ENTWANNE



Zeste de savoir

# La programmation orientée objet en Python

15 janvier 2019

# Table des matières

# 1. Introduction

Plus qu'un simple langage de script, Python est aussi un langage orienté objet.

Ce langage moderne et puissant est né au début des années 1990 sous l'impulsion de Guido van Rossum.

Apparue dans les années 60 quant à elle, la programmation orientée objet (POO) est un paradigme de programmation ; c'est-à-dire une façon de concevoir un programme informatique, reposant sur l'idée qu'un programme est composé d'objets interagissant les uns avec les autres.

En définitive, un objet est une donnée. Une donnée constituée de diverses propriétés, et pouvant être manipulée par différentes opérations.

La programmation orientée objet est le paradigme qui nous permet de définir nos propres types d'objets, avec leurs propriétés et opérations. Ce paradigme vient avec de nombreux concepts qui seront explicités le long de ce cours.

À travers ce tutoriel, nous allons nous intéresser à cette façon de penser et de programmer avec le langage Python.

Il vous est conseillé de maîtriser les bases de ce dernier (manipulation de valeurs, structures de contrôle, structures de données, fonctions) avant de continuer votre lecture.

Nous travaillerons ici avec la version 3 de Python (version 3.4 ou supérieure conseillée).

# 2. Objet et caractéristiques

Tout d'abord, qu'est-ce qu'un objet ? En introduction, nous avons dit qu'un objet avait des propriétés et pouvait être manipulé par des opérations, c'est bien beau tout ça mais qu'est-ce que ça peut bien vouloir dire concrétement ?

En introduction, nous avons décrit un objet comme une valeur possédant des propriétés et manipulée par des opérations.

Concrètement, un objet est constitué de 3 caractéristiques :

- Un type, qui identifie le rôle de l'objet (`int`, `str` et `list` sont des exemples de types d'objets) ;
- Des attributs, qui sont les propriétés de l'objet ;
- Des méthodes, les opérations qui s'appliquent sur l'objet.

Pour vous éclairer, prenons le code suivant :

```
>>> number = 5 # On instancie une variable `number` de type `int`
>>> number.numerator # `numerator` est un attribut de `number`
5
>>> values = [] # Variable `values` de type `list`
>>> values.append(number) # `append` est une méthode de `values`
>>> values
[5]
```

Toute valeur en Python est donc un objet.

## 2.1. Il a une drôle de tête ce type-là

Ainsi, tout objet est associé à un type. Un type définit la sémantique d'un objet. On sait par exemple que les objets de type `int` sont des nombres entiers, que l'on peut les additionner, les soustraire, etc.

Pour la suite de ce cours, nous utiliserons un type `User` représentant un utilisateur sur un quelconque logiciel. Nous pouvons créer ce nouveau type à l'aide du code suivant :

```
class User:
    pass
```

Nous reviendrons sur ce code par la suite, retenez simplement que nous avons maintenant à notre disposition un type `User`.

Pour créer un objet de type `User`, il nous suffit de procéder ainsi :

```
john = User()
```

On dit alors que `john` est une instance de `User`.

## 2.2. Montre-moi tes attributs

Ensuite, nous avons dit qu'un objet était constitué d'attributs. Ces derniers représentent des valeurs propres à l'objet.

Nos objets de type `User` pourraient par exemple contenir un identifiant (`id`), un nom (`name`) et un mot de passe (`password`).

En Python, nous pouvons facilement associer des valeurs à nos objets :

```
class User:
    pass

### Instanciation d'un objet de type User
john = User()

### Définition d'attributs pour cet objet
john.id = 1
john.name = 'john'
john.password = '12345'

print('Bonjour, je suis {}.'.format(john.name))
print('Mon id est le {}.'.format(john.id))
print('Mon mot de passe est {}.'.format(john.password))
```

Le code ci-dessus affiche :

```
Bonjour, je suis john.
Mon id est le 1.
Mon mot de passe est 12345.
```

Nous avons instancié un objet nommé `john`, de type `User`, auquel nous avons attribué trois attributs. Puis nous avons affiché les valeurs de ces attributs.

Notez que l'on peut redéfinir la valeur d'un attribut, et qu'un attribut peut aussi être supprimé à l'aide de l'opérateur `del`.

```
>>> john.password = 'mot de passe plus sécurisé !'
>>> john.password
'mot de passe plus sécurisé !'
>>> del john.password
>>> john.password
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'User' object has no attribute 'password'
```

Il est généralement déconseillé de nommer une valeur de la même manière qu'une fonction *built-in*. On évitera par exemple d'avoir une variable `id`, `type` ou `list`.
Dans le cas d'un attribut, cela n'est pas gênant car ne fait pas partie du même espace de noms. En effet, `john.id` n'entre pas en conflit avec `id`.

## 2.3. Discours de la méthode

Enfin, les méthodes sont les opérations applicables sur les objets. Ce sont en fait des fonctions qui recoivent notre objet en premier paramètre.

Nos objets `User` ne contiennent pas encore de méthode, nous découvrirons comment en ajouter dans le chapitre suivant. Mais nous pouvons déjà imaginer une méthode `check_pwd` (*check password*) pour vérifier qu'un mot de passe entré correspond bien au mot de passe de notre utilisateur.

```
def user_check_pwd(user, password):
    return user.password == password
```

```
>>> user_check_pwd(john, 'toto')
False
>>> user_check_pwd(john, '12345')
True
```

Les méthodes recevant l'objet en paramètre, elles peuvent en lire et modifier les attributs. Souvenez-vous par exemple de la méthode `append` des listes, qui permet d'insérer un nouvel élément, elle modifie bien la liste en question.

---

À travers cette partie nous avons défini et exploré la notion d'objet.

Un terme a pourtant été omis, le terme « classe ». Il s'agit en Python d'un synonyme de « type ». Un objet étant le fruit d'une classe, il est temps de nous intéresser à cette dernière et à sa construction.

# 3. Classes

Une classe est en fait la définition d'un type. `int`, `str` ou encore `list` sont des exemples de classes. `User` en est une autre.

Une classe décrit la structure des objets du type qu'elle définit : quelles méthodes vont être applicables sur ces objets.

## 3.1. La classe à Dallas

On définit une classe à l'aide du mot-clef `class` survolé plus tôt :

```
1 class User:
2     pass
```

(l'instruction `pass` sert ici à indiquer à Python que le corps de notre classe est vide)

Il est conseillé en Python de nommer sa classe en *CamelCase*, c'est à dire qu'un nom est composé d'une suite de mots dont la première lettre est une capitale. On préférera par exemple une classe `MonNomDeClasse` que `mon_nom_de_classe`. Exception faite des types *builtins* qui sont couramment écrits en lettres minuscules.

On instancie une classe de la même manière qu'on appelle une fonction, en suffixant son nom d'une paire de parenthèses. Cela est valable pour notre classe `User`, mais aussi pour les autres classes évoquées plus haut.

```
1 >>> User()
2 <__main__.User object at 0x7fc28e538198>
3 >>> int()
4 0
5 >>> str()
6 ''
7 >>> list()
8 []
```

La classe `User` est identique à celle du chapitre précédent, elle ne comporte aucune méthode. Pour définir une méthode dans une classe, il suffit de procéder comme pour une définition de fonction, mais dans le corps de la classe en question.

```
class User:
    def check_pwd(self, password):
        return self.password == password
```

Notre nouvelle classe `User` possède maintenant une méthode `check_pwd` applicable sur tous ses objets.

```
>>> john = User()
>>> john.id = 1
>>> john.name = 'john'
>>> john.password = '12345'
>>> john.check_pwd('toto')
False
>>> john.check_pwd('12345')
True
```

Quel est ce `self` reçu en premier paramètre par `check_pwd` ? Il s'agit simplement de l'objet sur lequel on applique la méthode, comme expliqué dans le chapitre précédent. Les autres paramètres de la méthode arrivent après.

La méthode étant définie au niveau de la classe, elle n'a que ce moyen pour savoir quel objet est utilisé. C'est un comportement particulier de Python, mais retenez simplement qu'appeler `john.check_pwd('12345')` équivaut à l'appel `User.check_pwd(john, '12345')`. C'est pourquoi `john` correspondra ici au paramère `self` de notre méthode.

`self` n'est pas un mot-clef du langage Python, le paramètre pourrait donc prendre n'importe quel autre nom. Mais il conservera toujours ce nom par convention.

Notez aussi, dans le corps de la méthode `check_pwd`, que `password` et `self.password` sont bien deux valeurs distinctes : la première est le paramètre reçu par la méthode, tandis que la seconde est l'attribut de notre objet.

## 3.2. Argumentons pour construire

Nous avons vu qu'instancier une classe était semblable à un appel de fonction. Dans ce cas, comment passer des arguments à une classe, comme on le ferait pour une fonction ?

Il faut pour cela comprendre les bases du mécanisme d'instanciation de Python. Quand on appelle une classe, un nouvel objet de ce type est construit en mémoire, puis initialisé. Cette initialisation permet d'assigner des valeurs à ses attributs.

L'objet est initialisé à l'aide d'une méthode spéciale de sa classe, la méthode `__init__`. Cette dernière recevra les arguments passés lors de l'instanciation.

```python
class User:
    def __init__(self, id, name, password):
        self.id = id
        self.name = name
        self.password = password

    def check_pwd(self, password):
        return self.password == password
```

Nous retrouvons dans cette méthode le paramètre `self`, qui est donc utilisé pour modifier les attributs de l'objet.

```
>>> john = User(1, 'john', '12345')
>>> john.check_pwd('toto')
False
>>> john.check_pwd('12345')
True
```

## 3.3. Comment veux-tu que je t'encapsule ?

### 3.3.1. Au commencement étaient les invariants

Les différents attributs de notre objet forment un état de cet objet, normalement stable. Ils sont en effet liés les uns aux autres, la modification d'un attribut pouvant avoir des conséquences sur un autre. Les invariants correspondent aux relations qui lient ces différents attributs.

Imaginons que nos objets `User` soient dotés d'un attribut contenant une évaluation du mot de passe (savoir si ce mot de passe est assez sécurisé ou non), il doit alors être mis à jour chaque fois que nous modifions l'attribut `password` d'un objet `User`.

Dans le cas contraire, le mot de passe et l'évaluation ne seraient plus corrélés, et notre objet `User` ne serait alors plus dans un état stable. Il est donc important de veiller à ces invariants pour assurer la stabilité de nos objets.

### 3.3.2. Protège-moi

Au sein d'un objet, les attributs peuvent avoir des sémantiques différentes. Certains attributs vont représenter des propriétés de l'objet et faire partie de son interface (tels que le prénom et le nom de nos objets `User`). Ils pourront alors être lus et modifiés depuis l'extérieur de l'objet, on parle dans ce cas d'attributs publics.

D'autres vont contenir des données internes à l'objet, n'ayant pas vocation à être accessibles depuis l'extérieur. Nous allons sécuriser notre stockage du mot de passe en ajoutant une méthode

pour le *hasher*[1] (à l'aide du module `crypt`), afin de ne pas stocker d'informations sensibles dans l'objet. Ce condensat du mot de passe ne devrait pas être accessible de l'extérieur, et encore moins modifié (ce qui en altérerait la sécurité).

De la même manière que pour les attributs, certaines méthodes vont avoir une portée publique et d'autres privée (on peut imaginer une méthode interne de la classe pour générer notre identifiant unique). On nomme encapsulation cette notion de protection des attributs et méthodes d'un objet, dans le respect de ses invariants.

Certains langages[2] implémentent dans leur syntaxe des outils pour gérer la visibilité des attributs et méthodes, mais il n'y a rien de tel en Python. Il existe à la place des conventions, qui indiquent aux développeurs quels attributs/méthodes sont publics ou privés. Quand vous voyez un nom d'attribut ou méthode débuter par un `_` au sein d'un objet, il indique quelque chose d'interne à l'objet (privé), dont la modification peut avoir des conséquences graves sur la stabilité.

```
import crypt

class User:
    def __init__(self, id, name, password):
        self.id = id
        self.name = name
        self._salt = crypt.mksalt() # sel utilisé pour le hash du
            mot de passe
        self._password = self._crypt_pwd(password)

    def _crypt_pwd(self, password):
        return crypt.crypt(password, self._salt)

    def check_pwd(self, password):
        return self._password == self._crypt_pwd(password)
```

```
>>> john = User(1, 'john', '12345')
>>> john.check_pwd('12345')
True
```

On note toutefois qu'il ne s'agit que d'une convention, l'attribut `_password` étant parfaitement visible depuis l'extérieur.

```
>>> john._password
'$6$DwdvE5H8sT71Huf/$9a.H/VIK4fdwIFdLJYL34yml/QC3KZ7'
```

1. Le *hashage* d'un mot de passe correspond à une opération non-réversible qui permet de calculer un condensat (*hash*) du mot de passe. Ce condensat peut-être utilisé pour vérifier la validité d'un mot de passe, mais ne permet pas de retrouver le mot de passe d'origine.
2. C++, Java, Ruby, etc.

Il reste possible de masquer un peu plus l'attribut à l'aide du préfixe `__`. Ce préfixe a pour effet de renommer l'attribut en y insérant le nom de la classe courante.

```
class User:
    def __init__(self, id, name, password):
        self.id = id
        self.name = name
        self.__salt = crypt.mksalt()
        self.__password = self.__crypt_pwd(password)

    def __crypt_pwd(self, password):
        return crypt.crypt(password, self.__salt)

    def check_pwd(self, password):
        return self.__password == self.__crypt_pwd(password)
```

```
>>> john = User(1, 'john', '12345')
>>> john.__password
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'User' object has no attribute '__password'
>>> john._User__password
'$6$kjwoqPPHRQAamRHT$591frrNfNNb3.RdLXYiB/bgdCC4Z0p.B'
```

Ce comportement pourra surtout être utile pour éviter des conflits de noms entre attributs internes de plusieurs classes sur un même objet, que nous verrons lors de l'héritage.

## 3.4. Tu aimes les glaces, canard ?

Un objet en Python est défini par sa structure (les attributs qu'il contient et les méthodes qui lui sont applicables) plutôt que par son type.

Ainsi, pour faire simple, un fichier sera un objet possédant des méthodes `read`, `write` et `close`. Tout objet respectant cette définition sera considéré par Python comme un fichier.

```
class FakeFile:
    def read(self, size=0):
        return ''
    def write(self, s):
        return 0
    def close(self):
        pass

```

```
f = FakeFile()
print('foo', file=f)
```

Python est entièrement construit autour de cette idée, appelée *duck-typing* : « Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard » (James Whitcomb Riley)

## 3.5. TP : Forum, utilisateurs et messages

Pour ce premier TP, nous allons nous intéresser aux classes d'un forum. Forts de notre type `User` pour représenter un utilisateur, nous souhaitons ajouter une classe `Post`, correspondant à un quelconque message.

Cette classe sera inititalisée avec un auteur (un objet `User`) et un contenu textuel (le corps du message). Une date sera de plus générée lors de la création.

Un `Post` possèdera une méthode `format` pour retourner le message formaté, correspondant au HTML suivant :

```
<div>
    <span>Par NOM_DE_L_AUTEUR le DATE_AU_FORMAT_JJ_MM_YYYY à
        HEURE_AU_FORMAT_HH_MM_SS</span>
    <p>
        CORPS_DU_MESSAGE
    </p>
</div>
```

De plus, nous ajouterons une méthode `post` à notre classe `User`, recevant un corps de message en paramètre et retournant un nouvel objet `Post`.

```
import crypt
import datetime

class User:
    def __init__(self, id, name, password):
        self.id = id
        self.name = name
        self._salt = crypt.mksalt()
        self._password = self._crypt_pwd(password)

    def _crypt_pwd(self, password):
        return crypt.crypt(password, self._salt)

    def check_pwd(self, password):
```

```
15          return self._password == self._crypt_pwd(password)
16
17      def post(self, message):
18          return Post(self, message)
19
20  class Post:
21      def __init__(self, author, message):
22          self.author = author
23          self.message = message
24          self.date = datetime.datetime.now()
25
26      def format(self):
27          date = self.date.strftime('le %d/%m/%Y à %H:%M:%S')
28          return
                '<div><span>Par {} {}</span><p>{}</p></div>'.format(self.author.nam
                date, self.message)
29
30  if __name__ == '__main__':
31      user = User(1, 'john', '12345')
32      p = user.post('Salut à tous')
33      print(p.format())
```

---

Nous savons maintenant définir une classe et ses méthodes, initialiser nos objets, et protéger les noms d'attributs/méthodes.

Mais jusqu'ici, quand nous voulons étendre le comportement d'une classe, nous la redéfinissons entièrement en ajoutant de nouveaux attributs/méthodes. Le chapitre suivant présente l'héritage, un concept qui permet d'étendre une ou plusieurs classes sans toucher au code initial.

# 4. Extension et héritage

Il n'est pas dans ce chapitre question de régler la succession de votre grand-tante par alliance, mais de nous intéresser à l'extension de classes.

Imaginons que nous voulions définir une classe `Admin`, pour gérer des administrateurs, qui réutiliserait le même code que la classe `User`. Tout ce que nous savons faire actuellement c'est copier/coller le code de la classe `User` en changeant son nom pour `Admin`.

Nous allons maintenant voir comment faire ça de manière plus élégante, grâce à l'héritage. Nous étudierons de plus les relations entre classes ansi créées.

Nous utiliserons donc la classe `User` suivante pour la suite de ce chapitre.

```
1  class User:
2      def __init__(self, id, name, password):
3          self.id = id
4          self.name = name
5          self._salt = crypt.mksalt()
6          self._password = self._crypt_pwd(password)
7
8      def _crypt_pwd(self, password):
9          return crypt.crypt(password, self._salt)
10
11     def check_pwd(self, password):
12         return self._password == self._crypt_pwd(password)
```

## 4.1. Hériter en toute simplicité

L'héritage simple est le mécanisme permettant d'étendre une unique classe. Il consiste à créer une nouvelle classe (fille) qui bénéficiera des mêmes méthodes et attributs que sa classe mère. Il sera aisé d'en définir de nouveaux dans la classe fille, et cela n'altèrera pas le fonctionnement de la mère.

Par exemple, nous voudrions étendre notre classe `User` pour ajouter la possibilité d'avoir des administrateurs. Les administrateurs (`Admin`) possèderaient une nouvelle méthode, `manage`, pour administrer le système.

```
class Admin(User):
    def manage(self):
        print('I am an über administrator!')
```

En plus des méthodes de la classe `User` (`__init__`, `_crypt_pwd` et `check_pwd`), `Admin` possède aussi une méthode `manage`.

```
>>> root = Admin(1, 'root', 'toor')
>>> root.check_password('toor')
True
>>> root.manage()
I am an über administrator!
>>> john = User(2, 'john', '12345')
>>> john.manage()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'User' object has no attribute 'manage'
```

Nous pouvons avoir deux classes différentes héritant d'une même mère

```
class Guest(User):
    pass
```

`Admin` et `Guest` sont alors deux classes filles de `User`.

L'héritage simple permet aussi d'hériter d'une classe qui hérite elle-même d'une autre classe.

```
class SuperAdmin(Admin):
    pass
```

`SuperAdmin` est alors la fille de `Admin`, elle-même la fille de `User`. On dit alors que `User` est une ancêtre de `SuperAdmin`.

On peut constater quels sont les parents d'une classe à l'aide de l'attribut spécial `__bases__` des classes :

```
>>> Admin.__bases__
(<class '__main__.User'>,)
>>> Guest.__bases__
(<class '__main__.User'>,)
>>> SuperAdmin.__bases__
(<class '__main__.Admin'>,)
```

Que vaudrait alors `User.__bases__`, sachant que la classe `User` est définie sans héritage ?

```
>>> User.__bases__
(<class 'object'>,)
```

On remarque que, sans que nous n'ayons rien demandé, `User` hérite de `object`. En fait, `object` est l'ancêtre de toute classe Python. Ainsi, quand aucune classe parente n'est définie, c'est `object` qui est choisi.

### 4.1.1. Sous-typage

Nous avons vu que l'héritage permettait d'étendre le comportement d'une classe, mais ce n'est pas tout. L'héritage a aussi du sens au niveau des types, en créant un nouveau type compatible avec le parent.

En Python, la fonction `isinstance` permet de tester si un objet est l'instance d'une certaine classe.

```
>>> isinstance(root, Admin)
True
>>> isinstance(root, User)
True
>>> isinstance(root, Guest)
False
>>> isinstance(root, object)
True
```

Mais gardez toujours à l'esprit qu'en Python, on préfère se référer à la structure d'un objet qu'à son type (*duck-typing*), les tests à base de `isinstance` sont donc à utiliser pour des cas particuliers uniquement, où il serait difficile de procéder autrement.

## 4.2. La redéfinition de méthodes, c'est super !

Nous savons hériter d'une classe pour y insérer de nouvelles méthodes, mais nous ne savons pas étendre les méthodes déjà présentes dans la classe mère. La redéfinition est un concept qui permet de remplacer une méthode du parent.

Nous voudrions que la classe `Guest` ne possède plus aucun mot de passe. Celle-ci devra modifier la méthode `check_pwd` pour accepter tout mot de passe, et simplifier la méthode `__init__`.

On ne peut pas à proprement parler étendre le contenu d'une méthode, mais on peut la redéfinir :

```python
class Guest(User):
    def __init__(self, id, name):
        self.id = id
        self.name = name
        self._salt = ''
        self._password = ''

    def check_pwd(self, password):
        return True
```

Cela fonctionne comme souhaité, mais vient avec un petit problème, le code de la méthode `__init__` est répété. En l'occurrence il ne s'agit que de 2 lignes de code, mais lorsque nous voudrons apporter des modifications à la méthode de la classe `User`, il faudra les répercuter sur `Guest`, ce qui donne vite quelque chose de difficile à maintenir.

Heureusement, Python nous offre un moyen de remédier à ce mécanisme, super ! Oui, `super`, littéralement, une fonction un peu spéciale en Python, qui nous permet d'utiliser la classe parente (*superclass*).

`super` est une fonction qui prend initialement en paramètre une classe et une instance de cette classe. Elle retourne un objet *proxy*[3] qui s'utilise comme une instance de la classe parente.

```
>>> guest = Guest(3, 'Guest')
>>> guest.check_pwd('password')
True
>>> super(Guest, guest).check_pwd('password')
False
```

Au sein de la classe en question, les arguments de `super` peuvent être omis (ils correspondront à la classe et à l'instance courantes), ce qui nous permet de simplifier notre méthode `__init__` et d'éviter les répétitions.

```python
class Guest(User):
    def __init__(self, id, name):
        super().__init__(id, name, '')

    def check_pwd(self, password):
        return True
```

On notera tout de même que contrairement aux versions précédentes, l'initialisateur de `User` est appelé en plus de celui de `Guest`, et donc qu'un sel et un *hash* du mot de passe sont générés alors qu'ils ne serviront pas.

3. Un *proxy* est un intermédiaire transparent entre deux entités.

Ça n'est pas très grave dans le cas présent, mais pensez-y dans vos développements futurs, afin de ne pas exécuter d'opérations coûteuses inutilement.

## 4.3. Une classe avec deux mamans

Avec l'héritage simple, nous pouvions étendre le comportement d'une classe. L'héritage multiple va nous permettre de le faire pour plusieurs classes à la fois.

Il nous suffit de préciser plusieurs classes entre parenthèses lors de la création de notre classe fille.

```
1  class A:
2      def foo(self):
3          return '!'
4
5  class B:
6      def bar(self):
7          return '?'
8
9  class C(A, B):
10     pass
```

Notre classe `C` a donc deux mères : `A` et `B`. Cela veut aussi dire que les objets de type `C` possèdent à la fois les méthodes `foo` et `bar`.

```
1  >>> c = C()
2  >>> c.foo()
3  '!'
4  >>> c.bar()
5  '?'
```

### 4.3.1. Ordre d'héritage

L'ordre dans lequel on hérite des parents est important, il détermine dans quel ordre les méthodes seront recherchées dans les classes mères. Ainsi, dans le cas où la méthode existe dans plusieurs parents, celle de la première classe sera conservée.

```
1  class A:
2      def foo(self):
3          return '!'
4
5  class B:
```

```
    def foo(self):
        return '?'

class C(A, B):
    pass

class D(B, A):
    pass
```

```
>>> C().foo()
'!'
>>> D().foo()
'?'
```

Cet ordre dans lequel les classes parentes sont explorées pour la recherche des méthodes est appelé *Method Resolution Order* (*MRO*). On peut le connaître à l'aide de la méthode `mro` des classes.

```
>>> A.mro()
[<class '__main__.A'>, <class 'object'>]
>>> B.mro()
[<class '__main__.B'>, <class 'object'>]
>>> C.mro()
[<class
    '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>
>>> D.mro()
[<class
    '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>
```

C'est aussi ce *MRO* qui est utilisé par `super` pour trouver à quelle classe faire appel. `super` se charge d'explorer le *MRO* de la classe de l'instance qui lui est donnée en second paramètre, et de retourner un *proxy* sur la classe juste à droite de celle donnée en premier paramètre.

Ainsi, avec `c` une instance de `C`, `super(C, c)` retournera un objet se comportant comme une instance de `A`, `super(A, c)` comme une instance de `B`, et `super(B, c)` comme une instance de `object`.

```
>>> c = C()
>>> c.foo() # C.foo == A.foo
'!'
>>> super(C, c).foo() # A.foo
'!'
>>> super(A, c).foo() # B.foo
'?'
```

```
>>> super(B, c).foo() # object.foo -> méthode introuvable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'foo'
```

Les classes parentes n'ont alors pas besoin de se connaître les unes les autres pour se référencer.

```
class A:
    def __init__(self):
        print("Début initialisation d'un objet de type A")
        super().__init__()
        print("Fin initialisation d'un objet de type A")

class B:
    def __init__(self):
        print("Début initialisation d'un objet de type B")
        super().__init__()
        print("Fin initialisation d'un objet de type B")

class C(A, B):
    def __init__(self):
        print("Début initialisation d'un objet de type C")
        super().__init__()
        print("Fin initialisation d'un objet de type C")

class D(B, A):
    def __init__(self):
        print("Début initialisation d'un objet de type D")
        super().__init__()
        print("Fin initialisation d'un objet de type D")
```

```
>>> C()
Début initialisation d'un objet de type C
Début initialisation d'un objet de type A
Début initialisation d'un objet de type B
Fin initialisation d'un objet de type B
Fin initialisation d'un objet de type A
Fin initialisation d'un objet de type C
<__main__.C object at 0x7f0ccaa970b8>
>>> D()
Début initialisation d'un objet de type D
Début initialisation d'un objet de type B
Début initialisation d'un objet de type A
Fin initialisation d'un objet de type A
Fin initialisation d'un objet de type B
Fin initialisation d'un objet de type D
```

```
<__main__.D object at 0x7f0ccaa971d0>
```

La méthode `__init__` des classes parentes n'est pas appelée automatiquement, et l'appel doit donc être réalisé explicitement.

C'est ainsi le `super().__init__()` présent dans la classe `C` qui appelle l'initialiseur de la classe `A`, qui appelle lui-même celui de la classe `B`. Inversement, pour la classe `D`, `super().__init__()` appelle l'initialiseur de `B` qui appelle celui de `A`.

On notera que les exemple donnés n'utilisent jamais plus de deux classes mères, mais il est possible d'en avoir autant que vous le souhaitez.

```
class A:
    pass

class B:
    pass

class C:
    pass

class D:
    pass

class E(A, B, C, D):
    pass
```

### 4.3.2. Mixins

Les *mixins* sont des classes dédiées à une fonctionnalité particulière, utilisable en héritant d'une classe de base et de ce *mixin*.

Par exemple, plusieurs types que l'on connaît sont appelés séquences (`str`, `list`, `tuple`). Ils ont en commun le fait d'implémenter l'opérateur `[]` et de gérer le *slicing*. On peut ainsi obtenir l'objet en ordre inverse à l'aide de `obj[::-1]`.

Un *mixin* qui pourrait nous être utile serait une classe avec une méthode `reverse` pour nous retourner l'objet inversé.

```
class Reversable:
    def reverse(self):
        return self[::-1]

class ReversableStr(Reversable, str):
    pass
```

```
7
8  class ReversableTuple(Reversable, tuple):
9      pass
```

```
1  >>> s = ReversableStr('abc')
2  >>> s
3  'abc'
4  >>> s.reverse()
5  'cba'
6  >>> ReversableTuple((1, 2, 3)).reverse()
7  (3, 2, 1)
```

Ou encore nous pourrions vouloir ajouter la gestion d'une photo de profil à nos classes `User` et dérivées.

```
1  class ProfilePicture:
2      def __init__(self, *args, **kwargs):
3          super().__init__(*args, **kwargs)
4          self.picture = '{}-{}.png'.format(self.id, self.name)
5
6  class UserPicture(ProfilePicture, User):
7      pass
8
9  class AdminPicture(ProfilePicture, Admin):
10     pass
11
12 class GuestPicture(ProfilePicture, Guest):
13     pass
```

```
1  >>> john = UserPicture(1, 'john', '12345')
2  >>> john.picture
3  '1-john.png'
```

## 4.4. TP : Fils de discussion

Vous vous souvenez de la classe `Post` pour représenter un message ? Nous aimerions maintenant pouvoir instancier des fils de discussion (`Thread`) sur notre forum.

Qu'est-ce qu'un fil de discussion ?

— Un message associé à un auteur et à une date ;
— Mais qui comporte aussi un titre ;

— Et une liste de posts (les réponses).

Le premier point indique clairement que nous allons réutiliser le code de la classe `Post`, donc en hériter.

Notre nouvelle classe sera initialisée avec un titre, un auteur et un message. `Thread` sera dotée d'une méthode `answer` recevant un auteur et un texte, et s'occupant de créer le post correspondant et de l'ajouter au fil. Nous changerons aussi la méthode `format` du `Thread` afin qu'elle concatène au fil l'ensemble de ses réponses.

La classe `Post` restera inchangée. Enfin, nous supprimerons la méthode `post` de la classe `User`, pour lui en ajouter deux nouvelles :

— `new_thread(title, message)` pour créer un nouveau fil de discussion associé à cet utilisateur ;
— `answer_thread(thread, message)` pour répondre à un fil existant.

```
1  import crypt
2  import datetime
3
4  class User:
5      def __init__(self, id, name, password):
6          self.id = id
7          self.name = name
8          self._salt = crypt.mksalt()
9          self._password = self._crypt_pwd(password)
10
11     def _crypt_pwd(self, password):
12         return crypt.crypt(password, self._salt)
13
14     def check_pwd(self, password):
15         return self._password == self._crypt_pwd(password)
16
17     def new_thread(self, title, message):
18         return Thread(title, self, message)
19
20     def answer_thread(self, thread, message):
21         thread.answer(self, message)
22
23 class Post:
24     def __init__(self, author, message):
25         self.author = author
26         self.message = message
27         self.date = datetime.datetime.now()
28
29     def format(self):
30         date = self.date.strftime('le %d/%m/%Y à %H:%M:%S')
31         return
               '<div><span>Par {} {}</span><p>{}</p></div>'.format(self.author.nar
               date, self.message)
```

```
32
33 class Thread(Post):
34     def __init__(self, title, author, message):
35         super().__init__(author, message)
36         self.title = title
37         self.posts = []
38
39     def answer(self, author, message):
40         self.posts.append(Post(author, message))
41
42     def format(self):
43         posts = [super().format()]
44         posts += [p.format() for p in self.posts]
45         return '\n'.join(posts)
46
47 if __name__ == '__main__':
48     john = User(1, 'john', '12345')
49     peter = User(2, 'peter', 'toto')
50     thread = john.new_thread('Bienvenue', 'Bienvenue à tous')
51     peter.answer_thread(thread, 'Merci')
52     print(thread.format())
```

# 5. Opérateurs

Il est maintenant temps de nous intéresser aux opérateurs du langage Python (`+`, `-`, `*`, etc.). En effet, un code respectant la philosophie du langage se doit de les utiliser à bon escient.

Ils sont une manière claire de représenter des opérations élémentaires (addition, concaténation, …) entre deux objets. `a + b` est en effet plus lisible qu'un `add(a, b)` ou encore `a.add(b)`.

Ce chapitre a pour but de vous présenter les mécanismes mis en jeu par ces différents opérateurs, et la manière de les implémenter.

## 5.1. Des méthodes un peu spéciales

Nous avons vu précédemment la méthode `__init__`, permettant d'initialiser les attributs d'un objet. On appelle cette méthode une méthode spéciale, il y en a encore beaucoup d'autres en Python. Elles sont reconnaissables par leur nom débutant et finissant par deux *underscores*.

Vous vous êtes peut-être déjà demandé d'où provenait le résultat affiché sur la console quand on entre simplement le nom d'un objet.

```
>>> john = User(1, 'john', '12345')
>>> john
<__main__.User object at 0x7fefd77fae10>
```

Il s'agit en fait de la représentation d'un objet, calculée à partir de sa méthode spéciale `__repr__`.

```
>>> john.__repr__()
'<__main__.User object at 0x7fefd77fae10>'
```

À noter qu'une méthode spéciale n'est presque jamais directement appelée en Python, on lui préférera dans le cas présent la fonction *builtin* `repr`.

```
>>> repr(john)
'<__main__.User object at 0x7fefd77fae10>'
```

Il nous suffit alors de redéfinir cette méthode `__repr__` pour bénéficier de notre propre représentation.

```
class User:
    ...

    def __repr__(self):
        return '<User: {}, {}>'.format(self.id, self.name)
```

```
>>> User(1, 'john', '12345')
<User: 1, john>
```

Une autre opération courante est la conversion de notre objet en chaîne de caractères afin d'être affiché via `print` par exemple. Par défaut, la conversion en chaîne correspond à la représentation de l'objet, mais elle peut être surchargée par la méthode `__str__`.

```
class User:
    ...

    def __repr__(self):
        return '<User: {}, {}>'.format(self.id, self.name)

    def __str__(self):
        return '{}-{}'.format(self.id, self.name)
```

```
>>> john = User(1, 'john', 12345)
>>> john
<User: 1, john>
>>> repr(john)
'<User: 1, john>'
>>> str(john)
'1-john'
>>> print(john)
1-john
```

## 5.2. Doux opérateurs

Les opérateurs sont un autre type de méthodes spéciales que nous découvrirons dans cette section.

En effet, les opérateurs ne sont rien d'autres en Python que des fonctions, qui s'appliquent sur leurs opérandes. On peut s'en rendre compte à l'aide du module `operator`, qui répertorie les fonctions associées à chaque opérateur.

```
>>> import operator
>>> operator.add(5, 6)
11
>>> operator.mul(2, 3)
6
```

Ainsi, chacun des opérateurs correspondra à une méthode de l'opérande de gauche, qui recevra en paramètre l'opérande de droite.

### 5.2.1. Opérateurs arithmétiques

L'addition, par exemple, est définie par la méthode `__add__`.

```
>>> class A:
...     def __add__(self, other):
...         return other # on considère self comme 0
...
>>> A() + 5
5
```

Assez simple, n'est-il pas ? Mais nous n'avons pas tout à fait terminé. Si la méthode est appelée sur l'opérande de gauche, que se passe-t-il quand notre objet se trouve à droite ?

```
>>> 5 + A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'A'
```

Nous ne supportons pas cette opération. En effet, l'expression fait appel à la méthode `int.__add__` qui ne connaît pas les objets de type `A`. Heureusement, ce cas a été prévu et il existe une fonction inverse, `__radd__`, appelée si la première opération n'était pas supportée.

```
>>> class A:
...     def __add__(self, other):
...         return other
...     def __radd__(self, other):
...         return other
```

```
...
>>> A() + 5
5
>>> 5 + A()
5
```

Il faut bien noter que `A.__radd__` ne sera appelée que si `int.__add__` a échoué.

Les autres opérateurs arithmétques binaires auront un comportement similaire, voici une liste des méthodes à implémenter pour chacun d'eux :

— **Addition/Concaténation** (`a + b`) — `__add__`, `__radd__`
— **Soustraction/Différence** (`a - b`) — `__sub__`, `__rsub__`
— **Multiplication** (`a * b`) — `__mul__`, `__rmul__`
— **Division** (`a / b`) — `__truediv__`, `__rtruediv__`
— **Division entière** (`a // b`) — `__floordiv__`, `__rfloordiv__`
— **Modulo/Formattage** (`a % b`) — `__mod__`, `__rmod__`
— **Exponentiation** (`a ** b`) — `__pow__`, `__rpow__`

On remarque aussi que chacun de ces opérateurs arithmétiques possède une version simplifiée pour l'assignation (`a += b`) qui correspond à la méthode `__iadd__`. Par défaut, les méthodes `__add__`/`__radd__` sont appelées, mais définir `__iadd__` permet d'avoir un comportement différent dans le cas d'un opérateur d'assignation, par exemple sur les listes :

```
>>> l = [1, 2, 3]
>>> l2 = l
>>> l2 = l2 + [4]
>>> l2
[1, 2, 3, 4]
>>> l
[1, 2, 3]
>>> l2 = l
>>> l2 += [4]
>>> l2
[1, 2, 3, 4]
>>> l
[1, 2, 3, 4]
```

### 5.2.2. Opérateurs arithmétiques unaires

Voici pour les opérateurs binaires, voyons maintenant les opérateurs unaires, qui ne prennent donc pas d'autre paramètre que `self`.

— **Opposé** (`-a`) — `__neg__`
— **Positif** (`+a`) - `__pos__`
— **Valeur abosule** (`abs(a)`) — `__abs__`

### 5.2.3. Opérateurs de comparaison

De la même manière que pour les opérateurs arithmétiques, nous avons une méthode spéciale par opérateur de comparaison. Ces opérateurs s'appliqueront sur l'opérande gauche en recevant le droit en paramètre. Ils devront retourner un booléen.

Contrairement aux opérateurs arithmétiques, il n'est pas nécessaire d'avoir deux versions pour chaque opérateur puisque Python saura directement quelle opération inverse tester si la première a échoué (`a == b` est équivalent à `b == a`, `a < b` à `b > a`, etc.).

— **Égalité** (`a == b`) — `__eq__`
— **Différence** (`a != b`) — `__neq__`
— **Stricte infériorité** (`a < b`) — `__lt__`
— **Infériorité** (`a <= b`) — `__le__`
— **Stricte supériorité** (`a > b`) — `__gt__`
— **Supériorité** (`a >= b`) — `__ge__`

On notera aussi que beaucoup de ces opérateurs peuvent s'inférer les uns les autres. Par exemple, il suffit de savoir calculer `a == b` et `a < b` pour définir toutes les autres opérations. Ainsi, Python dispose d'un décorateur, `total_ordering` du module `functools`, pour automatiquement générer les opérations manquantes.

```
>>> from functools import total_ordering
>>> @total_ordering
... class Inferior:
...     def __eq__(self, other):
...         return False
...     def __lt__(self, other):
...         return True
...
>>> i = Inferior()
>>> i == 5
False
>>> i > 5
False
>>> i < 5
True
>>> i <= 5
True
>>> i != 5
True
```

### 5.2.4. Autres opérateurs

Nous avons ici étudié les principaux opérateurs du langage. Ces listes ne sont pas exhaustives et présentent juste la méthodologie à suivre.

Pour une liste complète, je vous invite à consulter la documentation du module `operator` : https ://docs.python.org/3/library/operator.html .

## 5.3. TP : Arithmétique simple

Oublions temporairement nos utilisateurs et notre forum, et intéressons-nous à l'évaluation mathématique.

Imaginons que nous voulions représenter une expression mathématique, qui pourrait contenir des termes variables (par exemple, `2 * (-x + 1)`).

Il va nous falloir utiliser un type pour représenter cette variable `x`, appelé `Var`, et un second pour l'expression non évaluée, `Expr`. Les `Var` étant un type particulier d'expressions.

Nous aurons deux autres types d'expressions : les opérations arithmétiques unaires (`+`, `-`) et binaires (`+`, `-`, `*`, `/`, `//`, `%`, `**`). Vous pouvez vous appuyer un même type pour ces deux types d'opérations.

L'expression précédente s'évaluerait par exemple à :

```
BinOp(operator.mul, 2, BinOp(operator.add, UnOp(operator.neg,
    Var('x')), 1))
```

Nous ajouterons à notre type `Expr` une méthode `compute(**values)`, qui permettra de calculer l'expression suivant une valeur donnée, de façon à ce que `Var('x').compute(x=5)` retourne `5`.

Enfin, nous pourrons ajouter une méthode `__repr__` pour obtenir une représentation lisible de notre expression.

```
import operator

def compute(expr, **values):
    if not isinstance(expr, Expr):
        return expr
    return expr.compute(**values)

class Expr:
    def compute(self, **values):
        raise NotImplementedError

    def __pos__(self):
        return UnOp(operator.pos, self, '+')

    def __neg__(self):
        return UnOp(operator.neg, self, '-')

```

```
18      def __add__(self, rhs):
19          return BinOp(operator.add, self, rhs, '+')
20
21      def __radd__(self, lhs):
22          return BinOp(operator.add, lhs, self, '+')
23
24      def __sub__(self, rhs):
25          return BinOp(operator.sub, self, rhs, '-')
26
27      def __rsub__(self, lhs):
28          return BinOp(operator.sub, lhs, self, '-')
29
30      def __mul__(self, rhs):
31          return BinOp(operator.mul, self, rhs, '*')
32
33      def __rmul__(self, lhs):
34          return BinOp(operator.mul, lhs, self, '*')
35
36      def __truediv__(self, rhs):
37          return BinOp(operator.truediv, self, rhs, '/')
38
39      def __rtruediv__(self, lhs):
40          return BinOp(operator.truediv, lhs, self, '/')
41
42      def __floordiv__(self, rhs):
43          return BinOp(operator.floordiv, self, rhs, '//')
44
45      def __rfloordiv__(self, lhs):
46          return BinOp(operator.floordiv, lhs, self, '//')
47
48      def __mod__(self, rhs):
49          return BinOp(operator.mod, self, rhs, '*')
50
51      def __rmod__(self, lhs):
52          return BinOp(operator.mod, lhs, self, '*')
53
54  class Var(Expr):
55      def __init__(self, name):
56          self.name = name
57
58      def compute(self, **values):
59          if self.name in values:
60              return values[self.name]
61          return self
62
63      def __repr__(self):
64          return self.name
65
66  class Op(Expr):
67      def __init__(self, op, *args):
```

```
68          self.op = op
69          self.args = args
70
71      def compute(self, **values):
72          args = [compute(arg, **values) for arg in self.args]
73          return self.op(*args)
74
75  class UnOp(Op):
76      def __init__(self, op, expr, symbol=None):
77          super().__init__(op, expr)
78          self.symbol = symbol
79
80      def __repr__(self):
81          if self.symbol is None:
82              return super().__repr__()
83          return '{}{!r}'.format(self.symbol, self.args[0])
84
85  class BinOp(Op):
86      def __init__(self, op, expr1, expr2, symbol=None):
87          super().__init__(op, expr1, expr2)
88          self.symbol = symbol
89
90      def __repr__(self):
91          if self.symbol is None:
92              return super().__repr__()
93          return '({!r} {} {!r})'.format(self.args[0], self.symbol,
                self.args[1])
94
95  if __name__ == '__main__':
96      x = Var('x')
97      expr = 2 * (-x + 1)
98      print(expr)
99      print(compute(expr, x=1))
100
101     y = Var('y')
102     expr += y
103     print(compute(expr, x=0, y=10))
```

---

Les opérateurs sont une notion importante en Python, mais ils sont loin d'être la seule. Le chapitre suivant vous présentera d'autres concepts avancés du Python, qu'il est important de connaître, pour être en mesure de les utiliser quand cela s'avère nécessaire.

# 6. Programmation orientée objet avancée

Dans ce chapitre, nous allons nous intéresser à des concepts plus avancés de programmation orientée objet disponibles en Python, tels que les attributs/méthodes de classe ou les propriétés.

## 6.1. Les attributs entrent en classe

Nous avons déjà rencontré un attribut de classe, quand nous nous intéressions aux parents d'une classe. Souvenez-vous de `__bases__`, nous ne l'utilisions pas sur des instances mais sur notre classe directement.

En Python, les classes sont des objets comme les autres, et peuvent donc posséder leurs propres attributs.

```
>>> class User:
...     pass
...
>>> User.type = 'simple_user'
>>> User.type
'simple_user'
```

Les attributs de classe peuvent aussi se définir dans le corps de la classe, de la même manière que les méthodes.

```
class User:
    type = 'simple_user'
```

On notera à l'inverse qu'il est aussi possible de définir une méthode de la classe depuis l'extérieur :

```
>>> def User_repr(self):
...     return '<User>'
...
>>> User.__repr__ = User_repr
>>> User()
<User>
```

L'avantage des attributs de classe, c'est qu'ils sont aussi disponibles pour les instances de cette classe. Ils sont partagés par toutes les instances.

```
>>> john = User()
>>> john.type
'simple_user'
>>> User.type = 'admin'
>>> john.type
'admin'
```

C'est le fonctionnement du *MRO* de Python, il cherche d'abord si l'attribut existe dans l'objet, puis si ce n'est pas le cas, le cherche dans les classes parentes.

Attention donc, quand l'attribut est redéfini dans l'objet, il sera trouvé en premier, et n'affectera pas la classe.

```
>>> john = User()
>>> john.type
'admin'
>>> john.type = 'superadmin'
>>> john.type
'superadmin'
>>> User.type
'admin'
>>> joe = User()
>>> joe.type
'admin'
```

Attention aussi, quand l'attribut de classe est un objet mutable[4], il peut être modifié par n'importe quelle instance de la classe.

```
>>> class User:
...     users = []
...
>>> john, joe = User(), User()
>>> john.users.append(john)
>>> joe.users.append(joe)
>>> john.users
[<__main__.User object at 0x7f3b7acf8b70>, <__main__.User object at
    0x7f3b7acf8ba8>]
```

L'attribut de classe est aussi conservé lors de l'héritage, et partagé avec les classes filles (sauf lorsque les classes filles redéfinissent l'attribut, de la même manière que pour les instances).

4. Un objet mutable est un objet que l'on peut modifier (liste, dictionnaire) par opposition à un objet immutable (nombre, chaîne de caractères, *tuple*).

```
>>> class Guest(User):
...     pass
...
>>> Guest.users
[<__main__.User object at 0x7f3b7acf8b70>, <__main__.User object at
    0x7f3b7acf8ba8>]
>>> class Admin(User):
...     users = []
...
>>> Admin.users
[]
```

## 6.2. La méthode pour avoir la classe

Comme pour les attributs, des méthodes peuvent être définies au niveau de la classe. C'est par exemple le cas de la méthode `mro`.

```
int.mro()
```

Les méthodes de classe constituent des opérations relatives à la classe mais à aucune instance. Elles recevront la classe courante en premier paramètre (nommé `cls`, correspondant au `self` des méthodes d'instance), et auront donc accès aux autres attributs et méthodes de classe.

Reprenons notre classe `User`, à laquelle nous voudrions ajouter le stockage de tous les utilisateurs, et la génération automatique de l'`id`. Il nous suffirait d'une même méthode de classe pour stocker l'utilisateur dans un attribut de classe `users`, et qui lui attribuerait un `id` en fonction du nombre d'utilisateurs déjà enregistrés.

```
>>> root = Admin('root', 'toor')
>>> root
<User: 1, root>
>>> User('john', '12345')
<User: 2, john>
>>> guest = Guest('guest')
<User: 3, guest>
```

Les méthodes de classe se définissent comme les méthodes habituelles, à la différence près qu'elles sont précédées du décorateur `classmethod`.

```python
import crypt

class User:
    users = []

    def __init__(self, name, password):
        self.name = name
        self._salt = crypt.mksalt()
        self._password = self._crypt_pwd(password)
        self.register(self)

    @classmethod
    def register(cls, user):
        cls.users.append(user)
        user.id = len(cls.users)

    def _crypt_pwd(self, password):
        return crypt.crypt(password, self._salt)

    def check_pwd(self, password):
        return self._password == self._crypt_pwd(password)

    def __repr__(self):
        return '<User: {}, {}>'.format(self.id, self.name)

class Guest(User):
    def __init__(self, name):
        super().__init__(name, '')

    def check_pwd(self, password):
        return True

class Admin(User):
    def manage(self):
        print('I am an über administrator!')
```

Vous pouvez constater le résultat en réessayant le code donné plus haut.

## 6.3. Le statique c'est fantastique

Les méthodes statiques sont très proches des méthodes de classe, mais sont plus à considérer comme des fonctions au sein d'une classe.

Contrairement aux méthodes de classe, elles ne recevront pas le paramètre `cls`, et n'auront donc pas accès aux attributs de classe, méthodes de classe ou méthodes statiques.

Les méthodes statiques sont plutôt dédiées à des comportements annexes en rapport avec la classe, par exemple on pourrait remplacer notre attribut `id` par un *uuid* aléatoire, dont la génération ne dépendrait de rien d'autre dans la classe.

Elles se définissent avec le décorateur `staticmethod`.

```
import uuid

class User:
    def __init__(self, name, password):
        self.id = self._gen_uuid()
        self.name = name
        self._salt = crypt.mksalt()
        self._password = self._crypt_pwd(password)

    @staticmethod
    def _gen_uuid():
        return str(uuid.uuid4())

    def _crypt_pwd(self, password):
        return crypt.crypt(password, self._salt)

    def check_pwd(self, password):
        return self._password == self._crypt_pwd(password)
```

```
>>> john = User('john', '12345')
>>> john.id
'69ef1327-3d96-42a9-94e6-622619fbf666'
```

## 6.4. Attribut es-tu là ?

Nous savons récupérer et assigner un attribut dont le nom est fixé, cela se fait facilement à l'aide des instructions `obj.foo` et `obj.foo = value`.

Mais nous est-il possible d'accéder à des attributs dont le nom est variable ?

Prenons une instance `john` de notre classe `User`, et le nom d'un attribut que nous voudrions extraire :

```
>>> john = User('john', '12345')
>>> attr = 'name'
```

La fonction `getattr` nous permet alors de récupérer cet attribut.

```
>>> getattr(john, attr)
'john'
```

Ainsi, `getattr(obj, 'foo')` est équivalent à `obj.foo`.

On trouve aussi une fonction `hasattr` pour tester la présence d'un attribut dans un objet. Elle est construite comme `getattr` mais retourne un booléen pour savoir si l'attribut est présent ou non.

```
>>> hasattr(john, 'name')
True
>>> hasattr(john, 'last_name')
False
>>> hasattr(john, 'id')
True
```

De la même manière, les fonctions `setattr` et `delattr` servent respectivement à modifier et supprimer un attribut.

```
>>> setattr(john, 'name', 'peter') # équivalent à `john.name =
    'peter'`
>>> john.name
'peter'
>>> delattr(john, 'name') # équivalent à `del john.name`
>>> john.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'User' object has no attribute 'name'
```

## 6.5. La dynamique des propriétés

Les propriétés sont une manière en Python de « dynamiser » les attributs d'un objet. Ils permettent de générer des attributs à la volée à partir de méthodes de l'objet.

Un exemple valant mieux qu'un long discours :

```
class ProfilePicture:
    @property
    def picture(self):
        return '{}-{}.png'.format(self.id, self.name)

```

```
6 class UserPicture(ProfilePicture, User):
7     pass
```

On définit donc une propriété `picture`, qui s'utilise comme un attribut. Chaque fois qu'on appelle `picture`, la méthode correspondante est appelée et le résultat est calculé.

```
1 >>> john = UserPicture('john', '12345')
2 >>> john.picture
3 '1-john.png'
4 >>> john.name = 'John'
5 >>> john.picture
6 '1-John.png'
```

Il s'agit là d'une propriété en lecture seule, il nous est en effet impossible de modifier la valeur de l'attribut `picture`.

```
1 >>> john.picture = 'toto.png'
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 AttributeError: can't set attribute
```

Pour le rendre modifiable, il faut ajouter à notre classe la méthode permettant de gérer la modification, à l'aide du décorateur `@picture.setter` (le décorateur `setter` de notre propriété `picture`, donc).

On utilisera ici un attribut `_picture`, qui pourra contenir l'adresse de l'image si elle a été définie, et `None` le cas échéant.

```
1  class ProfilePicture:
2      def __init__(self, *args, **kwargs):
3          super().__init__(*args, **kwargs)
4          self._picture = None
5
6      @property
7      def picture(self):
8          if self._picture is not None:
9              return self._picture
10         return '{}-{}.png'.format(self.id, self.name)
11
12     @picture.setter
13     def picture(self, value):
14         self._picture = value
15
16 class UserPicture(ProfilePicture, User):
```

```
    pass
```

```
>>> john = UserPicture('john', '12345')
>>> john.picture
'1-john.png'
>>> john.picture = 'toto.png'
>>> john.picture
'toto.png'
```

Enfin, on peut aussi coder la suppression de l'attribut à l'aide de `@picture.deleter`, ce qui revient à réaffecter `None` à l'attribut `_picture`.

```
class ProfilePicture:
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._picture = None

    @property
    def picture(self):
        if self._picture is not None:
            return self._picture
        return '{}-{}.png'.format(self.id, self.name)

    @picture.setter
    def picture(self, value):
        self._picture = value

    @picture.deleter
    def picture(self):
        self._picture = None

class UserPicture(ProfilePicture, User):
    pass
```

```
>>> john = UserPicture('john', '12345')
>>> john.picture
'1-john.png'
>>> john.picture = 'toto.png'
>>> john.picture
'toto.png'
>>> del john.picture
>>> john.picture
'1-john.png'
```

## 6.6. L'art des classes abstraites

La notion de classes abstraites est utilisée lors de l'héritage pour forcer les classes filles à implémenter certaines méthodes (dites méthodes abstraites) et donc respecter une interface.

Les classes abstraites ne font pas partie du cœur même de Python, mais sont disponibles via un module de la bibliothèque standard, `abc` (*Abstract Base Classes*). Ce module contient notamment la classe `ABC` et le décorateur `abstractmethod`, pour définir respectivement une classe abstraite et une méthode abstraite de cette classe.

Une classe abstraite doit donc hériter d'`ABC`, et utiliser le décorateur cité pour définir ses méthodes abstraites.

```
1 import abc
2
3 class MyABC(abc.ABC):
4     @abc.abstractmethod
5     def foo(self):
6         pass
```

Il nous est impossible d'instancier des objets de type `MyABC`, puisqu'une méthode abstraite n'est pas implémentée :

```
1 >>> MyABC()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError:
      Can't instantiate abstract class MyABC with abstract methods foo
```

Il en est de même pour une classe héritant de `MyABC` sans redéfinir la méthode.

```
1 >>> class A(MyABC):
2 ...     pass
3 ...
4 >>> A()
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError:
      Can't instantiate abstract class A with abstract methods foo
```

Aucun problème par contre avec une autre classe qui redéfinit bien la méthode.

```
>>> class B(MyABC):
...     def foo(self):
...         return 7
...
>>> B()
<__main__.B object at 0x7f33065316a0>
>>> B().foo()
7
```

## 6.7. TP : Base de données

Pour ce dernier TP, nous aborderons les méthodes de classe et les propriétés.

Reprenons notre forum, auquel nous souhaiterions ajouter la gestion d'une base de données.

Notre base de données sera une classe avec deux méthodes, `insert` et `select`. Son implémentation est libre, elle doit juste respecter l'interface suivante :

```
>>> class A: pass
...
>>> class B: pass
...
>>>
>>> db = Database()
>>> obj = A()
>>> obj.value = 42
>>> db.insert(obj)
>>> obj = A()
>>> obj.value = 5
>>> db.insert(obj)
>>> obj = B()
>>> obj.value = 42
>>> obj.name = 'foo'
>>> db.insert(obj)
>>>
>>> db.select(A)
<__main__.A object at 0x7f033697f358>
>>> db.select(A, value=5)
<__main__.A object at 0x7f033697f3c8>
>>> db.select(B, value=42)
<__main__.B object at 0x7f033697f438>
>>> db.select(B, value=42, name='foo')
<__main__.B object at 0x7f033697f438>
>>> db.select(B, value=5)
ValueError: item not found
```

Nous ajouterons ensuite une classe `Model`, qui se chargera de stocker dans la base toutes les instances créées. `Model` comprendra une méthode de classe `get(**kwargs)` chargée de réaliser une requête `select` sur la base de données et de retourner l'objet correspondant. Les objets de type `Model` disposeront aussi d'une propriété `id`, retournant un identifiant unique de l'objet.

On pourra alors faire hériter nos classes `User` et `Post` de `Model`, afin que les utilisateurs et messages soient stockés en base de données. Dans un second temps, on pourra faire de `Model` une classe abstraite, par exemple en rendant abstraite la méthode `__init__`.

```
import abc
import datetime

class Database:
    data = []

    def insert(self, obj):
        self.data.append(obj)

    def select(self, cls, **kwargs):
        items = (item for item in self.data
                 if isinstance(item, cls)
                 and all(hasattr(item, k) and getattr(item, k) == v
                         for (k, v) in kwargs.items()))
        try:
            return next(items)
        except StopIteration:
            raise ValueError('item not found')

class Model(abc.ABC):
    db = Database()
    @abc.abstractmethod
    def __init__(self):
        self.db.insert(self)
    @classmethod
    def get(cls, **kwargs):
        return cls.db.select(cls, **kwargs)
    @property
    def id(self):
        return id(self)

class User(Model):
    def __init__(self, name):
        super().__init__()
        self.name = name

class Post(Model):
    def __init__(self, author, message):
        super().__init__()
        self.author = author
```

```
41          self.message = message
42          self.date = datetime.datetime.now()
43
44      def format(self):
45          date = self.date.strftime('le %d/%m/%Y à %H:%M:%S')
46          return
                '<div><span>Par {} {}</span><p>{}</p></div>'.format(self.author.nam
                date, self.message)
47
48  if __name__ == '__main__':
49      john = User('john')
50      peter = User('peter')
51      Post(john, 'salut')
52      Post(peter, 'coucou')
53
54      print(Post.get(author=User.get(name='peter')).format())
55      print(Post.get(author=User.get(id=john.id)).format())
```

---

Ces dernières notions ont dû compléter vos connaissances du modèle objet de Python, et vous devriez maintenant être prêts à vous lancer dans un projet exploitant ces concepts.

# 7. Conclusion

Ce cours touche maintenant à sa fin, mais votre apprentissage du Python continue. Avec la programmation objet, un nouveau monde s'offre à vous.

Vous allez pouvoir prendre en mains des *frameworks* tels que Django si le développement Web vous intéresse. Ou encore des bibliothèques comme PyGTK si vous êtes plutôt attirés par la programmation de *GUI*.

Enfin, si vous voulez compléter votre compréhension du modèle objet de Python, je peux vous orienter vers ce cours sur les notions avancées du langage .