12

Classes, méthodes, héritage

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 programmation orientée objet 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 chez les concepteurs de programmes 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.

Définition d'une méthode

Pour illustrer notre propos, nous allons définir une nouvelle 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"
```

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
```

À titre d'exercice, écrivez maintenant vous-même une fonction **affiche_heure()** , qui serve à visualiser le contenu d'un objet de classe **Time()** sous la forme conventionnelle « heures:minutes:secondes ». Appliquée à l'objet instant créé ci-dessus, cette fonction devrait donc afficher `11:34:25` :

```
>>> affiche_heure(instant)
11:34:25
```

Votre fonction ressemblera probablement à ceci :

```
>>> def affiche_heure(t):
...     print(str(t.heure) +":" +str(t.minute) +":" +str(t.seconde))
```

... ou mieux encore, à ceci :

```
>>> def affiche_heure(t):
...     print("{}:{}:{}".format(t.heure, t.minute, t.seconde))
```

en application de la technique de formatage des chaînes décrite à la page 140.

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()** 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*.

Vous avez évidemment déjà rencontré des méthodes à de nombreuses reprises dans les chapitres précédents de cet ouvrage, et vous savez donc déjà qu'une méthode est bien une fonction associée à une classe particulière d'objets. Il vous reste seulement à apprendre comment construire une telle fonction.

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, lequel doit être *une référence d'instance*, et ce paramètre particulier doit toujours être listé en premier.

Vous pourriez 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. Vous comprendrez cela plus facilement avec les exemples ci-après.

> *Remarquons que la définition d'une méthode comporte toujours au moins un paramètre : self, alors que la définition d'une fonction peut n'en comporter aucun.*

Voyons comment cela se passe en pratique :

Pour faire en sorte que la fonction **affiche_heure()** 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):
...     "Nouvelle classe temporelle"
...     def affiche_heure(t):
...         print("{}:{}:{}".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**. Étant donné son rôle particulier, il est cependant fortement recommandé de changer son nom en **self** :

```
>>> class Time(object):
...     "Nouvelle classe temporelle"
...     def affiche_heure(self):
...         print("{}:{}:{}".format(self.heure, self.minute, self.seconde))
```

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

Essai de la méthode, dans une instance quelconque

Nous disposons donc dès à présent d'une classe **Time()**, dotée d'une méthode **affiche_heure()**. En principe, nous devons maintenant pouvoir créer des objets de cette classe, et leur appliquer cette méthode. Voyons si cela fonctionne. Pour ce faire, commençons par instancier un objet :

```
>>> maintenant = Time()
```

Si nous essayons un peu trop vite de tester notre nouvelle méthode sur cet objet, cela ne marche pas :

```
>>> maintenant.affiche_heure()
AttributeError: 'Time' object has no attribute 'heure'
```

C'est normal : nous n'avons pas encore créé les attributs d'instance. Il faudrait faire par exemple :

```
>>> maintenant.heure = 13
>>> maintenant.minute = 34
>>> maintenant.seconde = 21
```

... et réessayer. À présent, ça marche :

```
>>> maintenant.affiche_heure()
13:34:21
```

À plusieurs reprises, 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. Entre autres désagréments, cela conduirait fréquemment à des erreurs comme celle que nous venons de rencontrer. Voyons donc à présent comment nous pouvons mieux faire.

La méthode constructeur

L'erreur que nous avons rencontrée au paragraphe précédent est-elle évitable ?

Elle ne se produirait effectivement pas, si nous nous étions arrangés pour que la méthode **affiche_heure()** puisse toujours afficher quelque chose, sans qu'il ne soit nécessaire d'effectuer au préalable une manipulation sur l'objet nouvellement créé. En d'autres termes, il serait judicieux que *les variables d'instance soient prédéfinies* 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 a ceci de particulier qu'*elle 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 », le mot **init**, puis encore deux caractères « souligné »).

Exemple

```
>>> class Time(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 cette classe et testons-en la méthode **affiche_heure()** :

```
>>> tstart = Time()
>>> tstart.affiche_heure()
12:0:0
```

Nous n'obtenons plus aucune erreur, cette fois. En effet : lors de son instanciation, l'objet **tstart** s'est vu attribuer automatiquement les trois attributs **heure**, **minute** et **seconde** par la méthode constructeur, avec 12 et zéro comme valeurs 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 apparaîtra plus clairement si nous ajoutons encore quelque chose.

Comme toute méthode qui se respecte, la méthode **__init__()** peut être dotée de paramètres. Et 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 :

```
...     def __init__(self, hh =12, mm =0, ss =0):
...         self.heure =hh
...         self.minute =mm
...         self.seconde =ss
```

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 = Time(10, 15, 18)
>>> recreation.affiche_heure()
10:15:18
```

Puisque les variables d'instance possèdent maintenant des valeurs par défaut, nous pouvons aussi bien créer de tels objets **Time()** en omettant un ou plusieurs arguments :

```
>>> rentree = Time(10, 30)
>>> rentree.affiche_heure()
10:30:0
```

ou encore :

```
>>> rendezVous = Time(hh =18)
>>> rendezVous.affiche_heure()
18:0:0
```

Exercices

12.1 Définissez une classe **Domino()** qui permette d'instancier des objets simulant les pièces d'un jeu de dominos. Le constructeur de cette classe initialisera les valeurs des points

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.

Exemples d'utilisation de cette classe :

```
>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.affiche_points()
face A : 2  face B : 6
>>> d2.affiche_points()
face A : 4  face B : 3
>>> print("total des points :", d1.valeur() + d2.valeur())
15
>>> liste_dominos = []
>>> for i in range(7):
...     liste_dominos.append(Domino(6, i))
>>> print(liste_dominos[3])
<__main__.Domino object at 0xb758b92c>
```

etc.

12.2 Définissez une classe **CompteBancaire()**, qui permette d'instancier des objets tels que **compte1**, **compte2**, etc. Le constructeur de cette classe initialisera deux attributs d'instance **nom** et **solde**, avec les valeurs par défaut 'Dupont' et 1000.
Trois autres méthodes seront définies :
- **depot(somme)** permettra d'ajouter une certaine somme au solde ;
- **retrait(somme)** permettra de retirer une certaine somme du solde ;
- **affiche()** permettra d'afficher le nom du titulaire et le solde de son compte.

Exemples d'utilisation de cette classe :

```
>>> compte1 = CompteBancaire('Duchmol', 800)
>>> compte1.depot(350)
>>> compte1.retrait(200)
>>> compte1.affiche()
Le solde du compte bancaire de Duchmol est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```

Espaces de noms des classes et instances

Vous avez appris précédemment (voir page 66) que les variables définies à l'intérieur d'une fonction sont des variables locales, inaccessibles aux instructions qui se trouvent à l'extérieur de la fonction. Cela vous permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence.

Pour décrire la même chose en d'autres termes, nous pouvons dire que chaque fonction possède son propre *espace de noms*, indépendant de l'espace de noms principal.

Vous avez appris également que les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais *en consultation seulement* : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier (à moins de faire appel à l'instruction **global**).

Il existe donc une sorte de hiérarchie entre les espaces de noms. Nous allons constater la même chose à propos des classes et des objets. En effet :

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées *variables de classe* ou *attributs de classe*.
- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées *variables d'instance* ou *attributs d'instance*.
- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal.
- Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

Considérons par exemple la classe **Time()** définie précédemment. À la page 177, nous avons instancié trois objets de cette classe : **recreation**, **rentree** et **rendezVous**. Chacun a été initialisé avec des valeurs différentes, indépendantes. Nous pouvons modifier et réafficher ces valeurs à volonté dans chacun de ces trois objets, sans que l'autre n'en soit affecté :

```
>>> recreation.heure = 12
>>> rentree.affiche_heure()
10:30:0
>>> recreation.affiche_heure()
12:15:18
```

Veuillez à présent encoder et tester l'exemple ci-dessous :

```
>>> class Espaces(object):                          # 1
...     aa = 33                                     # 2
...     def affiche(self):                        # 3
...         print(aa, Espaces.aa, self.aa)          # 4
...
>>> aa = 12                                         # 5
>>> essai = Espaces()                               # 6
>>> essai.aa = 67                                   # 7
>>> essai.affiche()                                 # 8
12 33 67
>>> print(aa, Espaces.aa, essai.aa)                 # 9
12 33 67
```

Dans cet exemple, le même nom **aa** est utilisé pour définir trois variables différentes : une dans l'espace de noms de la classe (à la ligne 2), une autre dans l'espace de noms principal (à la ligne 5), et enfin une dernière dans l'espace de nom de l'instance (à la ligne 7).
La ligne 4 et la ligne 9 montrent comment vous pouvez accéder à ces trois espaces de noms (de l'intérieur d'une classe, ou au niveau principal), en utilisant la qualification par points. Notez encore une fois l'utilisation de **self** pour désigner l'instance à l'intérieur de la définition d'une classe.

Héritage

Les classes constituent le principal outil de la programmation orientée objet (*Object Oriented Programming ou OOP*), qui est considérée de nos jours comme la technique de programmation la plus performante. L'un des principaux atouts de ce type de programmation réside dans le fait que l'on peut toujours se servir d'une classe préexistante pour en créer une nouvelle, qui *héritera* toutes ses propriétés mais pourra modifier certaines d'entre elles et/ou y ajouter les siennes propres. Le procédé s'appelle *dérivation*. Il permet de créer toute une hiérarchie de classes allant du général au particulier.

Nous pouvons par exemple définir une classe **Mammifere()**, qui contienne un ensemble de caractéristiques propres à ce type d'animal. À partir de cette classe *parente*, nous pouvons dériver une ou plusieurs classes *filles*, comme : une classe **Primate()**, une classe **Rongeur()**, une classe **Carnivore()**, etc., qui hériteront toutes les caractéristiques de la classe **Mammifere()**, en y ajoutant leurs spécificités.

Au départ de la classe **Carnivore()**, nous pouvons ensuite dériver une classe **Belette()**, une classe **Loup()**, une classe **Chien()**, etc., qui hériteront encore une fois toutes les caractéristiques de la classe parente avant d'y ajouter les leurs. Exemple :

```
>>> class Mammifere(object):
...     caract1 = "il allaite ses petits ;"

>>> class Carnivore(Mammifere):
...     caract2 = "il se nourrit de la chair de ses proies ;"

>>> class Chien(Carnivore):
...     caract3 = "son cri s'appelle aboiement ;"

>>> mirza = Chien()
>>> print(mirza.caract1, mirza.caract2, mirza.caract3)
il allaite ses petits ; il se nourrit de la chair de ses proies ;
son cri s'appelle aboiement ;
```

Dans cet exemple, nous voyons que l'objet **mirza** , qui est une instance de la classe **Chien()**, hérite non seulement l'attribut défini pour cette classe, mais également les attributs définis pour les classes parentes.

Vous voyez également dans cet exemple comment il faut procéder pour dériver une classe à partir d'une classe parente : on utilise l'instruction **class**, suivie comme d'habitude du nom que l'on veut attribuer à la nouvelle classe, et on place entre parenthèses le nom de la classe parente. Les classes les plus fondamentales dérivent quant à elles de l'objet « ancêtre » **object**.

Notez bien que les attributs utilisés dans cet exemple sont des attributs des classes (et non des attributs d'instances). L'instance **mirza** peut accéder à ces attributs, mais pas les modifier :

```
>>> mirza.caract2 = "son corps est couvert de poils ;"   # 1
>>> print(mirza.caract2)                                 # 2
son corps est couvert de poils ;                         # 3
>>> fido = Chien()                                       # 4
>>> print(fido.caract2)                                  # 5
il se nourrit de la chair de ses proies ;                # 6
```

Dans ce nouvel exemple, la ligne 1 ne modifie pas l'attribut **caract2** de la classe **Carnivore()**, contrairement à ce que l'on pourrait penser au vu de la ligne 3. Nous pouvons le vérifier en créant une nouvelle instance **fido** (lignes 4 à 6).

Si vous avez bien assimilé les paragraphes précédents, vous aurez compris que l'instruction de la ligne 1 crée une nouvelle variable d'instance associée seulement à l'objet **mirza**. Il existe donc dès ce moment deux variables avec le même nom **caract2** : l'une dans l'espace de noms de l'objet **mirza**, et l'autre dans l'espace de noms de la classe **Carnivore()**.

Comment faut-il alors interpréter ce qui s'est passé aux lignes 2 et 3 ?

Comme nous l'avons vu plus haut, l'instance **mirza** peut accéder aux variables situées dans son propre espace de noms, mais aussi à celles qui sont situées dans les espaces de noms de toutes les classes parentes. S'il existe des variables aux noms identiques dans plusieurs de ces espaces, laquelle sera sélectionnée lors de l'exécution d'une instruction comme celle de la ligne 2 ?

Pour résoudre ce conflit, Python respecte une règle de priorité fort simple. Lorsqu'on lui demande d'utiliser la valeur d'une variable nommée **alpha**, par exemple, il commence par rechercher ce nom dans l'espace local (le plus « interne », en quelque sorte). Si une variable **alpha** est trouvée dans l'espace local, c'est celle-là qui est utilisée, et la recherche s'arrête. Sinon, Python examine l'espace de noms de la structure parente, puis celui de la structure grand-parente, et ainsi de suite jusqu'au niveau principal du programme.

À la ligne 2 de notre exemple, c'est donc la variable d'instance qui sera utilisée. À la ligne 5, par contre, c'est seulement au niveau de la classe grand-parente qu'une variable répondant au nom **caract2** peut être trouvée. C'est donc celle-là qui est affichée.

Héritage et polymorphisme

Analysez soigneusement le script de la page suivante. Il met en œuvre plusieurs concepts décrits précédemment, en particulier le concept d'héritage.

Pour bien comprendre ce script, il faut cependant d'abord vous rappeler quelques notions élémentaires de *chimie*. Dans votre cours de chimie, vous avez certainement dû apprendre que les *atomes* sont des entités, constitués d'un certain nombre de *protons* (particules chargées d'électricité positive), *d'électrons* (chargés négativement) et de *neutrons* (neutres).

Le type d'atome (ou élément) est déterminé par le nombre de protons, que l'on appelle également *numéro atomique*. Dans son état fondamental, un atome contient autant d'électrons que de protons, et par conséquent il est électriquement neutre. Il possède également un nombre variable de neutrons, mais ceux-ci n'influencent en aucune manière la charge électrique globale.

Dans certaines circonstances, un atome peut gagner ou perdre des électrons. Il acquiert de ce fait une charge électrique globale, et devient alors un *ion* (il s'agit d'un *ion négatif* si l'atome a gagné un ou plusieurs électrons, et d'un *ion positif* s'il en a perdu). La charge électrique d'un ion est égale à la différence entre le nombre de protons et le nombre d'électrons qu'il contient.

Le script reproduit à la page suivante génère des objets **Atome()** et des objets **Ion()**. Nous avons rappelé ci-dessus qu'un ion est simplement un atome modifié. Dans notre programmation, la

classe qui définit les objets **Ion()** sera donc une *classe dérivée* de la classe **Atome()** : elle héritera d'elle tous ses attributs et toutes ses méthodes, en y ajoutant les siennes propres.

L'une de ces méthodes ajoutées (la méthode **affiche()**) remplace une méthode de même nom héritée de la classe **Atome()**. Les classes **Atome()** et **Ion()** possèdent donc chacune une méthode de même nom, mais qui effectuent un travail différent. On parle dans ce cas de *polymorphisme*. On pourra dire également que la méthode **affiche()** de la classe **Atome()** a été *surchargée*.

Il sera évidemment possible d'instancier un nombre quelconque d'atomes et d'ions à partir de ces deux classes. Or l'une d'entre elles, la classe **Atome()**, doit contenir une version simplifiée du tableau périodique des éléments (tableau de Mendeleïev), de façon à pouvoir attribuer un nom d'élément chimique, ainsi qu'un nombre de neutrons, à chaque objet généré. Comme il n'est pas souhaitable de recopier tout ce tableau dans chacune des instances, nous le placerons dans un *attribut de classe* . Ainsi ce tableau n'existera qu'en un seul endroit en mémoire, tout en restant accessible à tous les objets qui seront produits à partir de cette classe.

Voyons concrètement comment toutes ces idées s'articulent :

```
class Atome:
    """atomes simplifiés, choisis parmi les 10 premiers éléments du TP"""
    table =[None, ('hydrogène',0),('hélium',2),('lithium',4),
            ('béryllium',5),('bore',6),('carbone',6),('azote',7),
            ('oxygène',8),('fluor',10),('néon',10)]

    def __init__(self, nat):
        "le n° atomique détermine le n. de protons, d'électrons et de neutrons"
        self.np, self.ne = nat, nat           # nat = numéro atomique
        self.nn = Atome.table[nat][1]

    def affiche(self):
        print()
        print("Nom de l'élément :", Atome.table[self.np][0])
        print("%s protons, %s électrons, %s neutrons" % \
                  (self.np, self.ne, self.nn))

class Ion(Atome):
    """les ions sont des atomes qui ont gagné ou perdu des électrons"""

    def __init__(self, nat, charge):
        "le n° atomique et la charge électrique déterminent l'ion"
        Atome.__init__(self, nat)
        self.ne = self.ne - charge
        self.charge = charge

    def affiche(self):
        Atome.affiche(self)
        print("Particule électrisée. Charge =", self.charge)

### Programme principal : ###

a1 = Atome(5)
a2 = Ion(3, 1)
a3 = Ion(8, -2)
a1.affiche()
a2.affiche()
```

```
a3.affiche()
```

L'exécution de ce script provoque l'affichage suivant :

```
Nom de l'élément : bore
5 protons, 5 électrons, 6 neutrons

Nom de l'élément : lithium
3 protons, 2 électrons, 4 neutrons
Particule électrisée. Charge = 1

Nom de l'élément : oxygène
8 protons, 10 électrons, 8 neutrons
Particule électrisée. Charge = -2
```

Au niveau du programme principal, vous pouvez constater que l'on instancie les objets **Atome()** en fournissant leur numéro atomique (lequel doit être compris entre 1 et 10). Pour instancier des objets **Ion()**, par contre, on doit fournir un numéro atomique et une charge électrique globale (positive ou négative). La même méthode **affiche()** fait apparaître les propriétés de ces objets, qu'il s'agisse d'atomes ou d'ions, avec dans le cas de l'ion une ligne supplémentaire (*polymorphisme*).

Commentaires

La définition de la classe **Atome()** commence par l'assignation de la variable **table**. Une variable définie à cet endroit fait partie de l'espace de noms de la classe. C'est donc un *attribut de classe*, dans lequel nous plaçons une liste d'informations concernant les 10 premiers éléments du tableau périodique de *Mendeleïev*.

Pour chacun de ces éléments, la liste contient un tuple : (nom de l'élément, nombre de neutrons), à l'indice qui correspond au numéro atomique. Comme il n'existe pas d'élément de numéro atomique zéro, nous avons placé à l'indice zéro dans la liste, l'objet spécial *None*. Nous aurions pu placer à cet endroit n'importe quelle autre valeur, puisque cet indice ne sera pas utilisé. L'objet *None* de Python nous semble cependant particulièrement explicite.

Viennent ensuite les définitions de deux méthodes :

- Le constructeur **__init__()** sert essentiellement ici à générer trois *attributs d'instance*, destinés à mémoriser respectivement les nombres de protons, d'électrons et de neutrons pour chaque objet atome construit à partir de cette classe (rappelez-vous que les attributs d'instance sont des variables liées au paramètre **self**).
  Notez au passage la technique utilisée pour obtenir le nombre de neutrons à partir de l'attribut de classe, en mentionnant le nom de la classe elle-même dans une qualification par points, comme dans l'instruction : `self.nn = Atome.table[nat][1]`.
- La méthode **affiche()** utilise à la fois les attributs d'instance, pour retrouver les nombres de protons, d'électrons et de neutrons de l'objet courant, et l'attribut de classe (lequel est commun à tous les objets) pour en extraire le nom d'élément correspondant.

La définition de la classe **Ion()** inclut dans ses parenthèses le nom de la classe **Atome()** qui précède.

Les méthodes de cette classe sont des variantes de celles de la classe **Atome()**. Elles devront donc vraisemblablement faire appel à celles-ci. Cette remarque est importante : comment peut-on, à l'intérieur de la définition d'une classe, faire appel à une méthode définie dans une autre classe ?

Il ne faut pas perdre de vue, en effet, qu'une méthode se rattache toujours à l'instance qui sera générée à partir de la classe (instance représentée par **self** dans la définition). Si une méthode doit faire appel à une autre méthode définie dans une autre classe, il faut pouvoir lui transmettre la référence de l'instance à laquelle elle doit s'associer. Comment faire ? C'est très simple :

> *Lorsque dans la définition d'une classe, on souhaite faire appel à une méthode définie dans une autre classe, il suffit de l'invoquer directement, via cette autre classe, en lui transmettant la référence de l'instance comme premier argument.*

C'est ainsi que dans notre script, par exemple, la méthode **affiche()** de la classe **Ion()** peut faire appel à la méthode **affiche()** de la classe **Atome()** : les informations affichées seront bien celles de l'objet-ion courant, puisque sa référence a été transmise dans l'instruction d'appel :

```
        Atome.affiche(self)
```

Dans cette instruction, **self** est bien entendu la référence de l'instance courante.

De la même manière (vous en verrez de nombreux autres exemples plus loin), la méthode constructeur de la classe **Ion()** fait appel à la méthode constructeur de sa classe parente, dans :

```
        Atome.__init__(self, nat)
```

Cet appel est nécessaire, afin que les objets de la classe **Ion()** soient initialisés de la même manière que les objets de la classe **Atome()**. Si nous n'effectuons pas cet appel, les objets-ions n'hériteront pas automatiquement les attributs **ne**, **np** et **nn**, car ceux ci sont des *attributs d'instance* créés par la méthode constructeur de la classe **Atome()**, et celle-ci *n'est pas invoquée automatiquement* lorsqu'on instancie des objets d'une classe dérivée.

Comprenez donc bien que l'héritage ne concerne que les classes, et non les instances de ces classes. Lorsque nous disons qu'une classe dérivée hérite toutes les propriétés de sa classe parente, cela ne signifie pas que les propriétés des instances de la classe parente sont automatiquement transmises aux instances de la classe fille. En conséquence, retenez bien que :

> *Dans la méthode constructeur d'une classe dérivée, il faut presque toujours prévoir un appel à la méthode constructeur de sa classe parente.*

Résumé : définition et utilisation d'une classe

```
####################################
# Programme Python type            #
# auteur : G.Swinnen, Liège, 2009  #
# licence : GPL                    #
####################################

class Point(object):
    """point géométrique"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Rectangle(object):
    """rectangle"""
    def __init__(self, ang, lar, hau):
        self.ang = ang
        self.lar = lar
        self.hau = hau

    def trouveCentre(self):
        xc = self.ang.x + self.lar /2
        yc = self.ang.y + self.hau /2
        return Point(xc, yc)

class Carre(Rectangle):
    """carré = rectangle particulier"""
    def __init__(self, coin, cote):
        Rectangle.__init__(self,
                coin, cote, cote)
        self.cote = cote

    def surface(self):
        return self.cote**2

############################
## Programme principal : ##

# coord. de 2 coins sup. gauches :
csgR = Point(40,30)
csgC = Point(10,25)

# "boîtes" rectangulaire et carrée :
boiteR = Rectangle(csgR, 100, 50)
boiteC = Carre(csgC, 40)

# Coordonnées du centre pour chacune :
cR = boiteR.trouveCentre()
cC = boiteC.trouveCentre()

print("centre du rect. :", cR.x, cR.y)
print("centre du carré :", cC.x, cC.y)

print("surf. du carré :", end=' ')
print(boiteC.surface())
```

La classe est un moule servant à produire des objets. Chacun d'eux sera une instance de la classe considérée.

Les instances de la classe Point() seront des objets très simples qui possèderont seulement un attribut 'x' et un attribut 'y' ; ils ne seront dotés d'aucune méthode.

Le paramètre SELF désigne toutes les instances qui seront produites à partir de cette classe.

Les instances de la classe Rectangle() posséderont trois attributs. Le premier ('ang') doit être lui-même un objet de classe Point(). Il servira à mémoriser les coordonnées de l'angle supérieur gauche du rectangle. Les deux autres contiendront sa largeur et sa hauteur.

La classe Rectangle() comporte une méthode, qui renverra un objet de classe Point() au programme appelant.

Carre() est une classe dérivée, qui hérite les attributs et méthodes de la classe Rectangle(). Son constructeur doit faire appel au constructeur de la classe parente, en lui transmettant la référence de l'instance en cours de création (self) comme premier argument.

La classe Carre() comporte une méthode de plus que sa classe parente.

Pour créer (ou instancier) un objet, il suffit d'appeler une classe comme on appelle une fonction. La valeur renvoyée est une nouvelle instance de cette classe. Les instructions ci-contre créent donc deux objets de la classe Point() ...

... et celles-ci, encore deux autres objets.

Note : par convention, on donne aux classes des noms commençant par une majuscule.

La méthode trouveCentre() fonctionne pour les objets des deux types, puisque la classe Carre() a hérité de la classe Rectangle().

La méthode surface(), par contre, ne peut être invoquée que pour les objets Carre().

Modules contenant des bibliothèques de classes

Vous connaissez déjà depuis longtemps l'utilité des modules Python (cf. pages 50 et 71). Vous savez qu'ils servent à regrouper des bibliothèques de classes et de fonctions. À titre d'exercice de révision, vous allez créer vous-même un nouveau module de classes, en encodant les lignes d'instruction ci-dessous dans un fichier-module que vous nommerez **formes.py** :

```
class Rectangle(object):
    "Classe de rectangles"
    def __init__(self, longueur =0, largeur =0):
        self.L = longueur
        self.l = largeur
        self.nom ="rectangle"

    def perimetre(self):
        return "(%d + %d) * 2 = %d" % (self.L, self.l,
                                       (self.L + self.l)*2)
    def surface(self):
        return "%d * %d = %d" % (self.L, self.l, self.L*self.l)

    def mesures(self):
        print("Un %s de %d sur %d" % (self.nom, self.L, self.l))
        print("a une surface de %s" % (self.surface(),))
        print("et un périmètre de %s\n" % (self.perimetre(),))

class Carre(Rectangle):
    "Classe de carrés"
    def __init__(self, cote):
        Rectangle.__init__(self, cote, cote)
        self.nom ="carré"

if __name__ == "__main__":
    r1 = Rectangle(15, 30)
    r1.mesures()
    c1 = Carre(13)
    c1.mesures()
```

Une fois ce module enregistré, vous pouvez l'utiliser de deux manières : soit vous en lancez l'exécution comme celle d'un programme ordinaire, soit vous l'importez dans un script quelconque ou depuis la ligne de commande, pour en utiliser les classes. Exemple :

```
>>> import formes
>>> f1 = formes.Rectangle(27, 12)
>>> f1.mesures()
Un rectangle de 27 sur 12
a une surface de 27 * 12 = 324
et un périmètre de (27 + 12) * 2 = 78

>>> f2 = formes.Carre(13)
>>> f2.mesures()
Un carré de 13 sur 13
a une surface de 13 * 13 = 169
et un périmètre de (13 + 13) * 2 = 52
```

On voit dans ce script que la classe **Carre()** est construite par dérivation à partir de la classe **Rectangle()** dont elle hérite toutes les caractéristiques. En d'autres termes, la classe **Carre()** est une classe fille de la classe **Rectangle()**.

Vous pouvez remarquer encore une fois que le constructeur de la classe **Carre()** doit faire appel au constructeur de sa classe parente (`Rectangle.__init__(self, ...)`), en lui transmettant la référence de l'instance (**self**) comme premier argument.

Quant à l'instruction :

```
if __name__ == "__main__":
```

placée à la fin du module, elle sert à déterminer si le module est « lancé » en tant que programme autonome (auquel cas les instructions qui suivent doivent être exécutées), ou au contraire utilisé comme une bibliothèque de classes importée ailleurs. Dans ce cas cette partie du code est sans effet.

Exercices

12.3 Définissez une classe **Cercle()**. Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre **rayon**), vous définirez une méthode **surface()**, qui devra renvoyer la surface du cercle.

Définissez ensuite une classe **Cylindre()** dérivée de la précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres **rayon** et **hauteur**. Vous y ajouterez une méthode **volume()** qui devra renvoyer le volume du cylindre (rappel : volume d'un cylindre = surface de section × hauteur).

Exemple d'utilisation de cette classe :

```
>>> cyl = Cylindre(5, 7)
>>> print(cyl.surface())
78.54
>>> print(cyl.volume())
549.78
```

12.4 Complétez l'exercice précédent en lui ajoutant encore une classe **Cone()**, qui devra dériver cette fois de la classe **Cylindre()**, et dont le constructeur comportera lui aussi les deux paramètres **rayon** et **hauteur**. Cette nouvelle classe possédera sa propre méthode **volume()**, laquelle devra renvoyer le volume du cône (rappel : volume d'un cône = volume du cylindre correspondant divisé par 3).

Exemple d'utilisation de cette classe :

```
>>> co = Cone(5,7)
>>> print(co.volume())
183.26
```

12.5 Définissez une classe **JeuDeCartes()** permettant d'instancier des objets dont le comportement soit similaire à celui d'un vrai jeu de cartes. La classe devra comporter au moins les quatre méthodes suivantes :

- méthode constructeur : création et remplissage d'une liste de 52 éléments, qui sont eux-mêmes des tuples de 2 entiers. Cette liste de tuples contiendra les caractéristiques de chacune des 52 cartes. Pour chacune d'elles, il faut en effet mémoriser sépa-

rément un entier indiquant la valeur (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, les 4 dernières valeurs étant celles des valet, dame, roi et as), et un autre entier indiquant la couleur de la carte (c'est-à-dire 3,2,1,0 pour Cœur, Carreau, Trèfle et Pique).
Dans une telle liste, l'élément (11,2) désigne donc le valet de Trèfle, et la liste terminée doit être du type :

```
[(2, 0), (3,0), (3,0), (4,0), ..... (12,3), (13,3), (14,3)]
```

- méthode **nom_carte()** : cette méthode doit renvoyer, sous la forme d'une chaîne, l'identité d'une carte quelconque dont on lui a fourni le tuple descripteur en argument. Par exemple, l'instruction : `print(jeu.nom_carte((14, 3)))` doit provoquer l'affichage de : `As de pique`
- méthode **battre()** : comme chacun sait, battre les cartes consiste à les mélanger.
  Cette méthode sert donc à mélanger les éléments de la liste contenant les cartes, quel qu'en soit le nombre.
- méthode **tirer()** : lorsque cette méthode est invoquée, une carte est retirée du jeu. Le tuple contenant sa valeur et sa couleur est renvoyé au programme appelant. On retire toujours la première carte de la liste. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut alors renvoyer l'objet spécial **None** au programme appelant. Exemple d'utilisation de la classe **JeuDeCartes()** :

```
jeu = JeuDeCartes()                  # instanciation d'un objet
jeu.battre()                         # mélange des cartes
for n in range(53):                  # tirage des 52 cartes :
   c = jeu.tirer()
   if c == None:                     # il ne reste plus aucune carte
      print('Terminé !')             # dans la liste
   else:
      print(jeu.nom_carte(c))        # valeur et couleur de la carte
```

12.6 Complément de l'exercice précédent : définir deux joueurs A et B. Instancier deux jeux de cartes (un pour chaque joueur) et les mélanger. Ensuite, à l'aide d'une boucle, tirer 52 fois une carte de chacun des deux jeux et comparer leurs valeurs. Si c'est la première des deux qui a la valeur la plus élevée, on ajoute un point au joueur A. Si la situation contraire se présente, on ajoute un point au joueur B. Si les deux valeurs sont égales, on passe au tirage suivant. Au terme de la boucle, comparer les comptes de A et B pour déterminer le gagnant.