



POO en Python

Khalifa SYLLA
khalifasylla@gmail.com



Les Bases du langage PYTHON

Classe

Une classe est la définition d' un concept métier, elle contient des attributs (des valeurs) et des méthodes (des fonctions). En Python, le nom d'une classe ne peut commencer par un chiffre ou un symbole de ponctuation, et ne peut pas être un mot clé du langage comme **while** ou **if** .

À part ces contraintes, Python est très permissif sur le nom des classes et variables en autorisant même les caractères accentués. Cette pratique est cependant extrêmement déconseillée à cause des problèmes de compatibilité entre différents systèmes.

Les Bases du langage PYTHON

Classe

Syntaxe de l'implémentation d'une classe en Python ne possédant aucun membre : ni attribut, ni méthode.

```
class MaClass:
    # Pour l'instant, la classe est déclarée vide,
    # d'où l'utilisation du mot-clé 'pass'.
    pass
```

Le mot clé **class** est précédé du nom de la classe. Le corps de la classe est indenté comme le serait le corps d'une fonction. Dans un corps de classe, il est possible de définir :

- des fonctions (qui deviendront des méthodes de la classe) ;
- des variables (qui deviendront des attributs de la classe) ;
- des classes imbriquées, internes à la classe principale.

Les Bases du langage PYTHON

Classe

Syntaxe de l'implémentation d'une classe en Python ne possédant aucun membre : ni attribut, ni méthode.

```
class nomDeLaClasse:
```

```
    """documentation de la classe"""
```

Écrire une documentation **pertinente** fait partie des bonnes pratiques. On peut accéder à la documentation d'une classe par la commande : `nomDeLaClasse.__doc__`

Cette commande retourne une chaîne de caractère avec laquelle on pourra faire par exemple une affectation ou un affichage.

Les Bases du langage PYTHON

La déclaration d'une classe imbriquée dans une autre ressemble :

```
# Classe contenante, déclarée normalement.  
class Humain :  
  
    # Classes contenues, déclarées normalement aussi,  
    # mais dans le corps de la classe contenante.  
  
    class Femme :  
        pass  
  
    class Homme:  
        pass
```

La seule implication de l'imbrication est qu'il faut désormais passer par la classe Humain pour utiliser les classes Femme et Homme en utilisant l'opérateur d'accès « point » : Humain.Femme et Humain.Homme .

Exactement comme on le ferait pour un membre classique. L'imbrication n'impacte en rien le comportement du contenant ou du contenu.

Les Bases du langage PYTHON

Instance

- Si on exécute le code précédent de déclaration de classe vide, rien ne s'affiche. Ceci est
Après cette exécution, l'environnement Python sait qu'il existe désormais une classe nommée **MaClasse**. Maintenant, la classe peut être utilisée.
- L'instanciation, c'est le mécanisme qui permet de créer un objet à partir d'une classe. Une classe peut générer de multiples instances, mais une instance ne peut avoir comme origine qu'une seule classe. Ainsi donc, si l'on veut créer une instance de **MaClasse** :

Instance=Maclasse()

Les Bases du langage PYTHON

- Les parenthèses sont importantes , elles indiquent un appel de méthode. Dans le cas précis d'une instanciation de classe, la méthode s'appelle init : c ' est le constructeur de la classe. Si cette méthode n'est pas implémentée dans la classe, alors un constructeur par défaut est automatiquement appelé, comme c'est le cas dans cet exemple. Il est désormais possible d'afficher quelques informations sur la console :

```
print(instance)  
>>> <__main__.MaClasse object at 0x10f039f60>
```

Les Bases du langage PYTHON

Lorsqu'on affiche sur la sortie standard la variable **instance**, l'interpréteur informe qu'il s'agit d'un objet de type `__main__`. `MaClasse` dont l'adresse mémoire est `0x10f039f60`.

Il s'agit du module dans lequel `MaClasse` a été déclarée. En Python, un fichier source correspond à un module, et le fichier qui sert de point d'entrée à l'interpréteur est appelé `__main__`. Le nom du module est accessible via la variable spéciale `__name__`.

Les Bases du langage PYTHON

```
# Ceci est le module b.  
print("Nom du module du fichier b.py : " + __name__)  
chiffre = 42
```

```
# Ceci est le module a.  
print("Nom du module du fichier a.py : " + __name__)  
import b  
print(b.chiffre)
```

```
$> python a.py  
Nom du module du fichier a.py : __main__  
Nom du module du fichier b.py : b
```

Les Bases du langage PYTHON

Sortie standard

Le fichier a.py sert d'entrée à l'interpréteur Python : ce module se voit donc attribuer `_main_` comme nom. Lors de l'import du module b.py , le fichier est entièrement lu et affiche le nom du module qui correspond, lui, au nom du fichier. Une classe faisant toujours partie d'un module, la classe MaClasse a donc été assignée au `module_main_`.

Les Bases du langage PYTHON

L'adresse hexadécimale de la variable instance correspond à l'emplacement mémoire réservé pour stocker cette variable. Cette adresse permet, entre autres, de différencier deux variables qui pourraient avoir la même valeur.

```
a = MaClasse()
print("Info sur 'a': {}".format(a))
>>> Info sur 'a': <__main__.MaClasse object at 0x10b61f908>

b = a
print("Info sur 'b' : {}".format(b))
>>> Info sur 'b' : <__main__.MaClasse object at 0x10b61f908>

b = MaClasse()
print("Info sur 'b': {}".format(b))
>>> Info sur 'b': <__main__.MaClasse object at 0x10b6797b8>
```

Les Bases du langage PYTHON

L'adresse de **a** est **0x10b61f908**. Lorsqu'on assigne **a** à **b** et qu'on affiche **b**, on constate que les variables pointent vers la même zone mémoire. Par contre, si l'on assigne à **b** une nouvelle instance de **MaClasse**, alors son adresse n'est plus la même : il s'agit d'une nouvelle zone mémoire allouée pour stocker un nouvel exemplaire de **MaClasse**.

```
print(MaClasse)
>>> <class '__main__.MaClasse'>
```

```
classe = MaClasse
print(classe)
>>> <class '__main__.MaClasse'>
```

Les Bases du langage PYTHON

- En Python, tout est objet, y compris les classes. N'importe quel objet peut être affecté à une variable, et les classes ne font pas exception. Il est donc tout à fait valide d'assigner **MaClasse** à une variable **classe**, et son affichage sur la sortie standard confirme bien que **classe** est une classe, et pas une instance. D'où l'importance des parenthèses lorsqu'on désire effectuer une instantiation de classe.
- En effet, les parenthèses précisent bien qu'on appelle le constructeur de la classe, et on obtient par conséquent une instance. L'omission des parenthèses signifie que l'on désigne la classe elle-même.

Membres d'une classe

Attribut

Un attribut est une variable associée à une classe. Pour définir un attribut au sein d'une classe, il suffit :
d'assigner une valeur à cet attribut dans le corps de la classe :

```
class Cercle:
    # Déclaration d'un attribut de classe 'rayon'
    # auquel on assigne la valeur 2.
    rayon = 2

print(Cercle.rayon)

>>> 2
```

Les Bases du langage PYTHON

Attribut

d'assigner une valeur à cet attribut en dehors de la classe. On parle alors d'attribut dynamique :

```
class Cercle:
    # Le corps de la classe est laissé vide.
    pass

# Déclaration dynamique d'un attribut de classe 'rayon'
# auquel on assigne la valeur 2.
Cercle.rayon = 2

print(Cercle.rayon)

>>> 2
```

Les Bases du langage PYTHON

- Les attributs définis ainsi sont appelés «attributs de classe» car ils sont liés à la classe, par opposition aux «attributs d'instance» dont la vie est liée à l'instance à laquelle ils sont rattachés. Les attributs de classe sont automatiquement reportés dans les instances de cette classe et deviennent des attributs d'instance.

```
c = Cercle()
print(c.rayon)
>>> 2
```


Les Bases du langage PYTHON

Puisqu'un attribut de classe est lié à la classe, et non pas à l'instance, il existe durant toute la durée d'exécution du programme. Plus précisément, il existe tant que la classe à laquelle il est lié est définie. Un attribut d'instance, quant à lui, n'existe qu'à travers l'instance qui lui est liée. Ainsi, si l'instance est détruite, l'attribut d'instance l'est également.

Les Bases du langage PYTHON

```
# Déclaration d'une instance de Cercle.
c = Cercle()
# Déclaration d'un attribut d'instance 'rayon'.
c.rayon = 5
# Affichage de cet attribut d'instance.
print(c.rayon)
>>> 5

# Destruction de l'instance de Cercle.
del(c)
# Affichage de l'attribut d'instance.
print(c.rayon)
>>> Traceback (most recent call last):
      File "a.py", line 28, in <module>
        print(c.rayon)
NameError: name 'c' is not defined
# c n'est plus défini dans l'environnement d'exécution Python.
# Par conséquent, les attributs liés à cette instance non plus.
```

Les Bases du langage PYTHON

- Tandis qu'un attribut de classe survit à toutes les instances de cette classe.

```
# Déclaration d'une instance de Cercle.  
c = Cercle()  
  
# Destruction de l'instance de Cercle.  
del(c)  
  
# Affichage de l'attribut de classe.  
print(Cercle.rayon)  
  
>>> 2  
  
# L'attribut de classe existe toujours.
```

Les Bases du langage PYTHON

- Si l'attribut de classe est copié dans chaque objet instancié en tant qu'attribut d'instance, il n'en demeure pas moins que ces attributs demeurent indépendants l'un de l'autre. Toute modification sur l'attribut d'instance n'a aucun impact sur l'attribut de classe. De façon similaire, si la valeur de l'attribut de classe est changée, cela n'a aucun impact sur les objets déjà instanciés (mais cela en aura évidemment sur les futures instances) :

Les Bases du langage PYTHON

```
c.rayon = 4
print(c.rayon)

>>> 4
# Attribut de l'instance c.

print(Cercle.rayon)

>>> 2
# Attribut de la classe Cercle.

Cercle.rayon = 6
print(Cercle.rayon)

>>> 6
# Attribut de la classe Cercle dont la valeur
# vient d'être modifiée.

print(c.rayon)

>>> 4
# Attribut de l'instance c, qui demeure inchangé.
```

Les Bases du langage PYTHON

- Le choix entre attribut de classe et attribut d'instance dépend de la portée que l'on veut accorder à cet attribut. Si l'on désire que la valeur soit globale à tout le programme, alors l'attribut de classe est la solution car il ne dépend d'aucune instance. Cependant, toute modification de sa valeur a par conséquent des répercussions sur l'ensemble de l'application, ce qui peut entraîner des comportements inattendus si ce changement n'est pas totalement maîtrisé.
- Ces bogues ont d'autant plus de risques d'apparaître que le code est volumineux. L'attribut de classe est donc à utiliser avec précaution. Si la valeur de l'attribut est dépendante de chaque instance de la classe, alors l'attribut d'instance est la bonne solution. Cette valeur a une durée de vie égale à l'objet qui la contient, et tout changement n'a que des implications locales à l'objet.

Les Bases du langage PYTHON

Méthode

Une méthode est une fonction définie dans une classe ayant comme premier argument une instance de cette classe.

```
class Cercle:

    # Déclaration d'une méthode nommée perimetre.
    def perimetre(self):
        # Définition du corps de la méthode,

        # en l'occurrence avec une valeur de retour.
        return 2 * 3.14 * self.rayon

c = Cercle()
c.rayon = 2
# Appel de la méthode perimetre() de l'instance c.
print(c.perimetre())
>>> 12.56
```

Les Bases du langage PYTHON

- L'utilisation du mot-clé **def** se fait comme lorsqu'on définit une fonction dans l'espace de noms global. Le faire dans le corps de la classe, à l'instar des attributs, lie la fonction à la classe. Cependant, pour être appelée, une méthode doit obligatoirement prendre en premier argument une instance de la classe à laquelle elle est liée. La convention est de nommer cet argument **self**.
- NB: self n'est pas un mot clé du langage comme **class** ou **while**. Vous pouvez très bien donner un autre nom à ce premier paramètre. Ceci est cependant très déconseillé.

Les Bases du langage PYTHON

- Oublier l'argument self provoque une erreur lors de l'appel à cette «méthode»:

```
class Cercle2:

    def perimetre():
        return 2 * 3.14 * rayon

c2 = Cercle2()
c2.rayon = 2
print(c2.perimetre())
>>> Traceback (most recent call last):
      File "a.py", line 22, in <module>
        print(c2.perimetre())
      TypeError: perimetre() takes no arguments (1 given)
```

Les Bases du langage PYTHON

- Le message d'erreur peut prêter à confusion : aucun paramètre n'est donné à la méthode `perimetre()`, alors pourquoi l'interpréteur Python se plaint-il d'en recevoir un alors qu'il en attend à juste titre zéro ? Contrairement à d'autres langages orientés objet où l'instance est automatiquement accessible dans la méthode (via le mot clé `this` en C++ par exemple), en Python, l'instance est un paramètre de la méthode qui lui est automatiquement passé en argument à l'appel.
- Lorsqu'on appelle une méthode via une instance, c'est en fait la méthode «de classe» qui est appelée avec l'instance en premier paramètre `self`. Par conséquent, ces deux lignes sont complètement équivalentes :

```
print(c.perimetre())  
>>> 12.56
```

```
print(Cercle.perimetre(c))  
>>> 12.56
```

Les Bases du langage PYTHON

Si l'on appelle une méthode n'ayant pas self comme premier argument, Python devient un peu plus explicite :

```
print(Cercle2.perimetre(c2))
>>> Traceback (most recent call last):
      File "a.py", line 22, in <module>
        print(Cercle2.perimetre(c2))
TypeError: perimetre() takes no arguments (1 given)
```

- Si, conformément au message d'erreur, on omet l'argument de perimetre(), on obtient :

```
print(Cercle2.perimetre())
>>> Traceback (most recent call last):
      File "a.py", line 22, in <module>
        print(Cercle2.perimetre())
TypeError: unbound method perimetre() must be called with Cercle2 instance as first argument (got nothing instead)
```

Active Windows

Les Bases du langage PYTHON

- L'erreur est alors plus claire : la méthode doit être appelée avec une instance de Cercle2 comme premier argument. Ce n'est effectivement pas le cas puisque l'argument self n'a pas été déclaré dans la liste des paramètres.
- En outre, ce message d'erreur apporte une autre lumière sur les mécanismes en œuvre : il s'agit du terme «unbound method» (méthode non liée). En Python, tout est objet, y compris les méthodes:

```
# Affichage de la méthode « de classe ».  
print(Cercle.perimetre)  
>>> <unbound method Cercle.perimetre>
```

```
# Affichage de la méthode « d'instance ».  
print(c.perimetre)  
>>> <bound method Cercle.perimetre of <__main__.Cercle instance at 0x104520248>>
```

Les Bases du langage PYTHON

«unbound method» signifie que la méthode n'est liée à aucune instance, et par conséquent, ne peut être appelée sans argument. Afin de l'utiliser correctement, il faut donc lui fournir une instance de Cercle comme premier argument : c'est le fameux self. Par contre, une «bound method» est bien liée à une instance de Cercle, qui est à l'adresse 0x104520248. Puisqu'elle est déjà liée, plus besoin de lui fournir l'instance : l'argument self est déjà établi. En tant qu'objet, une méthode peut également être assignée à une variable :

Les Bases du langage PYTHON

```
////////////////////////////////////  
p = Cercle.perimetre  
c = Cercle()  
c.rayon = 2  
print(p)  
  
>>> <function Cercle.perimetre at 0x10a564730>  
# p est la méthode perimetre de la classe Cercle.  
# Cette méthode est stockée à l'adresse mémoire indiquée.  
  
print(p(c))  
  
>>> 12.56  
# L'appel de la méthode de classe avec une instance  
# en paramètre produit le résultat attendu.  
  
p = c.perimetre  
print(p)  
  
>>> <bound method Cercle.perimetre of <__main__.Cercle object at  
0x104520248>>  
# p est la méthode perimetre de la classe Cercle liée  
# à l'instance de Cercle qui est stockée à l'adresse indiquée.  
  
print(p())  
  
>>> 12.56  
# L'appel sans argument de la méthode liée à l'instance
```

Les Bases du langage PYTHON

- Il est bien important d'observer l'utilisation des parenthèses pour bien comprendre ces exemples. Les parenthèses sont utilisées afin de provoquer l'appel d'une méthode. Sans parenthèses, on n'appelle pas : on accède à la méthode comme s'il s'agissait d'un attribut «classique», par exemple un entier ou une chaîne de caractères. Une fois la variable p assignée, comme il s'agit désormais d'une méthode, l'utilisation des parenthèses est de mise pour effectivement appeler cette méthode.
- On dit que p est «appelable». Python propose la fonction native callable() qui vérifie si un objet peut être utilisé comme une fonction, s'il est callable.

```
print(callable(p))  
>>> True
```

Les Bases du langage PYTHON

- L'utilisation de méthodes en tant qu'objets permet de puissantes combinaisons : stockage en tant qu'attribut, liste de méthodes, méthodes qui en prennent d'autres en argument..

```
c = Cercle()
c.rayon = 3
# Déclaration d'une liste de méthodes.
methodes_cercle = [Cercle.diametre, Cercle.perimetre, Cercle.aire]
# Boucle parcourant cette liste de méthodes.
for m in methodes_cercle:
    # Affichage de l'appel de la méthode courante m
    # avec l'instance de Cercle c comme premier argument (self).
    print(m(c))
>>> 6 # Résultat de c.diametre().
18.84 # Résultat de c.perimetre().
28.26 # Résultat de c.aire().
```


Les Bases du langage PYTHON

Méthodes

- Il est bien important d'observer l'utilisation des parenthèses pour bien comprendre ces exemples. Les parenthèses sont utilisées afin de provoquer l'appel d'une méthode.
- Sans parenthèses, on n'appelle pas : on accède à la méthode comme s'il s'agissait d'un attribut «classique», par exemple un entier ou une chaîne de caractères. Une fois la variable p assignée, comme il s'agit désormais d'une méthode, l'utilisation des parenthèses est de mise pour effectivement appeler cette méthode.

Les Bases du langage PYTHON

- **Constructeur**

Pour des raisons de facilité d'écriture et de flexibilité, il est possible de personnaliser le processus d'instanciation d'une classe en implémentant la méthode `__init__` de cette classe :

```
class MaClasse:

    # Déclaration du constructeur de MaClasse
    # comme une méthode classique.
    def __init__(self):
        print("Construction de MaClasse")

# Instanciation de MaClasse, et donc appel au constructeur.
instance = MaClasse()

>>> Construction de MaClasse
```

Les Bases du langage PYTHON

- Le constructeur de MaClasse a été surchargé par cette implémentation, qui ne fait qu'afficher un message sur la sortie standard. Ce message s'affiche dès que l'instanciation de MaClasse est faite (lors de l'assignation à instance).
- L'une des utilisations principales d'un constructeur est d'assigner des valeurs par défaut aux attributs d'instance. En effet, les attributs d'instance sont souvent préférés aux attributs de méthode car il est finalement peu courant d'avoir besoin de variables qui puissent être accessibles durant toute l'existence de la classe. Il est possible de définir dynamiquement des attributs, mais afin de rendre le code plus lisible, il est préférable de centraliser la définition de ces attributs d'instance dans le constructeur de la classe.

Les Bases du langage PYTHON

- De plus, cela permet d'avoir un objet cohérent d'un point de vue métier : tous les attributs sont initialisés avec des valeurs qui ont du sens et qui ne provoqueront pas de comportement inattendu une fois l'objet instancié.

```
class MaClasse:

    def __init__(self):
        # Initialisation d'un attribut d'instance.
        self.alpha = 1

i = MaClasse()
print(i.alpha)
>>> 1
# L'attribut alpha a été initialisé à 1 dans le constructeur

print(i.beta)
>>> Traceback (most recent call last):
      File "a.py", line 8, in <module>
        print(i.beta)
AttributeError: MaClasse instance has no attribute 'beta'
# L'attribut beta n'existe pas : il n'a été déclaré
# ni dans la classe, ni dans le constructeur.
```

Les Bases du langage PYTHON

- Il est possible, comme pour toute méthode, de passer des arguments supplémentaires au constructeur afin de personnaliser l'instanciation :

```
class Cercle:

    def __init__(self, rayon):
        self.rayon = rayon

    def diametre(self):
        return self.rayon * 2

# Création d'un cercle dont le constructeur prend comme un argument
# (en plus de self) la longueur de son rayon.
c = Cercle(3)
print(c.diametre())
>>> 6
```

Les Bases du langage PYTHON

Destructeur

- Le pendant du constructeur, qui est la méthode appelée lors de l'instanciation d'une classe, est le destructeur, qui est la méthode appelée lorsque l'instance est détruite.
- L'environnement Python dispose d'un outil appelé le «ramassemiettes», qui se charge automatiquement de supprimer les objets qui ne sont plus utilisés dans le programme, libérant ainsi de la mémoire qui peut être recyclée pour instancier d'autres classes.
- La suppression d'un objet provoque l'appel de sa méthode `__del__`, tout comme l'instanciation invoque la méthode `__init__`.

Les Bases du langage PYTHON

Destructeur

À moins de l'appeler explicitement (`MonObjet._del_()`), le destructeur d'une instance ne sera appelé que par le ramasse miettes. Cet appel ne se fait que lorsque plus aucune référence n'est faite à cette instance, que ce soit via une variable, un attribut de classe, une liste... Pour supprimer une référence à une instance, il convient d'utiliser le mot-clé `del`.

Les Bases du langage PYTHON

```
class Test:
    # Surcharge du destructeur de la classe Test
    def __del__(self):
        print("Destruction")

# Première référence à l'instance de Test()
test1 = Test()
# Deuxième référence à l'instance de Test()
test2 = test1
# On supprime une référence à l'instance de Test()
del test1
# L'instance n'est pas détruite car il y a encore une référence.
print("Pas de destruction")
>>> Pas de destruction
# On supprime la deuxième référence à l'instance de Test()
del test2
>>> Destruction # Plus de référence à l'instance :
# le ramasse-miettes la détruit.
```


Les Bases du langage PYTHON

- L'intérêt du destructeur est de garder un environnement de programme «propre». Par exemple : ne pas laisser une ressource externe dans un état qui la rendrait inutilisable après la destruction de l'instance. Un fichier ouvert et qui n'a pas été fermé rend son écriture impossible (un système d'exploitation digne de ce nom n'autorisera pas une écriture pendant qu'une lecture est en cours). Fichiers, connexions réseau, espace mémoire... il existe beaucoup de ressources que le ramasse-miettes ne peut pas remettre en l'état après leur utilisation par l'application Python.

Encapsulation

L'encapsulation est le principe interdisant l'accès direct aux attributs d'une classe. On ne dialogue avec l'objet qu'à travers une interface définissant les services accessibles à ses utilisateurs.

Syntaxe

Tout attribut qui commence par les symboles `__` est considéré comme privé.

`__attributprive=valeur`

De la même manière on peut rendre une méthode privée, faire précéder son nom par un double underscore.

Encapsulation

Pour respecter le principe d'encapsulation, il est conseillé de déclarer tous les attributs de façon privée

```
class Etdudiant:
    """gestion informations étudiants"""
    def __init__(self,nom,ine,note):
        self.__nom = nom
        self.__ine = ine
        self.__note= note
    def afficherNote(self):
        print("La note :",self.__note)
```

Accesseurs

Un accesseur, appelé également getter, est une méthode retournant la valeur d'un attribut. On sera donc souvent amené à créer autant d'accesseurs que d'attributs.

```
class Etdudiant:
    """gestion informations étudiants"""
    def __init__(self,nom,ine,note):
        self.__nom = nom
        self.__ine = ine
        self.__note= note
    def afficherNote(self):
        print("La note :",self.__note)
    def getNote(self):
        return self.__note
print(etu.getNote())
```

Mutateurs

Un mutateur, appelé également setter, est une méthode fixant la valeur d'un attribut. On sera donc souvent amené à créer autant de mutateurs que d'attributs..

```
class Etdudiant:
```

```
    """gestion informations étudiants"""
```

```
    def __init__(self,nom,ine,note):
```

```
        self.__nom = nom
```

```
        self.__ine = ine
```

```
        self.__note= note
```

```
    def afficherNote(self):
```

```
        print("La note :",self.__note)
```

```
    def setNote(self,note):
```

```
        self.__note=note
```

```
etu.setNote(12)
```

Héritage

L'héritage est le mécanisme par lequel une classe possède les membres d'une autre classe, afin d'éventuellement les spécialiser ou d'en ajouter de nouveaux. La syntaxe Python est la suivante :

La classe Cercle hérite de la classe Forme et récupère donc les deux attributs x et y qui représentent les coordonnées de son centre. Cependant, ce centre est propre à l'instance de la forme, et ces attributs devraient être initialisés dans le constructeur. Lorsqu'une classe dérivée est instanciée, c'est son constructeur qui appelle le constructeur de la classe de base. Dans le cas d'un constructeur par défaut, comme c'est ici le cas pour la classe Cercle, cette tâche est effectuée automatiquement.

Définition de la classe de base.

```
class Forme:
```

```
    x = 0
```

```
    y = 0
```

Définition de la classe dérivée.

```
class Cercle(Forme):
```

```
    # Le corps de la classe dérivée est vide.
```

```
    pass
```

```
c = Cercle()
```

```
print(c.x, c.y)
```

```
>>> 0 0
```

Héritage

```
# Définition de la classe de base.
class Forme:
    # Constructeur de la classe de base.
    def __init__(self):
        print("Init Forme")
        # Initialisation des attributs d'instance.
        self.x = 0
        self.y = 0

# Définition de la classe dérivée.
class Cercle(Forme):
    # Constructeur de la classe dérivée, qui n'appelle pas
    # le constructeur de la classe de base.
    def __init__(self):
        print("Init Cercle")

c = Cercle()
>>> Init Cercle # Constructeur de Cercle appelé,
                # mais pas celui de Forme.
```

Mais en cas de réimplémentation du constructeur, il ne faut pas oublier de faire explicitement cet appel, sous peine de comportement inattendu

Héritage

```
c = Cercle()

>>> Init Cercle # Constructeur de Cercle appelé,
                # mais pas celui de Forme.

print(c.x, c.y)

>>> Traceback (most recent call last):
      File "/Users/mankalas/a.py", line 14, in <module>
        print(c.x, c.y)
AttributeError: 'Cercle' object has no attribute 'x'
```

le constructeur de la classe de base n'est pas appelé explicitement

Héritage

```
class Cercle(Forme):  
  
    def __init__(self):  
        # Appel explicite au constructeur de la classe de base.  
        Forme.__init__(self)  
        print("Init Cercle")
```

```
c = Cercle()  
>>> Init Forme  
Init Cercle  
  
print(c.x, c.y)  
>>> 0 0
```

Appeler le constructeur de la classe de base doit se faire explicitement :

Héritage

Il est important d'appeler le constructeur de la classe de base car il faut être certain que les membres de cette classe de base sont bien initialisés avant de potentiellement les utiliser dans le constructeur de la classe dérivée. Par exemple, le constructeur d'une classe de base **A** ouvre une communication réseau avec un serveur pour y lire des données et les stocker dans un attribut. Si une classe dérivée **B** utilise ces données pour s'initialiser et si elle ne fait pas appel au constructeur de la classe **A**, alors les données requises ne sont pas disponibles et le constructeur de **B** va échouer ou provoquer une erreur.

Héritage

Plutôt que d'écrire explicitement le nom de la classe de base afin d'appeler son constructeur, il existe une façon plus élégante et évolutive qui est d'utiliser l'objet `super`. `Super` est un objet un peu particulier qui délègue tout appel de méthode ou d'attribut à la classe mère de l'objet à partir duquel on l'appelle. Dans l'exemple précédent, il suffit de remplacer **`Forme.__init__(self)`** par **`super().__init__()`** pour obtenir un appel au constructeur de la classe de base.

En effet, étant donné que `super` délègue le travail à la classe de base, **`super().__init__()`** ne fait qu'appeler le constructeur de `Forme` en lui passant implicitement en argument l'instance de `Cercle` en cours d'initialisation :

Héritage

```
class Forme:
```

```
    def __init__(self):
        print("Init Forme instance", self)
        self.x = 0
        self.y = 0
```

```
class Cercle(Forme):
```

```
    def __init__(self):
        super().__init__()
        print("Init Cercle instance", self)
```

```
c = Cercle()
```

```
>>> Init Forme instance <__main__.Cercle object at 0x10c227860>
```

```
Init Cercle instance <__main__.Cercle object at 0x10c227860>
```

```
print(c.x, c.y)
```

```
>>> 0 0
```

```
# Les attributs hérités de Forme sont bien présents.
```

Héritage

Python fournit quelques fonctions de base afin de vérifier les liaisons d'héritage ;

- **type()** retourne le type de l'objet passé en paramètre. Cette valeur de retour est elle-même du type **type**, qui est un type spécial représentant les classes Python.
- **isinstance()** indique si l'objet passé en premier paramètre est bien une instance de la classe passée en second paramètre. Il est important de se souvenir qu'une classe dérivée est considérée comme une instance de sa classe de base.
- **issubclass()** détermine si la classe donnée en premier paramètre est bien une sous-classe de la classe donnée en second paramètre.

Héritage

```
class Forme:
    pass

class Cercle(Forme):
    pass

c = Cercle()
f = Forme()

print(type(c))
>>> <class '__main__.Cercle'>

print(type(f))
>>> <class '__main__.Forme'>

print(isinstance(c, Cercle))
>>> True # c est une instance de Cercle.

print(isinstance(c, Forme))
>>> True # c est également une instance de Forme.

print(isinstance(f, Cercle))
>>> False # f n'est pas une instance de Cercle...

print(isinstance(f, Forme))
>>> True # ... mais est bien une instance de Forme.

print(issubclass(Cercle, Cercle))
>>> True # Cercle est une sous-classe de Forme.

print(issubclass(Cercle, Forme))
>>> True # issubclass considère qu'une classe est sous-classe
# d'elle-même.
```

Polymorphisme

Le polymorphisme est un terme désignant les mécanismes où l'on manipule des objets de différents types en tant qu'objets d'un seul et unique type. Le polymorphisme permet de s'abstraire du type «véritable» des objets pour ne se contenter que du type dont ils héritent tous. Le terme «polymorphisme» s'applique aux méthodes d'une classe de base qui sont ré-implémentées dans les classes dérivées. On parle alors de «surchage» de méthode. Le polymorphisme est donc intimement lié au concept d'héritage. L'intérêt majeur est de pouvoir manipuler un ensemble hétéroclite de classes exactement de la même façon, sans avoir besoin de connaître leur implémentation réelle. En POO, il est préférable d'utiliser le moins d'informations possible pour mener à bien une tâche.

Polymorphisme

En d'autres termes, le polymorphisme est le mécanisme qui permet à une classe fille de redéfinir une méthode dont elle a hérité de sa classe mère, tout en gardant la même signature.

```
class rectangle:
    def __init__(self,x,y):
        self._x = x
        self._y = y
    def surface(self):
        return self._x*self._y
```


Polymorphisme

```
class paveDroit(rectangle):  
    def __init__(self,x,y,z):  
        super().__init__(x,y)  
        self.__z = z  
    def surface(self):  
        return 2*(self._x*self._y+self._x*self.__z+self._y*self.__z)
```

```
objetrect = rectangle(10,7)  
print(objetrect.surface())  
objetpave = paveDroit(10,7,5)  
print(objetpave.surface())
```

Polymorphisme Exemple 2

```
# La classe de base.

class Forme:

    # La méthode polymorphe, qui sera spécialisée
    # par chaque classe dérivée.
    def perimetre(self):
        # Message d'erreur.
        raise NotImplementedError("Impossible de calculer
le périmètre d'une forme générique")
```

```
from forme import Forme
```

```
# Une classe dérivée.
```

```
class Carre(Forme):
```

```
    def __init__(self, cote):
        self.cote = cote
```

```
# Surchage de la méthode de base.
```

```
    def perimetre(self):
        return 4 * self.cote
```

```
# Une autre classe dérivée.
```

```
class Cercle(Forme):
```

```
    def __init__(self, rayon):
        self.rayon = rayon
```

```
# Une nouvelle surcharge de la méthode de base.
```

Polymorphisme

```
from forme import Forme
```

```
# Une classe dérivée.
```

```
class Carre(Forme):
```

```
    def __init__(self, cote):  
        self.cote = cote
```

```
# Surchage de la méthode de base.
```

```
    def perimetre(self):  
        return 4 * self.cote
```

```
# Une autre classe dérivée.
```

```
class Cercle(Forme):
```

```
    def __init__(self, rayon):  
        self.rayon = rayon
```

```
# Une nouvelle surcharge de la méthode de base.
```

```
    def perimetre(self):  
        return 2 * 3.14 * self.rayon
```

```
# Création d'une liste de formes concrètes.  
formes = [Cercle(3), Carre(2), Carre(5)]
```

```
from formes_concretes import formes
```

```
# Parcours de la liste de formes, sans se soucier  
# de quelles formes concrètes il s'agit.
```

```
for forme in formes:  
    print(forme.perimetre())
```

```
$> python main.py  
18.84 # Périmètre de Cercle(3).  
8     # Périmètre de Carre(2).  
20    # Périmètre de Carre(5).
```

Fichiers

Python permet d'accéder au contenu d'un fichier texte enregistré sur disque. Par défaut, on supposera que le fichier texte est placé dans le répertoire d'exécution.

Soit par exemple le fichier fichier.txt suivant :

Mars
Jupiter
Uranus
Neptune

```
1 fichier = open("a.txt")  
2 contenu = fichier.read()  
3 print(contenu)
```

```
Mars  
Jupiter  
Uranus  
Neptune
```

Fichiers

Parcours des lignes d'un fichier

```
fichier = open("a.txt")

for L in fichier:
    print(L)
    print(len(L))
    print("-----")
```

Écriture dans un fichier

```
f = open("sortie.txt", "w")
message = "Mars\nJupiter\nSaturne\nUranus"
f.write(message)
```

Exceptions

Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution.

Les erreurs détectées durant l'exécution sont appelées des **exceptions** et ne sont pas toujours fatales, mais peuvent dans certains cas arrêter l'exécution normale d'un programme.

Analysons le script suivant :

```
chaine = input('Entrer un nombre : ')
```

```
nombre = float(chaine) # conversion de la chaine de caractère en  
réel
```

Exceptions

Quand vous entrez un nombre, tout se déroule normalement :

```
>>>
```

```
Entrer un nombre : 10
```

```
L'inverse de 10.0 est : 0.1
```

```
>>>
```

Mais que se passe-t-il autrement ?

```
>>>
```

```
Entrer un nombre : bonjour
```

```
Traceback (most recent call last):
```

```
  File "inverse.py", line 3, in <module>
    nombre = float(chaine)
```

```
ValueError: could not convert string to float: bonjour
```

```
>>>
```

Exceptions

Ou encore :

>>>

Entrer un nombre : 0

Traceback (most recent call last):

```
File "inverse.py", line 4, in <module>  
    inverse = 1.0/nombre
```

ZeroDivisionError: float division by zero

>>>

Exceptions

Il est possible de gérer les exceptions pour éviter l'arrêt brutal du programme.

Pour cela, on utilise conjointement les instructions `try` et `except`.

L'instruction `else` est optionnelle :

`try:`

```
chaine = input('Entrer un nombre : ')
nombre = float(chaine)
inverse = 1.0/nombre
```

`except:`

```
#ce bloc est exécuté si une exception est levée dans le #bloc try
print("Erreur !")
```

`Else:`

```
#on arrive ici si aucune exception n'est levée dans le #bloc try
print("L'inverse de",nombre,"est : ",inverse)
```

Exceptions

Il est possible de gérer les exceptions pour éviter l'arrêt brutal du programme.

Pour cela, on utilise conjointement les instructions `try` et `except`.

L'instruction `else` est optionnelle :

`try:`

```
chaine = input('Entrer un nombre : ')
nombre = float(chaine)
inverse = 1.0/nombre
```

`except:`

```
#ce bloc est exécuté si une exception est levée dans le #bloc try
,k print("Erreur !")
```

`Else:`

```
#on arrive ici si aucune exception n'est levée dans le #bloc try
print("L'inverse de",nombre,"est : ",inverse)
```

Exceptions

Résultat :

>>>

Entrer un nombre : 0

Division par zéro !

>>>

Entrer un nombre : bonjour
bonjour n'est pas un nombre !

>>>

***NB : Un programme bien écrit doit gérer
proprement les exceptions.***

Le module Turtle

La bibliothèque standard de Python dispose d'un module appelé Turtle et qui permet de créer des dessins dans une fenêtre. On peut tracer des lignes droites, des arcs de cercle et effectuer des remplissages. Par défaut, les dessins s'exécutent de manière animée et non instantanée.

```
from turtle import *  
hideturtle()  
goto(50,150)  
goto(200,0)  
  
goto(170,-30)  
goto(0,0)
```

Turtle

Rôle	commande	Action
Importation	<code>from turtle import *</code>	Importe toutes les fonctionnalités du module Turtle
Déplacement	<code>goto(x, y)</code>	Déplace la tortue du point courant vers le point (x, y)
Dessin désactivé	<code>up()</code>	Le stylet de dessin de la tortue est levé
Dessin réactivé	<code>down()</code>	Le stylet de dessin de la tortue est abaissé
Disque au point courant	<code>dot()</code>	Dessine un point noir de 5 pixels de diamètre
	<code>dot(color)</code>	Dessine un point de couleur color de 5 pixels de diamètre
	<code>dot(d, color)</code>	Dessine un point de couleur color de d pixels de diamètre
Cacher le pointeur	<code>hideturtle()</code>	rend invisible la « tête » de déplacement de la tortue
Couleurs	<code>"black" "white" "red" "green" "blue" "orange"</code>	nom de couleurs