'avons placé jusqu'ici que des variables (les **attributs d'instance**). Il nous faut à présent doter ces classes d'une **fonctionnalité**.

L'idée de base de la **P**rogrammation **O**rientée **O**bjet consiste en effet à regrouper dans un même ensemble (l'**objet**), à la fois un certain nombre de données (ce sont les **attributs d'instance**), et les **algorithmes** destinés à effectuer divers traitements sur ces données (ce sont les **méthodes**, à savoir des **fonctions particulières encapsulées dans l'objet**).

Objet = [attributs + méthodes]

Cette façon d'associer dans une même « capsule » les propriétés d'un objet et les fonctions qui permettent d'agir sur elles, correspond à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel qui nous entoure.

Considérons par exemple un **widget** « **bouton** » dans une application graphique. Il nous paraît raisonnable de souhaiter que l'objet informatique que nous appelons ainsi ait un comportement qui ressemble à celui d'un bouton d'appareil quelconque dans le monde réel.

Or nous savons que la fonctionnalité d'un bouton réel (sa capacité de fermer ou d'ouvrir un circuit électrique) est bien intégrée dans l'objet lui-même (au même titre que d'autres propriétés, telles que sa taille, sa couleur, etc.).

De la même manière, nous souhaiterons donc que les différentes caractéristiques de notre bouton logiciel (sa taille, son emplacement, sa couleur, le texte qu'il supporte), mais aussi la définition de ce qui se passe lorsque l'on effectue différentes actions de la souris sur ce bouton, soient regroupés dans une entité bien précise à l'intérieur du programme, de telle sorte qu'il n'y ait pas de confusion entre ce bouton et un autre, ou a fortiori entre ce bouton et d'autres entités.

1. Définition d'une méthode

Pour illustrer notre propos, nous allons définir une nouvelle **classe**, la classe « **Time()** », laquelle devrait nous permettre d'effectuer toute une série d'opérations sur des instants, des durées, etc.

```
class Time(object):
    "définition d'objets temporels"
    pass
```

D'après Apprendre à programmer avec Python de Gérard Swinnen :

La commande « **pass** » ne fait rien, elle permet juste ici d'éviter un message d'erreur lors de la compilation tant que nous n'avons rien demandé de faire dans une nouvelle classe créée.

Créons à présent un objet de ce type, et ajoutons-lui des variables d'instance pour mémoriser les heures, minutes et secondes.

```
instant = Time()
instant.heure = 11
instant.minute = 34
instant.seconde = 25
```

Écrivez une fonction « **affiche_heure(t)** » , afin de faire afficher le contenu d'un objet (ici l'objet « **instant** » de classe **Time()** sous la forme conventionnelle :« **heures : minutes : secondes** ».

Appliquée à l'objet « **instant** » créé ci-dessus, cette fonction devrait donc afficher

```
*** Remote Interpreter Reinitialized ***
>>> affiche_heure(instant)
11 : 34 : 25
```

Si par la suite vous deviez utiliser fréquemment des **objets** de la classe **Time()**, cette fonction d'affichage vous serait probablement fort utile.
Il serait donc judicieux d'arriver à « **encapsuler** » cette fonction « **affiche_heure(t)** » dans la classe **Time()** elle-même, de manière à s'assurer qu'elle soit toujours automatiquement disponible, chaque fois que l'on aura à manipuler des objets de la classe Time().
Une **fonction** que l'on aura ainsi **encapsulée** dans une classe s'appelle préférentiellement une **méthode**. Comment construire une telle fonction ?

1.1 Définition concrète d'une méthode dans un script

On définit une méthode comme on définit une fonction, c'est-à-dire en écrivant un bloc d'instructions à la suite du mot réservé « **def** », mais cependant avec deux différences :

- La définition d'une méthode est toujours placée **à l'intérieur** de la définition **d'une classe**, de manière à ce que la relation qui lie la méthode à la classe soit clairement établie.
- La définition d'une méthode doit toujours **comporter au moins un paramètre**, (alors qu'une fonction peut n'en comporter aucun) lequel doit être une **référence d'instance**, et ce paramètre particulier doit toujours être listé en premier.

On peut en principe utiliser un nom de variable quelconque pour ce premier paramètre, mais il est vivement conseillé de respecter la convention qui consiste à toujours lui donner le nom : « **self** ».

Ce paramètre « **self** » est nécessaire, parce qu'il faut pouvoir désigner l'instance à laquelle la méthode sera associée, dans les instructions faisant partie de sa définition.

Voyons comment cela se passe en pratique :

Pour faire en sorte que la fonction « **affiche_heure(t)** » devienne une méthode de la classe **Time()**, il nous suffit de **déplacer sa définition à l'intérieur** de celle de **la classe** :

```
class Time(object):
    "définition d'objets temporels"
    def affiche_heure(t):
        print("{0}:{1}:{2}".format(t.heure, t.minute, t.seconde))
```

Techniquement, c'est tout à fait suffisant, car le paramètre t peut parfaitement désigner l'instance à laquelle seront attachés les attributs heure, minute et seconde. Il est cependant fortement recommandé de changer son nom en « **self** »

```
class Time(object):
    "définition d'objets temporels"
    def affiche_heure(self):
        print("{0}:{1}:{2}".format(self.heure, self.minute, self.seconde))
```

La définition de la **méthode** « **affiche_heure(self)** » fait maintenant partie du bloc d'instructions indentées suivant l'instruction « **class** » (et dont fait partie aussi la chaîne documentaire « Nouvelle classe temporelle »).

1.2 Essai de la méthode, dans une instance quelconque

Nous disposons donc dèsormais d'une **classe** « **Time()** », dotée d'une **méthode** « **affiche_heure(self)** ». En principe, nous devons maintenant pouvoir créer des objets de cette classe, et leur appliquer cette méthode.

Pour ce faire, commençons par instancier un objet appelé « **maintenant** » et à lui affecter les variables « **heure, minute**, **seconde** »

```
maintenant=Time()
maintenant.heure=13
maintenant.minute=34
maintenant.seconde=21
```

Tester alors cette nouvelle méthode.

```
*** Remote Interpreter Reinitialized ***
>>> maintenant.affiche_heure()
13:34:21
```

Nous avons cependant déjà signalé qu'il n'est pas recommandable de créer ainsi des attributs d'instance par assignation directe en dehors de l'objet lui-même.

Voyons donc à présent comment nous pouvons mieux faire.

2. La méthode « constructeur »

Il est judicieux que les **variables d'instance** soient définies elles aussi **à l'intérieur de la classe**, avec pour chacune d'elles une **valeur « par défaut »**. Pour obtenir cela, nous allons faire appel à une **méthode particulière**, que l'on désignera par la suite sous le nom de **constructeur**.

Une **méthode « constructeur »** est exécutée automatiquement lorsque l'on instancie un nouvel objet à partir de la classe. On peut donc y placer tout ce qui semble nécessaire pour initialiser automatiquement l'objet que l'on crée.

Afin qu'elle soit reconnue comme telle par Python, la **méthode « constructeur »** devra obligatoirement s'appeler « **__init__** »
« **deux caractères 'souligné'** » (tiret du 8), le mot « **init** », puis encore « **deux caractères 'souligné'** ».

Exemple :

```
class Time1(object):
    "Encore une nouvelle classe temporelle"
    def __init__(self):
        self.heure =12
        self.minute =0
        self.seconde =0
    def affiche_heure(self):
        print("{}:{}:{}".format(self.heure, self.minute, self.seconde))
```

Comme précédemment, créons un objet de la classe « **Time()** »

```
tstart1 = Time1()
```

Et testons-en la méthode « **affiche_heure()** ».

```
*** Remote Interpreter Reinitialized ***
>>> tstart1.affiche_heure()
12:0:0
```

Lors de son instanciation, l'objet « **tstart1** » s'est vu attribuer automatiquement les trois **attributs « heure »**, **« minute »** et **« seconde »** par la méthode **« constructeur »**, avec '12', '0' et '0' par défaut.
Dès lors qu'un objet de cette classe existe, on peut donc tout de suite demander l'affichage de ces attributs.

L'intérêt de cette technique apparaît plus clairement si nous ajoutons encore quelque chose. Comme toute méthode, la méthode « **__init__()** » peut être dotée de **paramètres**.
Dans le cas de cette méthode particulière qu'est le constructeur, les paramètres peuvent jouer un rôle très intéressant, parce qu'ils vont permettre d'initialiser certaines de ses variables d'instance au moment même de l'instanciation de l'objet.

Veuillez donc reprendre l'exemple précédent, en modifiant la définition de la méthode « **__init__()** » comme suit :

```
class Time2(object):
    "Encore une nouvelle classe temporelle"
    def __init__(self, hh=12, mm=0, ss=0):
        self.heure = hh
        self.minute = mm
        self.seconde = ss
    def affiche_heure(self):
        print("{}:{}:{}".format(self.heure, self.minute, self.seconde))
```

Notre nouvelle méthode « **__init__()** » comporte à présent 3 paramètres, avec pour chacun une valeur par défaut. Nous obtenons ainsi une classe encore plus perfectionnée.
Lorsque nous instancions un objet de cette classe, nous pouvons maintenant initialiser ses principaux attributs à l'aide d'arguments, au sein même de l'instruction d'instanciation.
Et si nous omettons tout ou partie d'entre eux, les attributs reçoivent de toute manière des valeurs par défaut. Lorsque l'on écrit l'instruction d'instanciation d'un nouvel objet, et que l'on veut lui transmettre des arguments, il suffit de placer ceux-ci dans les parenthèses qui accompagnent le nom de la classe.
On procède donc exactement de la même manière que lorsque l'on invoque une fonction quelconque.

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet « **Time()** ».

```
recreation=Time2(9, 50,0)
```

Et testons-en la méthode « **affiche_heure()** ».

```
*** Remote Interpreter Reinitialized ***
>>> recreation.affiche_heure()
9:50:0
```

Lorsque les variables d'instance possèdent des valeurs par défaut, nous pouvons aussi bien créer des objets « **Time()** » en omettant un ou plusieurs arguments.

```
reprise=Time2(10, 5)
```

Et testons-en la méthode « **affiche_heure()** ».

```
*** Remote Interpreter Reinitialized ***
>>> reprise.affiche_heure()
10:5:0
```

Ou encore :

```
fin=Time2(hh=18)
```

Et testons-en la méthode « **affiche_heure()** ».

```
*** Remote Interpreter Reinitialized ***
>>> fin.affiche_heure()
18:0:0
```

3. Exercices

3.1 Domino

Définissez une classe « **Domino()** » qui permette d'instancier deux objets « **domino1** » et « **domino2** » simulant 2 pièces d'un jeu de dominos.

Le constructeur de cette classe initialisera (Méthode « __init__ ») les valeurs des points (« pointA » et « pointB ») présents sur les deux faces A et B du domino (valeurs par défaut = 0).

Deux autres méthodes seront définies :

- Une méthode « **affiche_points()** » qui affiche les points présents sur les deux faces.
- Une méthode « **valeur()** » qui renvoie la somme des points présents sur les 2 faces.

Pour tester le programme, vous utiliserez les dominos :

- domino1 = Domino(2,6)
- domino2 = Domino(4,3)

Résultat attendu :

```
*** Remote Interpreter Reinitialized ***
>>> domino1.affiche_point()
face A:  2
face B:  6
>>> domino2.affiche_point()
face A:  4
face B:  3
>>> domino1.valeur()
8
>>> domino2.valeur()
7
>>>
```

3.2 Compte Bancaire

Définissez une classe « **CompteBancaire()** », qui permette d'instancier des objets tels que « **compte1** », « **compte2** », …

Le constructeur de cette classe initialisera (Méthode « __init__ ») les attributs d'instance (« nom » et « solde ») avec les valeurs par défaut 'DUPONT' et 1000.

Trois autres méthodes seront définies :

- Une méthode « **depot(somme)** » qui permet d'ajouter une somme donnée au compte.
- Une méthode « **retrait(somme)** » qui permet de retirer une somme donnée du compte.
- Une méthode « **affiche()** » qui permet d'afficher le nom du titulaire et le solde de son compte.

Pour tester le programme, vous utiliserez :

- compte1 = CompteBancaire('DURAND', 800)
- compte1.depot(350)
- compte1.retrait(200)

Résultat attendu :

```
*** Remote Interpreter Reinitialized ***
>>> compte1.affiche()
'Le solde du compte bancaire de DURAND est de 950 euros.'
>>>
```

3.3 Voiture

Définissez une classe « **Voiture()** » qui permette d'instancier des objets reproduisant le comportement de voitures automobiles.
Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :

marque = 'Ford' ; couleur = 'rouge' ; pilote = 'personne' ; vitesse = 0

Lorsque l'on instanciera un nouvel objet « Voiture », on pourra choisir sa marque et sa couleur, mais pas son conducteur ni sa vitesse.

Les méthodes suivantes seront à définir :

- Une méthode « **choix_conducteur(nom)** » qui permet de désigner ou de changer le conducteur.
- Une méthode « **accelerer(taux, duree)** », qui permet de faire varier la vitesse de la voiture. La variation de vitesse obtenue sera égale au produit « **taux × duree** ». Par exemple, si la voiture accélère avec un taux de 1,3 m/s pendant 20 secondes, son gain de vitesse est égal à 26 m/s. Des taux négatifs sont acceptés, (ce qui permet de ralentir). La variation de vitesse ne peut pas être acceptée si le conducteur est « personne » !

- Une méthode « affiche_tout » qui permet de faire apparaître les propriétés de la voiture, c'est-à-dire sam arque, sa couleur, le nom du conducteur et sa vitesse.

Pour tester le programme, vous utiliserez :
- voiture1 = Voiture('Peugeot', 'bleue')
- voiture2 = Voiture(couleur = 'verte')
- voiture3 = Voiture('Mercedes')
- voiture1.choix_conducteur('Roméo')
- voiture2.choix_conducteur('Juliette')
- voiture2.accelerer(1.8, 12)
- voiture3.accelerer(1.9, 11)

Résultat attendu :

```
*** Remote Interpreter Reinitialized ***
>>> voiture1.affiche_tout()
La voiture de marque Peugeot et de couleur bleue est pilotée par Roméo à la vitese de 0 m/s.
>>> voiture2.affiche_tout()
La voiture de marque Ford et de couleur verte est pilotée par Juliette à la vitese de 21.6 m/s.
>>> voiture3.affiche_tout()
La voiture de marque Mercedes et de couleur rouge est pilotée par personne à la vitese de 0 m/s.
>>>
```

3.4 Satellite

Définissez une classe « **Satellite()** » qui permet d'instancier des objets simulant des satellites artificiels lancés dans l'espace et en rotation autour de la Terre. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :

masse = 100 ; vitesse = 0

Lorsque l'on instanciera un nouvel objet « **Satellite()** », on pourra choisir son nom, sa masse et sa vitesse.
Les méthodes suivantes devront définies :
- Une méthode « **impulsion(force, duree)** » qui permettra de faire varier la vitesse du satellite. La variation de vitesse « **Δv** » subie par un objet de masse « **m** » soumis à l'action d'une force « **F** » pendant une durée « **Δt** » est égale à « **(F×Δt)/m** ». Par exemple, un satellite de masse 300 kg qui subit une force de 600 N pendant 10 secondes voit sa vitesse varier de 20 m/s.
- Une méthode « **affiche_vitesse()** » qui affiche le nom du satellite et sa vitesse.
- Une méthode « **energie()** » qui renverra la valeur de l'énergie cinétique du satellite. L'énergie cinétique « **Ec** » du satellite est égale à la moitié du produit de sa masse multipliée par le carré de la vitesse.

Les valeurs affichées devront être arrondies au dixième.

Pour tester le programme, vous créerez les deux satellites suivants :

- satellite1 : nom : Zoé, masse = 250 kg, vitesse = 10 m/s
- impulsion : F = 500 N, Δt = 15 s

- satellite2 = nom : Hercule, masse = 350 kg, vitesse = 20 m/s
- impulsion : F = 600 N, Δt = 20 s

Résultats attendus :

```
*** Remote Interpreter Reinitialized ***
>>> satellite1.energie()
200000.0
>>> satellite1.affiche_vitesse()
La vitesse du satellite Zoé est égale à 40.0 m/s.
>>> satellite2.energie()
515714.3
>>> satellite2.affiche_vitesse()
La vitesse du satellite Hercule est égale à 54.3 m/s.
>>>
```

D'après Apprendre à programmer avec Python de Gérard Swinnen :