7 Programmation orientée objet

1 Concepts de base

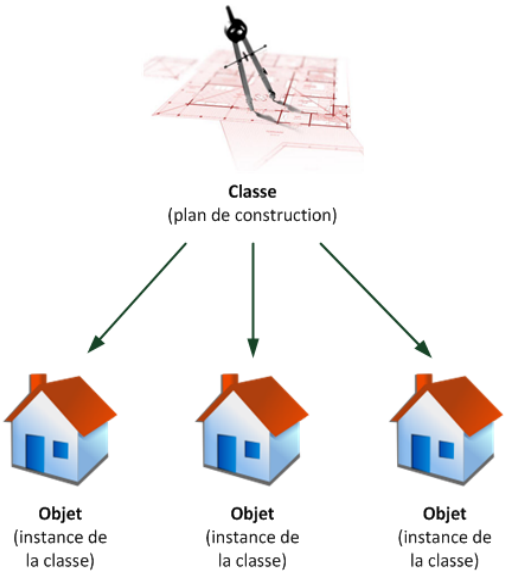
La programmation orientée objet (*Object Oriented Programming* ou *OOP)* permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent entre eux et avec le monde extérieur. L'idée de base de la programmation orientée objet consiste à regrouper dans un même ensemble (***l'objet***), à la fois un certain nombre de données (***les attributs d'instance***) et les algorithmes destinés à effectuer divers traitements sur ces données (***les méthodes***). Ainsi, un objet peut se résumer par l'égalité suivante :

$$\textbf{Objet} = [\textbf{attributs} + \textbf{méthodes}]$$

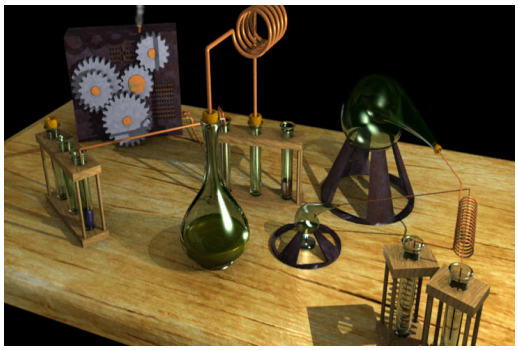
Cette façon d'associer dans une même "capsule" les propriétés d'un objet et les fonctions qui permettent d'agir sur elles, correspond chez les concepteurs de programmes à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel. Considérons 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 du 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 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 lorsqu'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.

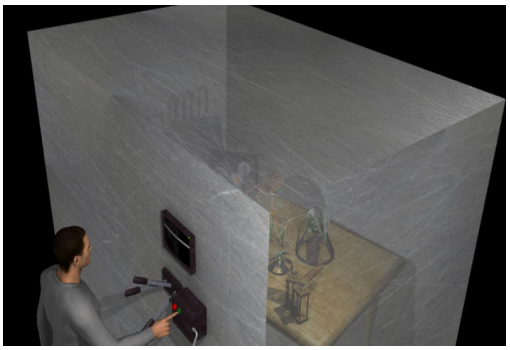
Les ***classes*** sont les principaux outils de la programmation orienté objet et consistent en la définition des caractéristiques et des comportements propres à tous les objets d'une même famille. Ainsi, une classe est avant tout une ***structure de données*** dans la mesure où il s'agit d'une entité chargée de gérer, classer et stocker des données sous une certaine forme. La principale différence entre une classe et une quelconque structure de données réside dans le fait qu'une classe regroupe des données, sous la forme d'***attributs***, mais également le moyen de les traiter, sous la forme de ***méthodes***.

En résumé, une classe peut être identifiée à un moule utilisé pour créer des objets d'une même famille. Ce moule se compose d'attributs, des données représentant l'état de l'objet, et de méthodes, des opérations applicables aux objets. Un objet, appelés également ***instance de classe***, est dès lors la réalisation concrète d'une classe possédant des caractéristiques qui leur sont propres. Par exemple, une maison est une instance particulière d'un plan qui la définit :



Le premier bénéfice de la programmation orientée objet réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'**encapsulation** : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermées » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'**interface** de l'objet. Ainsi, de l'extérieur, un objet est perçu comme une boîte noire ayant certaines propriétés et un comportement spécifié. Cette boite noire établit une séparation entre la fonctionnalité d'un objet et la manière dont cette fonctionnalité est réellement implémentée dans l'objet :

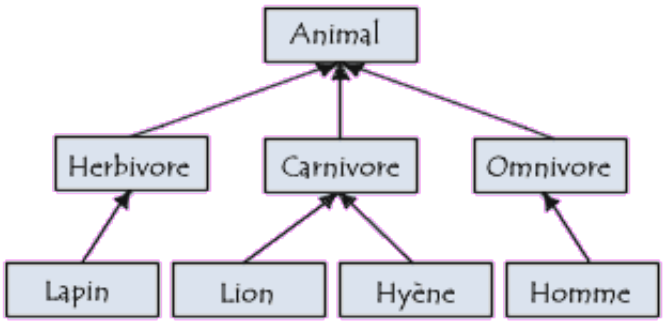




L'encapsulation consiste donc à garantir l'intégrité des données contenues dans l'objet en s'assurant que son utilisateur ne puisse pas modifier des attributs clés ou manipuler incorrectement l'objet sans passer par le contrôle de l'interface. L'usage d'un objet à travers son interface permet d'en simplifier la manipulation tout en évitant à son utilisateur de devoir savoir faire appel aux mécanismes fondamentaux régissant son fonctionnement.

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de **construire de nouveaux objets à partir d'objets préexistants**, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de **dérivation** et de **polymorphisme :**

- La **dérivation** ou **l'héritage** est le mécanisme qui permet de construire une classe « enfant » au départ d'une classe « parente ». L'enfant ainsi obtenu hérite de toutes les propriétés et de toutes les fonctionnalités de son ancêtre, tout en s'enrichissant de ses propres attributs et méthodes. Il est possible de représenter, sous la forme d'une *arborescence de classes*, la relation de parenté qui existe entre les différentes classes. Cette arborescence commence par une classe générale, appelée *superclasse*, puis les classes dérivées, appelées *classes filles* ou *sous-classes*, deviennent de plus en plus spécialisées :



- Le **polymorphisme** permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte. En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.

2 Mise en œuvre en Python

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction **class**. Nous allons apprendre à utiliser cette instruction, en définissant un type d'objets très rudimentaire, lequel sera simplement un nouveau type de données. Nous avons déjà utilisé des types de données jusqu'à présent, mais il s'agissait à chaque fois de types intégrés dans le langage lui-même. Nous allons maintenant créer un nouveau type composite : le type `Point`.

Ce type correspondra au concept géométrique de **point**. Dans un plan, on représente un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point $(25;17)$. Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type **float.** Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet. Pour y arriver, nous allons définir une **classe** `Point()` :

```
>>> class Point(object):
...     "Définition d'un point géométrique"
```

Les définitions de classes peuvent être situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer). L'exemple ci-dessus est probablement le plus simple qui puisse se concevoir. Une seule ligne nous a suffi pour définir le nouveau type d'objet `Point()`.

Remarquons d'emblée que :

- L'instruction `class` est un nouvel exemple d'**instruction composée**. N'oubliez pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne. Dans notre exemple ultra-simplifié, cette ligne n'est rien d'autre qu'un simple commentaire. Comme nous l'avons vu précédemment pour les fonctions, vous pouvez insérer une chaîne de caractères directement après l'instruction `class`, afin de mettre en place un commentaire qui sera automatiquement incorporé dans le dispositif de documentation interne de Python.
- Les parenthèses sont destinées à contenir la référence d'une classe préexistante. Cela est requis pour permettre le mécanisme d'**héritage**. Toute classe nouvelle que nous créons peut en effet hériter d'une **classe parente** un ensemble de caractéristiques, auxquelles elle ajoutera les siennes propres. Lorsque l'on désire créer une classe fondamentale – c'est-à-dire ne dérivant d'aucune autre, comme c'est le cas ici avec notre classe `Point()` – la référence à indiquer doit être par convention le nom spécial `object`, lequel désigne l'ancêtre de toutes les classes. On parle alors de *superclasse*.
- Une convention très répandue veut que l'on donne aux classes **des noms qui commencent par une majuscule**. Dans la suite de ce texte, nous respecterons cette convention, ainsi qu'une autre qui demande que dans les textes explicatifs, on associe à chaque nom de classe une paire de parenthèses, comme nous le faisons déjà pour les noms de fonctions.

Nous venons donc de définir une **classe** `Point()`. Nous pouvons à présent nous en servir pour **créer des objets de cette classe**, que l'on appellera aussi des **instances** de cette classe. L'opération s'appelle pour cette raison une **instanciation**. Créons par exemple un nouvel objet `p9` [1] :

```
>>> p9 = Point()
```

Après cette instruction, la variable `p9` contient la référence d'un nouvel **objet** `Point()`. Nous pouvons dire également que `p9` est une nouvelle **instance** de la classe `Point()`.

Voyons maintenant si nous pouvons faire quelque chose avec notre nouvel objet `p9` :

```
>>> print(p9)
<__main__.Point object at 0xb76f132c>
```

Le message renvoyé par Python indique que `p9` est une instance de la classe `Point()`, laquelle est définie elle-même au niveau principal (`main`) du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale.

Comme nous l'avons expliqué pour les fonctions, les chaînes de documentation de divers objets Python sont associées à l'attribut prédéfini **__doc__**. Il est donc toujours possible de retrouver la documentation associée à un objet Python quelconque, en invoquant cet attribut.

```
>>> print(p9.__doc__)
Définition d'un point géométrique
```

1. Sous Python, on peut donc instancier un objet à l'aide d'une simple instruction d'affectation. D'autres langages imposent l'emploi d'une instruction spéciale, souvent appelée `new` pour bien montrer que l'on crée un nouvel objet à partir d'un moule. Exemple : `p9 = new Point()`.

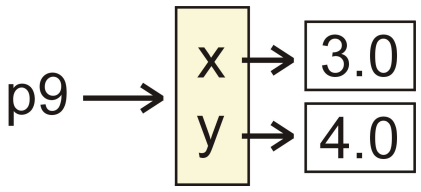
2.1 Attributs (ou variables) d'instances

L'objet que nous venons de créer est juste une coquille vide. Nous allons à présent lui ajouter des composants, par simple assignation, en utilisant le système de qualification des noms par points :

```
>>> p9.x = 3.0
>>> p9.y = 4.0
```

Les variables `x` et `y` que nous avons ainsi définies en les liant d'emblée à `p9`, sont désormais des **attributs** de l'objet `p9`. On peut également les appeler des **variables d'instance**. Elles sont en effet incorporées, ou plutôt **encapsulées**, dans cette instance (ou objet).

Le diagramme d'état ci-contre montre le résultat de ces assignations : la variable `p9` contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs `x` et `y`. Ceux-ci contiennent les références des valeurs 3.0 et 4.0, mémorisées ailleurs.



On pourra utiliser les attributs d'un objet dans n'importe quelle expression, exactement comme toutes les variables ordinaires :

```
>>> print(p9.x)
3.0
>>> print(p9.x**2 + p9.y**2)
25.0
```

Du fait de leur **encapsulation** dans l'objet, les attributs sont des variables distinctes d'autres variables qui pourraient porter le même nom. Par exemple, l'instruction `x = p9.x` signifie : « extraire de l'objet référencé par `p9` la valeur de son attribut `x`, et assigner cette valeur à la variable `x` ». Il n'y a pas de conflit entre la variable indépendante `x` et l'attribut `x` de l'objet `p9`. L'objet `p9` contient en effet son propre espace de noms, indépendant de l'espace de nom principal où se trouve la variable `x`.

Nous venons de voir qu'il est très aisé d'ajouter un attribut à un objet en utilisant une simple instruction d'assignation telle que `p9.x = 3.0` On peut se permettre cela sous Python (c'est une conséquence de son caractère foncièrement dynamique), mais cela n'est pas vraiment recommandable, comme vous le comprendrez plus loin. Nous n'utiliserons donc cette façon de faire que de manière anecdotique, et uniquement dans le but de simplifier nos premières explications concernant les attributs d'instances. La bonne manière de procéder sera développée bientôt.

2.2 Méthodes

Les fonctions peuvent utiliser des objets comme paramètres, et elles peuvent également fournir un objet comme valeur de retour. Par exemple, vous pouvez définir une fonction telle que celle-ci :

```
>>> def affiche_point(p):
...     print("({abscisse} ; {ordonnee})".format(abscisse=p.x, ordonnee=p.y))
```

Le paramètre `p` utilisé par cette fonction doit être un objet de type `Point()`, dont l'instruction qui suit utilisera les variables d'instance `p.x` et `p.y`. Lorsqu'on appelle cette fonction, il faut donc lui fournir un objet de type `Point()` comme argument. Essayons avec l'objet `p9` défini précédemment :

```
>>> affiche_point(p9)
(3.0 ; 4.0)
```

Si par la suite vous deviez fréquemment utiliser des objets de la classe `Point()`, cette fonction d'affichage vous serait probablement fort utile. Il serait donc judicieux d'arriver à **encapsuler** cette fonction `affiche_point()` dans la classe `Point()` 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 `Point()`.

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.

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 (habituellement, `self`), 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 même 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. Par conséquent, la définition d'une méthode comporte toujours au moins un paramètre, `self`, alors que la définition d'une fonction peut n'en comporter aucun.

Pour faire en sorte que la fonction `affiche_point()` devienne une méthode de la classe `Point()`, il nous suffit de déplacer sa définition à l'intérieur de celle de la classe :

```
>>> class Point(object):
...     "Nouvelle classe de points"
...     def affiche_point(self):
...         print("({abscisse} ; {ordonnee})".format(abscisse=self.x, ordonnee=self.y))
```

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

Nous disposons donc dès à présent d'une classe `Point()`, dotée d'une méthode `affiche_point()`. 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 :

```
>>> monPoint = Point()
```

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

```
>>> monPoint.affiche_point()
AttributeError: 'Point' object has no attribute 'x'
```

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

```
>>> monPoint.x = 1
>>> monPoint.y = 3
```

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

```
>>> monPoint.affiche_point()
(1;3)
```

Notons que cette invocation de méthode est une simplification de l'écriture suivante qui indique, de manière plus claire mais plus lourde, que la méthode `affiche_point` de la classe générale `Point()` est appliquée à l'objet particulier `monPoint` :

```
>>> Point.affiche_point(monPoint)
(1;3)
```

Nous avons précédemment 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.

2.3 La méthode constructeur

L'erreur que nous avons rencontrée précédemment ne se produirait pas, si nous nous étions arrangés pour que la méthode `affiche_point()` puisse toujours afficher quelque chose, sans qu'il ne soit nécessaire d'effectuer au préalable une manipulation sur l'objet nouvellement créé. Il serait donc 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 »). Par exemple :

```
>>> class Point(object):
...     "Encore une nouvelle classe de points"
...     def __init__(self):
...         self.x =0
...         self.y =0
...     def affiche_point(self):
...         print("({abscisse} ; {ordonnee})".format(abscisse=self.x, ordonnee=self.y))
```

Comme précédemment, créons un objet de cette classe et testons-en la méthode `affiche_point()` :

```
>>> unPoint = Point()
>>> unPoint.affiche_point()
(0;0)
```

Cette fois, nous n'obtenons plus d'erreur. En effet, lors de son instanciation, l'objet `unPoint` s'est vu attribuer automatiquement les deux attributs `x` et `y` par la méthode constructeur, avec zéro comme valeurs par défaut. Dès lors qu'un objet de cette classe existe, on peut donc tout de suite demander l'affichage de ses attributs.

Comme toute méthode, 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, coord_x =0, coord_y =0):
...         self.x =coord_x
...         self.y =coord_y
```

Notre nouvelle méthode `__init__()` comporte à présent 2 paramètres, avec pour chacun une valeur par défaut. Lorsque nous instancions un objet de cette nouvelle 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 qu'en invoquant une fonction quelconque.

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet `Point()` :

```
>>> A = Point(-5, 4)
>>> A.affiche_point()
(-5;4)
```

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

```
>>> B = Point(10)
>>> B.affiche_point()
(10;0)
```

Dès à présent, *nous nous fixons pour règle de ne plus modifier les attributs d'un objet par assignation directe depuis le monde extérieur*, comme nous l'avons fait précédemment. Nous veillerons au contraire à toujours utiliser pour cela des méthodes mises en place spécifiquement dans ce but. L'ensemble de ces méthodes constituera ce que nous appellerons désormais l'*interface* de l'objet.

2.4 Objets composés d'objets

Supposons maintenant que nous voulions définir une classe qui servira à représenter des **rectangles**. Pour simplifier, nous allons considérer que ces rectangles seront toujours orientés horizontalement ou verticalement, et jamais en oblique. Il existe plusieurs possibilités de définir de tels rectangles. Nous pourrions par exemple spécifier la position du centre du rectangle (deux coordonnées) et préciser sa taille (largeur et hauteur). Nous pourrions aussi spécifier les positions du coin supérieur gauche et du coin inférieur droit. Ou encore la position du coin supérieur gauche et la taille. Admettons que ce soit cette dernière convention qui soit retenue.

Définissons donc notre nouvelle classe :

```
>>> class Rectangle(object):
    """Définition d'une classe de rectangles"""
    def __init__(self, coin, largeur, hauteur):
        self.coin = coin
        self.largeur = largeur
        self.hauteur = hauteur
    def trouveCentre(self):
        xc = self.coin.x + self.largeur/2
        yc = self.coin.y + self.hauteur/2
        return Point(xc, yc)
```

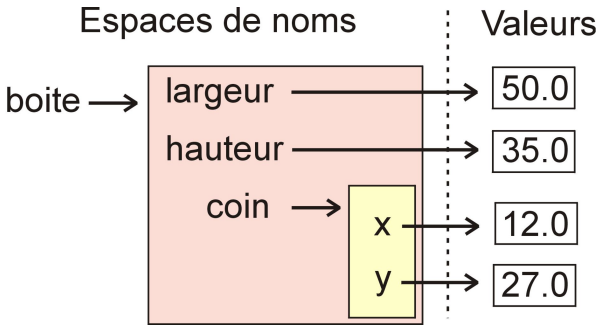
Selon la définition ci-dessus, les instances de la classe `Rectangle()` posséderont trois attributs : Le premier, `coin`, doit être lui-même un objet de la classe `Point()`. Il servira à mémoriser les coordonnées du coin supérieur gauche du rectangle. Les deux autres contiendront sa `largeur` et sa `hauteur`. Remarquons que la classe `Rectangle()` comporte une méthode, `trouveCentre`, qui renverra au programme appelant un objet de la classe `Point()`, lequel contiendra les coordonnées du centre du rectangle.

Pour accéder aux coordonnées du coin supérieur gauche, nous utilisons la *qualification des noms hiérarchisée* (à l'aide de points) que nous avons déjà rencontré à plusieurs reprises. Ainsi, l'expression `self.coin.x` signifie : "*Aller à l'objet référencé par la référence d'instance* `self`. *Dans cet objet, repérer l'attribut* `coin`, *puis aller à l'objet référencé dans cet attribut. Une fois cet autre objet trouvé, sélectionner son attribut* `x`".

Servons-nous à présent de cette classe pour en créer une instance :

```
>>> coin = Point(12, 27)
>>> boite = Rectangle(coin, 50.0, 35.0)
>>> centre = boite.trouveCentre()
>>> centre.affiche_point()
(37.0;44.5)
```

Nous créons tout d'abord une instance de la classe `Point()`. Ensuit, nous définissons un nouvel objet de la classe `Rectangle()` en spécifiant son coin supérieur gauche ainsi que sa largeur et sa hauteur. Ainsi nous avons placé un objet à l'intérieur d'un autre objet ! Vous pourrez peut-être mieux vous représenter la situation à l'aide d'un diagramme tel que celui-ci :



Le nom `boite` se trouve **dans l'espace de noms principal**. Il référence un autre **espace de noms** réservé à l'objet correspondant, dans lequel sont mémorisés les noms `largeur`, `hauteur` et `coin`. Ceux-ci référencent à leur tour, soit **d'autres espaces de noms** (cas du nom « coin »), soit **des valeurs** bien déterminées, lesquelles sont mémorisées ailleurs. Python réserve des espaces de noms différents pour chaque module, chaque classe, chaque instance, chaque fonction. Vous pouvez tirer parti de tous ces espaces de noms bien compartimentés afin de réaliser des **programmes robustes**, c'est-à-dire des programmes dont les différents composants ne peuvent pas facilement interférer.

3 Espaces de noms des classes et instances

Vous avez appris précédemment 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** ou attributs de classe. Ces variables sont partagées par *tous* les objets de la classe.
- Chaque instance d'une classe obtient son propre espace de noms. Les variables qui en font partie sont appelées **variables** ou **attributs d'instance**. Elles sont accessibles à *un seul* objet de la classe.
- 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 `Point()` définie précédemment. Nous avons instancié deux objets de cette classe : `A` et `B`. Chacun a été initialisé avec des valeurs différentes, indépendantes. Nous pouvons modifier et réafficher ces valeurs à volonté dans chacun de ces deux objets, sans que l'autre n'en soit affecté :

```
>>> A.x = 12
>>> B.affiche_point()
(10;0)
>>> A.affiche_point()
(12;4)
```

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

```
>>> class Espaces(object):                    # 1
...     aa = 33                               # 2
...     def __init__(self):                   # 3
            self.aa = 67                      # 4
...     def affiche(self):                    # 5
...         print(aa, Espaces.aa, self.aa)    # 6
...
>>> aa = 12                                   # 7
>>> essai = Espaces()                         # 8
>>> essai.affiche()                           # 9
12 33 67
>>> print(aa, Espaces.aa, essai.aa)           # 10
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 de l'instance (à la ligne 4) et enfin une dernière dans l'espace de noms principal (à la ligne 7).
La ligne 6 et la ligne 10 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.

4 Héritage

Les classes constituent le principal outil de la programmation orientée objet, 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** de 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 de plus en plus spécialisées. L'avantage de l'héritage réside dans le fait qu'il permet d'éviter de repartir à zéro lorsqu'on veut spécialiser une classe existante. De cette manière, il est possible d'acheter dans le commerce des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir.

Nous pouvons par exemple reprendre notre classe `Rectangle()`, qui contient un ensemble de caractéristiques propre aux rectangles : la position de son coin supérieur gauche, sa hauteur et sa largeur. A partir de cette classe *parente*, nous pouvons dériver une classe *fille* comme la classe `Carre()` qui héritera de toutes les caractéristiques de la classe `Rectangle()`, en y ajoutant ses spécificités :

```
class Point(object):
    """Définition d'un point mathématique"""
    def __init__(self, coord_x =0, coord_y=0):
        self.x =coord_x
        self.y =coord_y

    def affiche_point(self):
        print("({abscisse};{ordonnee})".format(abscisse=self.x, ordonnee=self.y))

class Rectangle(object):
    """Définition d'un rectangle horizontal"""
    def __init__(self, coin, largeur, hauteur):
        self.coin =coin
        self.largeur =largeur
        self.hauteur =hauteur

    def trouveCentre(self):
        xc = self.coin.x + self.largeur/2
        yc = self.coin.y + self.hauteur/2
        return Point(xc, yc)

class Carre(Rectangle):
    """Définition d'un carré horizontal"""
    def __init__(self, coin, cote):
        Rectangle.__init__(self, coin, cote, cote)
        self.cote = cote

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

Cet exemple nous montre 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. Notons que les classes les plus fondamentales dérivent quant à elles toutes de la *superclasse* `object`.

Dans l'exemple ci-dessus, on voit donc 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()`. Notons que le constructeur de la classe `Carre()` est une variante de celui de la classe `Rectangle()`. Elle doit donc vraisemblablement faire appel à celle-ci. Or, comment peut-on, à l'intérieur d'une classe, faire appel à une méthode définie dans une autre classe ? Il ne faut pas perdre de vue 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 `__init__` de la classe `Carre()` peut faire appel au constructeur `__init__` de la classe parente `Rectangle()` : les informations affichées seront bien celles de l'objet-carré courant, puisque sa référence a été transmise dans l'instruction d'appel :

```
        Rectangle.__init__(self, coin, cote, cote)
```

Cet appel est nécessaire, afin que les objets de la classe `Carre()` soient initialisés de la même manière que les objets de la classe `Rectangle()`. Si nous n'effectuons pas cet appel, les objets-carrés n'hériteront pas automatiquement des attributs `coin`, `largeur` et `hauteur`, car ceux-ci sont des *attributs d'instance* créés par la méthode constructeur de la classe `Rectangle()`, et celle-ci *n'est pas invoquée automatiquement* lorsqu'on instancie des objets d'une classe dérivée. Remarquons finalement que la classe `Carre()` possède un attribut d'instance `(self.cote)` ainsi qu'une méthode `(surface)` de plus que sa classe parente.

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 de 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.

```
>>> csgR = Point(40, 30)
>>> csgC = Point(10, 25)
>>> boiteR = Rectangle(csgR, 100, 50)
>>> boiteC = Carre(csgC, 40)
>>> cR = boiteR.trouveCentre()
>>> cC = boiteC.trouveCentre()
>>> cR.affiche_point()
(90.0;55.0)
>>> cC.affiche_point()
(30.0;45.0)
>>> print("Surface du carré :", boiteC.surface())
Surface du carré : 1600
>>> print("Surface du rectangle :", boiteR.surface())
...
AttributeError: 'Rectangle' object has no attribute 'surface'
```

Nous voyons ci-dessus que l'objet `boiteC`, qui est une instance de la classe `Carre()`, hérite non seulement de la méthode `surface` définie pour cette classe, mais également de la méthode `trouveCentre` définie pour la classe parente. Remarquons finalement que l'objet `boiteR`, qui est une instance de la classe `Rectangle()`, hérite uniquement de la méthode définie dans cette classe mais en aucun cas de la méthode `surface` définie dans la classe hiérarchiquement inférieure.

5 Polymorphisme

Lorsqu'une même méthode est à même de traiter des informations ou des objets de nature différente, on parle de **polymorphisme**. Cette caractéristiques est un des concepts essentiels de la programmation orientée objet. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est relatif aux méthodes des objets. En Python, on distingue essentiellement deux types de polymorphisme :

- Le **polymorphisme ad hoc**, appelé également **surcharge d'opérateur**, permet de définir des opérateurs dont l'utilisation sera différente selon le type des paramètres qui leur sont passés. Il est donc possible par exemple de surcharger l'opérateur "+" et de lui faire réaliser des actions différentes selon qu'il s'agit d'une opération entre deux entiers (addition : 3+4 = 7) ou entre deux chaînes de caractères (concaténation : "parle" + "ment" = "parlement"). Ainsi, l'opérateur reste le même mais son action est différente. En anglais, on parle alors d'**overloading**. La surcharge d'opérateur en Python se fait de la même manière que la définition d'une méthode à l'exception que le nom de la méthode à définir est spécifié par le langage. Ainsi, pour surcharger l'opérateur "+" de telle sorte qu'il puisse également additionner deux objets de type `Point()`, il suffit de définir une méthode `__add__` dans la classe `Point()` :

```
class Point(object):
    "Encore une nouvelle classe de points"
    def __init__(self, coord_x, coord_y):
        self.x = coord_x
        self.y = coord_y

    def affiche_point(self):
        print("({abscisse};{ordonnee})".format(abscisse=self.x, ordonnee=self.y))

    def __add__(self, autre):
        x_res = self.x + autre.x
        y_res = self.y + autre.y
        res = Point(x_res, y_res)
        return res
```

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

```
>>> A = Point(4, 7)
>>> B = Point(-2, 5)
>>> C = A + B
>>> C.affiche_point()
(2;12)
```

Notons que l'addition `A + B` est automatiquement convertie par Python sous la forme d'un appel à la méthode `__add__`, c'est-à-dire sous la forme `A.__add__(B)`. La surcharge d'opérateurs facilite la lisibilité du code tout en étendant les fonctionnalité de certains opérateurs. La plupart des opérateurs Python peuvent être surchargés de cette manière en définissant la méthode correspondante. Parmi les plus importants, nous retiendrons surtout les opérateurs suivants :

Opérateur	Méthode spéciale
+	__add__
-	__sub__
*	__mul__
/	__div__
<, <=, >, >=, ==	__lt__, __le__, __gt__, __ge__, __eq__
print	__str__

- Le **polymorphisme d'héritage**, également appelé **redéfinition**, permet de redéfinir une méthode dans des classes héritant d'une classe de base en la spécialisant. Il est alors possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque. Ceci permet de faire abstraction des détails des classes spécialisées d'une famille d'objets, en les masquant par une interface commune qui est la classe de base.

  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 ci-dessous génère des objets `Atome()` et des objets `Ion()`. Un ion étant simplement un atome modifié, 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()`) redéfinit 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 d'héritage**. On pourra dire également que la méthode `affiche()` de la classe `Atome()` a été **redéfinie**.

```
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)] # attribut de classe

    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("Nom de l'élément :", Atome.table[self.np][0])
        print("{n_prot} protons, {n_electr} électrons, {n_neutr} neutrons"\
        .format(n_prot = self.np, n_electr = self.ne, n_neutr = 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)
```

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 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. Une variable définie à cet endroit fait partie de l'espace de noms de la classe.

Pour chacun des 10 premiers éléments du tableau périodique, 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` Python nous semble cependant particulièrement explicite.

Viennent ensuite les définitions de deux méthodes :

- Le constructeur `__init__()` sert ici à générer trois **attributs d'instance**, destinés à mémoriser 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 font donc appel à celles-ci en leur transmettant la référence de l'instance à laquelle elles doivent s'associer. C'est ainsi que la méthode `affiche()` de la classe `Ion()` fait 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)
```

Testons maintenant ces deux classes :

```
>>> a1 = Atome(5)
>>> a2 = Ion(3, 1)
>>> a3 = Ion(8,-2)
>>> a1.affiche()
Nom de l'élément : bore
5 protons, 5 électrons, 6 neutrons
>>> a2.affiche()
Nom de l'élément : lithium
3 protons, 2 électrons, 4 neutrons
Particule électrisée. Charge = 1
>>> a3.affiche()
Nom de l'élément : oxygène
8 protons, 10 électrons, 8 neutrons
Particule électrisée. Charge = -2
```

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 `Ions()`, 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 : il s'agit de exemple de *polymorphisme d'héritage*.

6 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

attribut d'instance : caractéristique de chaque instance d'une classe qui n'existe, par rapport à une allocation mémoire, que dans une instance de cette classe. Avant qu'une instanciation de la classe ne soit effectuée, l'attribut n'a aucune existence physique. Si plusieurs instances de la même classe coexistent, il y a un exemplaire de chaque attribut d'instance de la classe par instance créée.

attribut de classe : caractéristique propre à chaque instance d'une classe qui existe, par rapport à une allocation mémoire, dès que la classe est chargée en mémoire. Quel que soit le nombre d'instanciations de la classe (0, 1 ou plus), un attribut de classe existe en un et un seul exemplaire. Un tel attribut joue un peu le rôle d'un variable globale d'un programme non-objet : sa classe lui sert de "boîte de rangement".

classe fille (sous-classe) : nouvelle classe créée par dérivation d'une classe existante.

classe parente : classe de laquelle une classe fille hérite ses attributs et méthodes.

classe : type de données abstrait défini par l'utilisateur et servant de moule à une famille d'objets ayant une structure et un comportement commun. Une classe décrit le nom et la nature des attributs que possèdera chaque objet ainsi que les méthodes qui s'appliqueront aux objets.

constructeur : méthode portant le nom `__init__` renvoyant implicitement la référence de l'instance construite. Un constructeur sert en particulier à initialiser les attributs d'instance. Lorsqu'une classe ne contient pas de constructeur, l'interpréteur en ajoute automatiquement un qui ne comporte qu'un appel au constructeur sans paramètre de la classe parente.

dérivation : synonyme d'héritage.

encapsulation : mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès direct aux données d'un objet mais en ne proposant que des méthodes de manipulation de cet objet. Ces méthodes forment l'*interface* de l'objet et permettent de maintenir l'ensemble des données de la classe dans un état cohérent. L'objet est ainsi vu par son utilisateur comme une boîte noire ayant certaines propriétés et un comportement spécifié. La manière dont ces propriétés ont été implémentées est alors cachées aux utilisateurs de la classe.

espace de noms : un espace de noms est semblable à un dictionnaire dans lequel les clés sont les noms des variables et les valeurs du dictionnaire sont les valeurs des variables. A n'importe quel point d'un programme Python, il y a plusieurs espaces de noms disponibles. Chaque fonction a son propre espace de noms, appelé *espace de noms local*, qui suit les variables de la fonction, y compris ses arguments et les variables définies localement. Chaque module a son propre espace de nom, appelé *espace de noms global*, qui suit les variables du module, y compris les fonctions, les classes, les modules importés et les variables globales du module.

héritage : principe propre à la programmation orientée objet permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'*héritage* (parfois appelé *dérivation*) provient du fait que la classe dérivée contient les attributs et les méthodes de sa classe parente. L'intérêt majeur est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.

instance : synonyme d'objet.

instanciation : action d'instancier, de créer un objet à partir d'une classe, c'est-à-dire d'un modèle.

interface : définition d'un contrat qu'une classe s'engage à respecter via une image abstraite qu'elle fournit d'elle-même à l'extérieur. On y spécifie la fonctionnalité de la méthode à l'aide de commentaires.

méthode : fonctionnalité observable chez chacun des objets d'une classe et définie sous la forme d'une fonction encapsulée dans une classe d'objets. L'invocation d'une méthode se fait à l'aide de la notation pointée et est applicable uniquement aux objets de la classe.

objet : une classe A est un modèle décrivant un type d'objets. Une instance de la classe A est un objet construit selon le modèle fourni par la classe A. Il s'agit d'une réalisation concrète de la classe A possédant des caractéristiques qui leur sont propres.

paradigme de programmation : façon de programmer, modèle qui oriente notre manière de penser pour formuler et résoudre un problème. Un paradigme fournit la vue qu'a le développeur de l'exécution de son programme. Parmi les paradigmes les plus connus, nous retiendrons la programmation impérative, la programmation fonctionnelle et la programmation orientée objet.

polymorphisme ad hoc : possibilité d'avoir des méthodes de même nom, avec des fonctionnalités similaires, dans des classes sans aucun rapport entre elles. Le polymorphisme ad hoc permet ainsi de redéfinir des opérateurs dont l'utilisation sera différente selon le type des paramètres qui leur sont passés.

polymorphisme d'héritage : spécialisation d'une méthode héritée, possibilité de redéfinir une méthode héritée d'une classe parente. Ce principe permet d'appeler la méthode d'un objet sans se soucier de son type.

programmation orientée objet : paradigme de programmation consistant en la définition et en l'interaction de briques logicielles appelées objets. Un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre.

référence d'instance : paramètre obligatoire dans la définition d'une méthode et désignant l'instance à laquelle la méthode sera associée. Par convention, ce paramètre prend le nom `self`.

spécialisation (overriding) : possibilité de redéfinir une méthode dans des classes héritant d'une classe de base.

structure de données : structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement. En informatique, il existe différents types de structure de données : les structures finies (constantes, variables,...), indexées (séquences, tableaux associatifs,...), récursives (arbres, graphes,...).

superclasse : classe de base située au sommet de la hiérarchie des classes. En Python, elle est appelée la classe `Object`.

surcharge (overloading) : possibilité d'avoir des méthodes de même nom, avec des fonctionnalités similaires, dans des classes sans aucun rapport entre elles.

7 Exercices

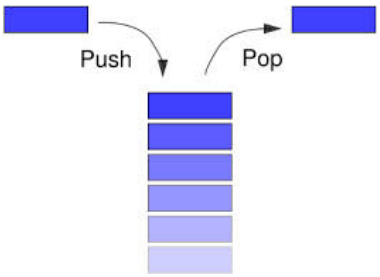
Exercice 7.1
Modifiez la classe `Point()` du cours de telle sorte qu'elle admette une méthode `distance(P)` prenant comme unique paramètre un point P et calculant la distance séparant l'instance considérée et le point P. Ainsi,

```
>>> A = Point(-3,4)
>>> B = Point(2,16)
>>> A.affiche_point()
(-3;4)
>>> B.affiche_point()
(2;16)
>>> print("Distance = ", A.distance(B))
Distance = 13.0
```

Exercice 7.2
Modifiez la classe `Rectangle()` du cours de telle sorte qu'elle admette une méthode `deepCopy()` qui retourne une véritable copie (et non un alias) de l'instance considérée. Ainsi,

```
>>> coin = Point(12, 27)
>>> boite = Rectangle(coin, 50, 35)
>>> copie = boite.deepCopy()
>>> if boite == copie:
        print("'copie' est un alias de 'boite'")
    else:
        print("'copie' est une véritable copie de 'boite'")
'copie' est une véritable copie de 'boite'
```

Exercice 7.3
Une **pile** (*stack* en anglais) est une structure de données fondée sur le principe "*dernier arrivé, premier sorti*" (ou *LIFO* pour *Last In, First Out*), ce qui signifie que le dernier élément ajouté à une pile est le premier à être récupéré. Le fonctionnement est donc celui d'une pile d'assiettes : on ajoute des assiettes sur la pile et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.



Définissez une classe `Stack()` permettant d'instancier des piles. Son constructeur initialisera l'attribut `items` qui contiendra, sous forme de liste initialement vide, les éléments de la pile. Cinq autres méthodes seront définies :

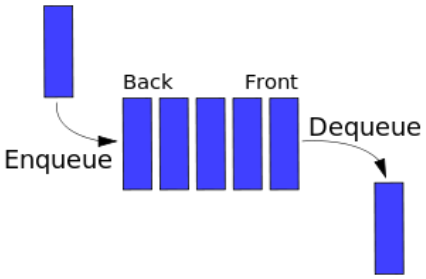
- `push(item) --> None` permettra d'ajouter l'élément `item` sur le haut de la pile ;
- `pop() --> Obj` enlèvera et retournera l'élément situé sur le haut de la pile ;
- `isEmpty() --> Bool` renverra `True` si la pile est vide et `False` sinon ;
- `getSize() --> Int` retournera le nombre d'éléments situés dans la pile.
- `printContent() --> None` affichera horizontalement le contenu de la pile du premier élément (haut de pile) au dernier (fond de pile).

Ainsi,

```
>>> pile = Stack()
>>> for k in range(1, 5):
        pile.push(k)
>>> pile.printContent()
4 3 2 1
>>> print(pile.getSize(), "élément(s)")
4 élément(s)
>>> for k in range(pile.getSize()):
        pile.pop()
>>> print(pile.getSize(), "élément(s)")
0 élément(s)
```

Le corps principal de votre programme utilisera la classe `Stack()` pour convertir un entier décimal en son code binaire correspondant à l'aide de l'algorithme présenté sur le site www.apprendre-en-ligne.net.

Exercice 7.4
Une **file** (*queue* en anglais) est une structure de données fondée sur le principe "*premier arrivé, premier sorti*" (ou *FIFO* pour *First In, First Out*), ce qui signifie que le premier élément ajouté à une file est le premier à être récupéré. Le fonctionnement est donc celui d'une file d'attente : les premières personnes à arriver sont les premières à sortir de la file.



Définissez une classe `Queue()`, permettant d'instancier des files. Son constructeur initialisera l'attribut `items` qui contiendra, sous forme de liste initialement vide, les éléments de la file. Cinq autres méthodes seront définies :

- `enqueue(item) --> None` permettra d'ajouter l'élément `item` à l'arrière de la file ;
- `dequeue() --> Obj` permettra de retirer l'élément situé à l'avant de la file ;
- `isEmpty() --> Bool` renverra `True` si la file est vide et `False` sinon ;
- `getSize() --> Int` retournera le nombre d'éléments situé dans la file.
- `printContent() --> None` affichera horizontalement le contenu de la file de gauche (dernier élément) à droite (premier élément).

Ainsi,

```
>>> file = Queue()
>>> for k in range(1, 5):
        file.enqueue(k)
>>> file.printContent()
4 3 2 1
>>> print(file.getSize(), "élément(s)")
4 élément(s)
>>> for k in range(file.getSize()):
        file.dequeue()
>>> print(file.getSize(), "élément(s)")
0 élément(s)
```

Le corps principal de votre programme utilisera la classe `Queue()` pour simuler le jeu de la patate chaude.

Exercice 7.5
Définissez une classe `Voiture()` permettant d'instancier des objets reproduisant le comportement de voitures. 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 sa vitesse, ni le nom de son conducteur. Les méthodes suivantes seront définies :

- `choix_conducteur(nom)` permettra de désigner (ou changer) le nom du conducteur.
- `accelerer(taux, duree)` permettra de faire varier la vitesse de la voiture. La variation de vitesse obtenue sera égale au produit : `taux` × `duree`. Si la voiture accélère au taux de $1.3\,m/s^2$ pendant 20 secondes, son gain de vitesse doit être égal à $26\,m/s$. Des taux négatifs seront acceptés (ce qui permettra de décélérer jusqu'à l'arrêt de la voiture). La variation de vitesse ne sera pas autorisée si le conducteur est '*personne*'.
- `affiche_tout()` permettra de faire apparaître les propriétés présentes de la voiture, c'est-à-dire sa marque, sa couleur, le nom de son conducteur, sa vitesse.

En outre, la classe `Voiture` stockera dans une variable de classe, nommée `count`, le nombre d'instances créées. Ce compteur sera incrémenté à chaque instanciation d'un nouvel objet de la classe `Voiture`.

Ainsi,

```
>>> print("Nombre de voitures existantes: ", Voiture.count)
Nombre de voitures existantes: 0
>>> a1 = Voiture('Peugeot', 'bleue')
>>> a2 = Voiture(couleur = 'verte')
>>> a3 = Voiture('Mercedes')
>>> print("Nombre de voitures existantes: ", Voiture.count)
```

```
Nombre de voitures existantes: 3
>>> a1.choix_conducteur('Roméo')
>>> a2.choix_conducteur('Juliette')
>>> a2.accelerer(1.8, 12)
>>> a3.accelerer(1.9, 11)
Cette voiture n'a pas de conducteur !
>>> a2.affiche_tout()
Ford verte pilotée par Juliette, vitesse = 21.6 m/s.
>>> a3.affiche_tout()
Mercedes rouge pilotée par personne, vitesse = 0 m/s.
```

Exercice 7.6
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, si possible, une certaine somme du solde ;
- `affiche()` permettra d'afficher le nom du titulaire et le solde de son compte.

Ainsi,

```
>>> 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.retrait(3000)
Le solde du compte de Dupont est insuffisant pour ce retrait.
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```

Exercice 7.7
Ecrivez un nouveau script qui récupère le code de l'exercice 7.6 en l'important comme un module. Définissez-y une nouvelle classe `CompteEpargne()`, dérivant de la classe `CompteBancaire()` importée, qui permette de créer des comptes d'épargne rapportant un certain intérêt au cours du temps. Pour simplifier, nous admettrons que ces intérêts sont calculés tous les mois.
Le constructeur de cette nouvelle classe devra initialiser un taux d'intérêt mensuel par défaut égal à 0.3%. Une méthode `changeTaux(valeur)` devra permettre de modifier ce taux à volonté tout en assurant que sa valeur reste positive. Une méthode `capitalisation(nombreMois)` devra :

- afficher le nombre de mois et le taux d'intérêt pris en compte ;
- calculer le solde atteint en capitalisant les intérêts composés, pour le taux et le nombre de mois qui auront été choisis.

Ainsi,

```
>>> compte = CompteEpargne('Duvivier', 600)
>>> compte.depot(350)
>>> compte.affiche()
Le solde du compte bancaire de Duvivier est de 950 euros.
>>> compte.capitalisation(12)
Capitalisation sur 12 mois au taux mensuel de 0.3 %
>>> compte.affiche()
Le solde du compte bancaire de Duvivier est de 984.769981274 euros.
>>> compte.changeTaux(.5)
>>> compte.capitalisation(12)
Capitalisation sur 12 mois au taux mensuel de 0.5 %
>>> compte.affiche()
Le solde du compte bancaire de Duvivier est de 1045.50843891 euros.
```

Exercice 7.8
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.

Ainsi,

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

Finalement, ajoutez 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.

Ainsi,

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

Exercice 7.9
Cet exercice a pour objectif de simuler un jeu de cartes :

(1) Définissez une classe `Carte()` permettant d'instancier des objets dont le comportement est similaire à celui d'une carte tirée d'un jeu de Jass. Cette classe sera composée de deux attributs de classe `Valeurs` et `Couleurs` permettant de faire la correspondance entre le numéro de la valeur et de la couleur d'une carte particulière et sa traduction sous forme de chaîne de caractères. Cette classe devra comporter au moins les trois méthodes suivantes :

- Le constructeur permet d'initialiser deux attributs d'instance sous forme d'entiers, le premier indiquant la valeur (un entier entre 6 et 14, les 4 dernières valeurs étant celle des valet, dame, roi et as) et un autre entier indiquant la couleur de la carte (c'est-à-dire 0, 1, 2, 3 pour Cœur, Carreau, Trèfle et Pique). Ainsi, le couple $(11; 2)$ correspond au "Valet de Trèfle".
- Les opérateurs de comparaison "<", ">" et "=" doivent être surchargés de telle sorte qu'il soit possible de comparer deux cartes entre elles à partir de leur valeur respective. Par exemple, la méthode `__lt__` surchargeant l'opérateur "<" retournera `True` si la valeur de la première carte est plus petite que celle de la deuxième et `False` sinon.
- La fonction `print` appliquée à une carte doit renvoyer, sous la forme d'une chaîne de caractères, l'identité de la carte. A ce dessein, la méthode surchargeant la fonction `print` utilisera les variables de classe `Valeurs` et `Couleurs`.

Ainsi,

```
>>> carte1 = Carte(11, 2)
>>> carte2 = Carte(11, 3)
>>> carte3 = Carte(7, 1)
>>> print(carte1)
Valet de trèfle
>>> print(carte2)
Valet de pique
>>> print(carte3)
7 de carreau
>>> if carte1 == carte2:
        print("Les cartes ont même valeur")
Les cartes ont même valeur
>>> if carte3 < carte1:
        print("La première carte est plus grande que la troisième")
La première carte est plus grande que la troisième
```

(2) Définissez une classe `Jeu()` permettant d'instancier des objets dont le comportement est similaire à celui d'un vrai jeu de cartes. Cette classe devra comporter au moins les trois méthodes suivantes :

- Méthode constructeur : création et remplissage d'une liste d'instance `cartes` de 36 éléments différents qui sont eux-mêmes des objets de la classe `Carte()`. Cette liste sera remplie à l'aide de deux boucles, l'une gérant la couleur et l'autre la valeur des cartes nouvellement créées et insérées.
- Méthode `battre()` : cette méthode servira à mélanger les cartes du jeu, c'est-à-dire les éléments de la liste `cartes` contenant les cartes, quel qu'en soit le nombre.
- Méthode `tirer()` : lorsque cette méthode est invoquée, une carte est retirée du jeu. L'objet de la classe `Carte()` est alors retourné au programme appelant. On retire toujours la dernière carte de la liste que l'on efface ensuite de la liste. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut renvoyer l'entier 0.

Ainsi,

```
monJeu = Jeu()                  # instanciation d'un objet
monJeu.battre()                 # mélange des cartes
c = monJeu.tirer()              # tirage des 36 cartes
while not c:
    print(c)
    c = monJeu.tirer()
```

(3) Définissez une classe `Main()` dérivée de la classe `Jeu()` et simulant une main de cartes que reçoit un joueur lors d'une partie. Cette classe héritera uniquement des méthodes de la classe `Jeu()` mais pas de ses attributs d'instance. Elle devra comporter au moins les trois méthodes suivantes :

- Méthode constructeur : initialisation d'une liste d'instance vide `cartes` permettant de récupérer les cartes distribuées.
- Méthode `ajouter(carte)` : cette méthode permettra d'ajouter une nouvelle carte à la liste d'instance `cartes`.
- Méthode `trier()` : cette méthode permettra de trier les cartes d'une main par valeur uniquement (d'abord, les 6, ensuite les 7, ..., finalement les As).

Ainsi,

```
monJeu = Jeu()                  # instanciation d'un jeu
monJeu.battre()                 # mélange des cartes
maMain = Main()                 # instanciation d'une main
for n in range(4):              # tirage de 4 cartes
    maMain.ajouter(monJeu.tirer())
maMain.trier()                  # trier les cartes par valeurs
for n in range(4):
    carte = maMain.tirer()
    print(carte)
```

Exercice 7.10
Utilisez les classes définies dans l'exercice précédent pour simuler une partie du jeu "La Bataille" : Instancier un jeu de carte, le brasser, distribuer deux mains de 18 cartes à 2 joueurs A et B. Ensuite, à l'aide d'une boucle, tirer 12 fois une carte de chacune des deux mains et comparer leurs valeurs. Un point est ajouté au joueur possédant la carte dont la valeur est la plus élevée. Si les deux valeurs sont égales, on passe au tirage suivant. Au terme de la boucle, comparer les comptes de A et de B pour déterminer le gagnant et afficher les 6 cartes restantes de chacun des deux joueurs dans l'ordre.