Chapitre 2

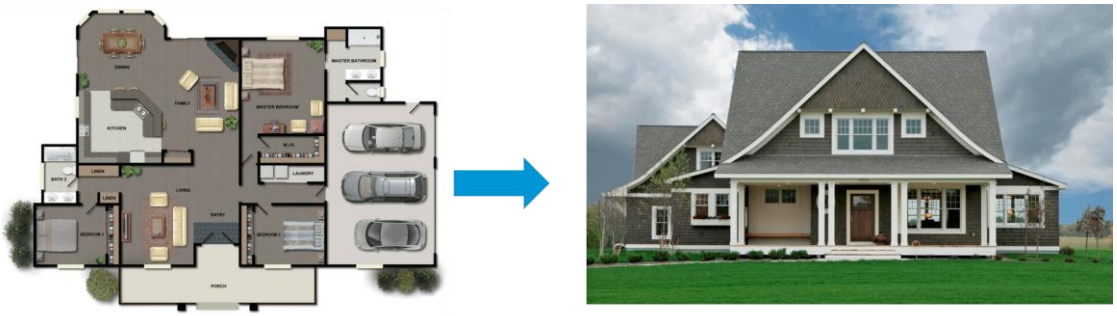
Programmation orientée Objet en Python

Notions de base : Classe, Constructeur, Objets, Attributs, Méthodes, Héritage, Méthodes spéciales

1. Introduction

Qu'est-ce qu'une classe Python?

Une classe en python est le modèle à partir duquel des objets spécifiques sont créés. Il vous permet de structurer votre logiciel de manière particulière. Les classes nous permettent de regrouper logiquement nos données et nos fonctions de manière qu'elles soient faciles à utiliser. Considérez l'image ci-dessous :



(A) (B)

La première image (A) représente un modèle de maison pouvant être considéré comme une **classe**. Avec le même plan, nous pouvons créer plusieurs maisons et celles-ci peuvent être considérées comme des **objets**. À l'aide d'une classe, vous pouvez ajouter de la cohérence à vos programmes afin qu'ils puissent être utilisés de manière plus propre et efficace.

Ecrire un programme orientée objet, consiste en effet à regrouper dans un même ensemble, à la fois un certain nombre de données (variables de classe et variables d'instance accessibles via la notation par points), et les algorithmes destinés à effectuer divers traitements sur ces données (fonctions accessibles via la notation par points).

2. Création d'une classe en Python :

Pour créer une nouvelle classe d'objets en Python, on utilise le mot clé **class** qui permet de déclarer un nouveau type en Python.

Syntaxe :

```
class Nomclasse :
        <instruction1> # au moins d'une ligne de code indentée : commentaire, pass, …
        …
        <instructionN>
```

Exemple:
Définition d'une classe d'objets nommée Point.

```
class Point : #  Par convention le nom de la classe commence par une majuscule.
        def __init__ (self, x, y):
                self.x = x
                self.y = y
```

La méthode __init__ :

- Abréviation de “initialization”
- Est une méthode spéciale (le nom de la fonction est entouré de deux traits de soulignement « _ »)
- Appelée *constructeur* de la classe (permet de créer des objets de type Point).
- Initialise les champs x et y (associées à self) au moment de la création d’objet avec les valeurs passées en paramètres
- **self** : désigne l’objet qui sera créé et initialisé au moment de l’appel de __init__ (self doit être placé toujours en première position).

3. Création d’un objet de type Point :

Pour créer un Point, vous appelez Point comme s'il s'agissait d'une fonction : **nom_objet = nomClasse( )**

- L’objet à créer est appelée **instance**
- L’opération de création d’objet est : **instanciation**
- La méthode qui assure la création d’objet est : le **constructeur** (la méthode __init__)

Exemple :

```
>>> p1 = Point(3, 5)
```

La méthode __init__ est appelée automatiquement d’une manière implicite (via le nom de la classe et non pas avec son propre nom __init__) . Le paramètre nommé **self** sera remplacé par **p1**, x par 3 et y par 5.
Autres exemples que l’on écrit couramment :

```
>>> #Appel au constructeur __init__ de la classe dict pour créer un dictionnaire vide
>>> d = dict()
>>> #Appel au constructeur __init__ de la classe list pour créer une liste vide
>>> l = list()
```

La variable p1 enregistre la référence (l’adresse) de l’objet Point créé.

```
>>> print(p1)
```

On dit que p1 est un objet, une instance de type Point, une référence vers un objet de type Point ou tout simplement p1 est un Point.

Pour connaitre la classe d’un objet ou savoir si un objet appartient à une classe, Python propose deux fonction : ***type*** et ***isinstance***.

```
>>> type(p1) # Point
>>> isinstance([1,2,3],list) #True
>>> isinstance (p1, Point) #True
```

Remarque :

```
>>> Point(3, 5) #instanciation : création d’un nouvel objet de type Point
>>> p1 = Point (3, 5) # sauvegarde de la référence du nouvel objet créé dans p1
>>> print(p1) # <__main__.Point object >
>>> p2 = p1 # p2 est un alias de p1 référence le même objet Point (at 0x023BFEB0).
>>> print (p1 == p2) #True

>>> p1 = Point (3, 5); print(p1) # <__main__.Point object at 0x023BFEB0>
>>> p2 = Point (3, 5); print(p2) #<__main__.point object at 0x023BFF30>
>>> print(p1 == p2) #False, Deux références différentes
>>> id(p1) ; id(p2) # Deux identifiants différents, donc deux objets différents
```

4. Attributs

En Python, on distingue deux types d'attributs : Les attributs d'instances et les attributs de classes.

4.1. Attributs d'instances

Les attributs d'instances sont des attributs sont :

- Déclarés à l'intérieur du constructeur __init__ et associés au mot clé self (**self.attribut**)
- Accessibles via le nom de l'instance (**objet.attribut**)
- Leurs valeurs ne sont pas partagés entre les objets de cette classe (Les valeurs des attributs d'instance peuvent différer d'un objet à l'autre d'une même classe).

Exemple :

```
class Point :
        def __init__ (self, x, y):
                self.x = x
                self.y = y
```

Les attributs d'un objet sont accessibles et modifiables après la construction de l'objet.

```
>>> p1 = Point(3, 5);
>>> p1.x # 3
>>> p1.x = 5 # Affecter à l'attribut x la valeur 5
```

Lorsque vous accédez à un attribut d'un objet qui n'existe pas, Python créera simplement un nouvel attribut associé à cet objet. Donc en Python, il est toujours possible après la définition d'une classe, d'ajouter des attributs pour une instance de cette classe.

```
>>> p1 = Point(3, 5);
>>> p2 = Point(1, 2);
>>> p1.z = 0 # Ajouter un attribut z initialisé à 0 à l'objet p1
>>> print (p1.z) ; # 0
>>> print (p2.z) ; # Erreur
```

4.2. Attributs de classes

Les attributs de classes sont :

- Déclarés dans la classe et en dehors du constructeur __init__.
- Accessibles via le nom de la classe (**nomClasse.attributClasse**) et aussi via les objets de cette classe (**objet.attributClasse**).
- Leurs valeurs sont partagés entre tous les objets de cette classe.

```
>>> class Etudiant :
        institut = "IPEIN" # attribut de classe : même valeur pour tous les étudiants
        def __init__(self, nom, prenom, niveau, groupe) :
                self.nom = nom          # attribut d'instance
                self.prenom = prenom    # attribut d'instance
                self.niveau = niveau    # attribut d'instance
                self.groupe = groupe    # attribut d'instance
>>> print (Etudiant.institut) #IPEIN
>>> e1 = Etudiant ("Suissi", "Ali", 1, "PT2")
>>> print(e1.institut, e1.prenom) # IPEIN Ali
>>> e2 = Etudiant ("SAIED", "Amel", 2, "SP2")
>>> print(e2.institut, e2.prenom) # IPEIN Amel
```

5. Les méthodes d'une classe

Une méthode est une fonction définie à l'intérieur d'une classe et qui utilise ou/et modifie les attributs de l'objet de cette classe qui lui fait appel. Elle comporte au moins un paramètre nommé **self** qui sera remplacé lors de l'appel de la fonction par l'objet lui-même (l'objet sur lequel on effectue le traitement).

Syntaxe :

```
class NomClasse :
        def nomMethode (self, paramètres) :
                …
```

Python offre la possibilité d'appeler une méthode de deux manières :

- **objet.methode( )** # la plus utilisée
- **NomClasse.methode(objet)**

Exemple :

```
>>> L = list() ; # [] : liste vide
>>> #la méthode append de la classe list modifie les données de l'objet L de type list
>>> L.append(2)
>>> list.append(L,2) # Autre format d'appel
```

Exemple :

```
class Point :

        def __init__ (self, x, y):
                self.x = x
                self.y = y

        def afficher (self) : #self : c'est le point à afficher
                print (" Point(x={},y={})".format(self.x,self.y))

>>> p  = Point(1,2) #Créer un Point p
#self est remplacé par l'objet p au moment de l'appel de la méthode afficher
>>> p.afficher()
>>> Point.afficher(p) #Autrement
```

Exemple :

```
class Valise:
    def __init__(self):
        self.elements = []
    def ajouter(self, x):
        self.elements.append(x)
>>> v = Valise()
>>> v.ajouter("livre")
>>> print(v.elements)
```

6. Les attributs et les méthodes privées

En Python chaque attribut ou méthode précédée par un double trait de soulignement (__) signifie que vous ne devriez pas accéder à cette méthode ou à cet attribut en dehors de la classe où il est défini.
Syntaxe :

self.__nomAttributPrive
def __nomMethodePrivee(self)

Exemple :

```
class CompteBancaire:
    def __init__(self, nom, solde):
        self.nom     = nom   #Attribut publique
        self.__solde = solde #Attribut privé

    def retrait(self, somme): #Méthode publique
        if operationValide(self.solde, somme):
            self.__solde = self.__solde - somme

    def __operationValide(solde, somme): #Méthode privée
        return somme <= solde
```

```
>>> cb = CompteBancaire("Ali",1000)
>>> cb.__solde
AttributeError: 'CompteBancaire' object has no attribute '__solde'
```

C'est une convention de codage permettant de cacher quelques membres de la classe aux développeurs.

NB : En Python tout est publique ! Lorsque vous utilisez un double trait de soulignement (__), vous pouvez toujours accéder aux variables privées.
Chaque membre privé est renommé par Python en lui attachant un trait de soulignement et le nom sa la classe.
Syntaxe : Si une Classe **X** possède un attribut privé **__a**, il sera renommé à **_X__a**.

Exemple :

```
>>> cb._CompteBancaire__solde
1000
```

7. Héritage en Python

L'héritage est l'un des aspects fondamentaux de la programmation orienté objet. Les classes que l'on définit peuvent hériter (utiliser sans redéclarer) des attributs et des méthodes d'autres classes. La classe dérivée (qui hérite) possède alors les attributs et méthodes de la classe dont elle dérive (héritée) et ses propres attributs et méthodes.

Par défaut, toutes les classes sont descendantes de la classe ***object*** et ainsi tous les objets que l'on crée en Python possèdent les attributs et méthodes de cette classe.

7.1. Définir une classe dérivée (classe fille/sous classe) à partir d'une classe de base (classe mère/super classe) :

Lorsqu'on veut que l'héritage provienne d'une classe précise, on doit le préciser dans la définition de la nouvelle classe. Ce procédé s'appelle la **dérivation de classe**.
Syntaxe:

```
Class NomClasseDerivee (NomClassedeBase):
        <instruction1>
        …
        <instructionN>
```

Lorsqu'une classe hérite d'une autre elle peut :
- Utiliser les attributs de la classe parente.
- Utiliser les méthodes de la classe parente.
- Définir de nouveaux attributs.
- Définir de nouvelles méthodes.
- Redéfinir certaines méthodes.

L'héritage dénote une relation de généralisation/spécialisation : toute relation d'héritage peut se traduire par la phrase « La classe dérivée est une version spécialisée de la classe de base ». Le principe de généralisation/spécialisation invoque donc une relation dite relation « est-un ».
Exemples :
- Un Mini-bus est un Bus → class Mini_bus (Bus) : …
- Un Bus est une Véhicule → class Bus (Véhicule) : …

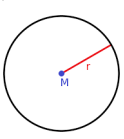
Parmi les avantages de l'héritage, on peut citer :
- Factorisation de comportements communs à une hiérarchie : donc le code sera de taille plus faible.
- Lors d'une dérivation, seul le code spécifique reste à écrire : donc le développement sera plus rapide.

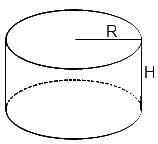
7.2. Constructeur de la classe fille :

Syntaxe:

```
class nomClasseDerivee (nomClasseBase):
  def  __init__ (self, paramClasseBase, paramClasseDerivee)
       nomClasseBase.__init__(self, paramClasseBase)
       self.attributClasseDerivee = paramClasseDerivee
```

Exemple :
- Un Point est défini par deux nombres réels (x et y).
- Un *Cercl*e est défini par un Point qui représente son centre et un rayon r (nombre réel)
- Un Cylindre est défini par le rayon du disque r et la hauteur h du cylindre.





Dans cet exemple, nous allons coder les classes comme suit :

1. On commence tout d'abord par définir la classe **Point** définie par les attributs : **x** et **y** de type réel. On ajoutera un constructeur permettant d'initialiser les valeurs de x et de y.

```
class Point : #  Par convention le nom de la classe commence par une majuscule.
     def __init__ (self, x, y):
            self.x = x
            self.y = y
```

2. Ensuite on implémente une classe **Cercle** définie par les attributs : **centre** de type Point et **rayon** de type réel. On définit un constructeur qui permet d'initialiser les attributs centre et rayon. On ajoutera une méthode une méthode *surface*, qui devra renvoyer la surface du cercle (Surface cercle= Pi * rayon **2).

```
from math import pi

class Cercle:
      def __init__(self, centre, rayon):
            self.centre = centre #de type Point
            self.rayon = rayon #de type réel
      def surface(self):
            return pi * self.rayon**2
```

3. Enfin la classe **Cylindre** définie par les attributs : centre de type Point, rayon et hauteur de types réel.
   Afin d'éviter la redéfinition des mêmes attributs (centre et rayon) dans deux classes différentes, la classe Cylindre devra hériter de la classe Cercle et on y ajoutera l'attribut : hauteur de type réel.
   Le constructeur de cette nouvelle classe comportera, en plus du paramètre self, trois autres paramètres : les deux premiers paramètres rayon et hauteur seront utilisés pour initialiser les attributs centre et rayon déclarés dans le constructeur de la classe Cercle et le troisième paramètre hauteur pour initialiser l'attribut hauteur déclaré dans le constructeur de la classe Cylindre.
   On ajoutera une méthode *volume* qui devra renvoyer le volume du cylindre (rappel : volume d'un cylindre = surface de section × hauteur).

```
class Cylindre(Cercle):
     def __init__(self, centre, rayon, hauteur):
            Cercle.__init__(self, centre, rayon)
            self.hauteur = hauteur #de type réel
     def volume(self):
            #surface() est héritée de la classe parente
            return self.surface()* self.hauteur
```

Vérifions :

```
>>> issubclass(Point ,object)
>>> issubclass(Cerlce ,object)
>>> issubclass(Cylindre,Cercle)
>>> dir(object)
>>> dir(Point)
>>> dir(Cercle)
>>> dir(Cylindre)
>>> p = Point(1,2)
>>> c = Cercle(P,5) ; c.surface()
>>> cyl = Cylindre(P,5,10) ;  cyl.volume()
```

7.3. Surcharge des méthodes spéciales :

Si l'on souhaite modifier une méthode de la classe parente en gardant le même nom pour la méthode de la classe descendante, on dit que l'on procède à une ***surcharge de méthode***. La méthode de la classe mère reste inchangée.

La surcharge de méthode est en particulier très utile pour les **méthodes spéciales** de Python (appelées «méthodes magiques»). En effet, les classes Python possèdent des méthodes spéciales dont les identificateurs commencent et se terminent par double trait de soulignement telles que __len__, __init__.

Certaines sont associées aux opérateurs. Par exemple l'opérateur == est défini par la méthode **__eq__** . On peut et on doit parfois surcharger cet opérateur.

Exemple :

```
>>> p1 = point(1,2)
>>> p2 = point(1,2)
>>> p1 == p2
```

Surcharger la méthode __eq__ de la manière suivante :

```
def __eq__(self , other):
        return (self.x == other.x) and (self.y == other.y)
```

Essayez :

```
>>> p1 == p2
```

Les autres principales méthodes spéciales que l'on surcharge assez souvent :

- __str__ : qui contrôle la conversion en chaîne de caractères, utilisée de façon transparente par la fonction print.
- __repr__ : qui contrôle l'affichage de l'objet dans le Shell.
- Les opérateurs de comparaison : __ne__ pour != , __le__ pour <=, __lt__ pour <, __ge__ pour >=, __gt__ pour >.
- Les opérations : __add__ pour + , __sub__ pour -, __mul__ pour * , __div__ pour /, __floordiv__ pour //.

Exemple :

```
def __str__(self):
        return "Point (x={}, y={})".format(self.x, self.y)

>>> p =Point(2,3)
>>> print(p)
Point (x=2, y=3)
```

Exemple :

```
def __repr__(self):
        return "Point (x={}, y={})".format(self.x, self.y)

>>> p =Point(2,3)
>>> p
Point (x=2, y=3)
```