**Première approche des classes**

[Connectez-vous](https://openclassrooms.com/fr/login) ou [inscrivez-vous](https://openclassrooms.com/fr/register) gratuitement pour bénéficier de toutes les fonctionnalités de ce cours !

Dans ce chapitre, sans plus attendre, nous allons créer nos premières classes, nos premiers attributs et nos premières méthodes. Nous allons aussi essayer de comprendre les mécanismes de la programmation orientée objet en Python.

Au-delà du mécanisme, l'orienté objet est une véritable philosophie et Python est assez différent des autres langages, en termes de philosophie justement. Restez concentrés, ce langage n'a pas fini de vous étonner !

**Les classes, tout un monde**

Dans la partie précédente, j'avais brièvement décrit les objets comme des variables pouvant contenir elles-mêmes des fonctions et variables. Nous sommes allés plus loin tout au long de la seconde partie, pour découvrir que nos « fonctions contenues dans nos objets » sont appelées des méthodes. En vérité, je me suis cantonné à une définition « pratique » des objets, alors que derrière la POO (Programmation Orientée Objet) se cache une véritable philosophie.

**Pourquoi utiliser des objets ?**

Les premiers langages de programmation n'incluaient pas l'orienté objet. Le langage C, pour ne citer que lui, n'utilise pas ce concept et il aura fallu attendre le C++ pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du C.

Java, un langage apparu à peu près en même temps que Python, définit une philosophie assez différente de celle du C++ : contrairement à ce dernier, le Java exige que tout soit rangé dans des classes. Même l'application standardHello Worldest contenue dans une classe.

En Python, la liberté est plus grande. Après tout, vous avez pu passer une partie de ce tutoriel sans connaître la façade objet de Python. Et pourtant, le langage Python est totalement orienté objet : *en Python, tout est objet*, vous n'avez pas oublié ? Quand vous croyez utiliser une simple variable, un module, une fonction…, ce sont des objets qui se cachent derrière.

Loin de moi l'idée de faire un comparatif entre différents langages. Ce sur quoi je souhaite attirer votre attention, c'est que plusieurs langages intègrent l'orienté objet, chacun avec une philosophie distincte. Autrement dit, si vous avez appris l'orienté objet dans un autre langage, tel que le C++ ou le Java, ne tenez pas pour acquis que vous allez retrouver les même mécanismes et surtout, la même philosophie. Gardez autant que possible l'esprit dégagé de tout préjugé sur la philosophie objet de Python.

Pour l'instant, nous n'avons donc vu qu'un aspect technique de l'objet. J'irais jusqu'à dire que ce qu'on a vu jusqu'ici, ce n'était qu'une façon « un peu plus esthétique » de coder : il est plus simple et plus compréhensible d'écrirema\_liste.append(5)queappend\_to\_list(ma\_liste, 5). Mais derrière la POO, il n'y a pas qu'un souci esthétique, loin de là.

**Choix du modèle**

Bon, comme vous vous en souvenez sûrement (du moins, je l'espère), une classe est un peu un modèle suivant lequel on va créer des objets. C'est dans la classe que nous allons définir nos méthodes et attributs, les attributs étant des variables contenues dans notre objet.

Mais qu'allons-nous modéliser ? L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter des données un peu plus complexes qu'un simple nombre, ou qu'une chaîne de caractères. Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie. Mais on serait bien limité si on ne pouvait faire ses propres classes.

Pour l'instant, nous allons modéliser… une personne. C'est le premier exemple qui me soit venu à l'esprit, nous verrons bien d'autres exemples avant la fin de la partie.

**Convention de nommage**

Loin de moi l'idée de compliquer l'exercice mais si on se réfère à [la PEP 8 de Python](https://www.python.org/dev/peps/pep-0020/), il est préférable d'utiliser pour des noms de classes la convention dite **Camel Case**.

Les PEP sont les « Python Enhancement Proposals », c'est à dire les propositions d'amélioration de Python.

Cette convention n'utilise pas le signe souligné \_ pour séparer les mots. Le principe consiste à mettre en majuscule chaque lettre débutant un mot, par exemple :MaClasse.

C'est donc cette convention que je vais utiliser pour les noms de classes. Libre à vous d'en changer, encore une fois rien n'est imposé.

Pour définir une nouvelle classe, on utilise le mot-cléclass.

Sa syntaxe est assez intuitive :class NomDeLaClasse:.

N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.

Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne ? Beaucoup de choses, vous en conviendrez. On ne va en retenir que quelques-unes : le nom, le prénom, l'âge, le lieu de résidence… allez, cela suffira.

Cela nous fait donc quatre attributs. Ce sont les variables internes à notre objet, qui vont le caractériser. Une personne telle que nous la modélisons sera caractérisée par son nom, son prénom, son âge et son lieu de résidence.

Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe. Voyons cela de plus près.

**Nos premiers attributs**

Nous avons défini les attributs qui allaient caractériser notre objet de classePersonne. Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un **constructeur**, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.

Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.

Voyons le code, ce sera plus parlant :

class Personne: # Définition de notre classe Personne

"""Classe définissant une personne caractérisée par :

- son nom

- son prénom

- son âge

- son lieu de résidence"""

def \_\_init\_\_(self): # Notre méthode constructeur

"""Pour l'instant, on ne va définir qu'un seul attribut"""

self.nom = "Dupont"

Voyons en détail :

* D'abord, la définition de la classe. Elle est constituée du mot-cléclass, du nom de la classe et des deux points rituels « : ».
* Unedocstringcommentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous vous lancerez dans de grands projets, notamment à plusieurs.
* La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom\_\_init\_\_, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés(\_\_nommethode\_\_)sont des **méthodes spéciales**. Notez que, dans notre définition de méthode, nous passons un premier paramètre nomméself.
* Une nouvelledocstring. Je ne complique pas inutilement, je précise donc qu'on va simplement définir un seul attribut pour l'instant dans notre constructeur.
* Dans notre constructeur, nous trouvons l'instanciation de notre attributnom. On crée une variableself.nomet on lui donne comme valeurDupont. Je vais détailler un peu plus bas ce qui se passe ici.

Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

>>> bernard = Personne()

>>> bernard

<\_\_main\_\_.Personne object at 0x00B42570>

>>> bernard.nom

'Dupont'

>>>

Quand on demande à l'interpréteur d'afficher directement notre objetbernard, il nous sort quelque chose d'un peu imbuvable… Bon, l'essentiel est la mention précisant la classe dont l'objet est issu. On peut donc vérifier que c'est bien notre classePersonnedont est issu notre objet. On essaye ensuite d'afficher l'attributnomde notre objetbernardet on obtient'Dupont'(la valeur définie dans notre constructeur). Notez qu'on utilise le point (.), encore et toujours utilisé pour une relation d'appartenance (nomest un attribut de l'objetbernard). Encore un peu d'explications :

**Quand on crée notre objet…**

Quand on tapePersonne(), on appelle le constructeur de notre classePersonne, d'une façon quelque peu indirecte que je ne détaillerai pas ici. Celui-ci prend en paramètre une variable un peu mystérieuse :self. En fait, il s'agit tout bêtement de notre objet en train de se créer. On écrit dans cet objet l'attributnomle plus simplement du monde :self.nom = "Dupont". À la fin de l'appel au constructeur, Python renvoie notre objetselfmodifié, avec notre attribut. On va réceptionner le tout dans notre variablebernard.

Si ce n'est pas très clair, pas de panique ! Vous pouvez vous contenter de vous familiariser avec la syntaxe du constructeur Python, qui sera souvent la même, et laisser l'aspect un peu théorique de côté, pour plus tard. Nous aurons l'occasion d'y revenir avant la fin du chapitre.

**Étoffons un peu notre constructeur**

Bon, on avait dit quatre attributs, on n'en a fait qu'un. Et puis notre constructeur pourrait éviter de donner les mêmes valeurs par défaut à chaque fois, tout de même !

C'est juste. Dans un premier temps, on va se contenter de définir les autres attributs, le prénom, l'âge, le lieu de résidence. Essayez de le faire, normalement vous ne devriez éprouver aucune difficulté.

Voici le code, au cas où :

class Personne:

"""Classe définissant une personne caractérisée par :

- son nom

- son prénom

- son âge

- son lieu de résidence"""

def \_\_init\_\_(self): # Notre méthode constructeur

"""Constructeur de notre classe. Chaque attribut va être instancié

avec une valeur par défaut... original"""

self.nom = "Dupont"

self.prenom = "Jean" # Quelle originalité

self.age = 33 # Cela n'engage à rien

self.lieu\_residence = "Paris"

Cela vous paraît évident ? Encore un petit code d'exemple :

>>> jean = Personne()

>>> jean.nom

'Dupont'

>>> jean.prenom

'Jean'

>>> jean.age

33

>>> jean.lieu\_residence

'Paris'

>>> # Jean déménage…

... jean.lieu\_residence = "Berlin"

>>> jean.lieu\_residence

'Berlin'

>>>

Je sens un courant d'air… les habitués de l'objet, une minute.

Cet exemple me paraît assez clair, sur le principe de définition des attributs, accès aux attributs d'un objet créé, modification des attributs d'un objet.

Une toute petite explication en ce qui concerne la ligne 11 : dans beaucoup de tutoriels, on déconseille de modifier un attribut d'instance (un attribut d'un objet) comme on vient de le faire, en faisant simplementobjet.attribut = valeur. Si vous venez d'un autre langage, vous pourrez avoir entendu parler des accesseurs et mutateurs. Ces concepts sont repris dans certains tutoriels Python, mais ils n'ont pas précisément lieu d'être dans ce langage. Tout cela, je le détaillerai dans le prochain chapitre. Pour l'instant, il vous suffit de savoir que, quand vous voulez modifier un attribut d'un objet, vous écrivezobjet.attribut = nouvelle\_valeur. Nous verrons les cas particuliers plus loin.

Bon. Il nous reste encore à faire un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons… le nom et le prénom, pour commencer.

class Personne:

"""Classe définissant une personne caractérisée par :

- son nom

- son prénom

- son âge

- son lieu de résidence"""

def \_\_init\_\_(self, nom, prenom):

"""Constructeur de notre classe"""

self.nom = nom

self.prenom = prenom

self.age = 33

self.lieu\_residence = "Paris"

Et en images :

>>> bernard = Personne("Micado", "Bernard")

>>> bernard.nom

'Micado'

>>> bernard.prenom

'Bernard'

>>> bernard.age

33

>>>

N'oubliez pas que le premier paramètre doit êtreself. En dehors de cela, un constructeur est une fonction plutôt classique : vous pouvez définir des paramètres, par défaut ou non, nommés ou non. Quand vous voudrez créer votre objet, vous appellerez le nom de la classe en passant entre parenthèses les paramètres à utiliser. Faites quelques tests, avec plus ou moins de paramètres, je pense que vous saisirez très rapidement le principe.

**Attributs de classe**

Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils sont propres à l'objet : si vous créez plusieurs objets, les attributsnom,prenom,… de chacun ne seront pas forcément identiques d'un objet à l'autre. Mais on peut aussi définir des attributs dans notre classe. Voyons un exemple :

class Compteur:

"""Cette classe possède un attribut de classe qui s'incrémente à chaque

fois que l'on crée un objet de ce type"""

objets\_crees = 0 # Le compteur vaut 0 au départ

def \_\_init\_\_(self):

"""À chaque fois qu'on crée un objet, on incrémente le compteur"""

Compteur.objets\_crees += 1

On définit notre attribut de classe directement dans le corps de la classe, sous la définition et ladocstring, avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe. Et on y accède de cette façon également, en dehors de la classe. Voyez plutôt :

>>> Compteur.objets\_crees

0

>>> a = Compteur() # On crée un premier objet

>>> Compteur.objets\_crees

1

>>> b = Compteur()

>>> Compteur.objets\_crees

2

>>>

À chaque fois qu'on crée un objet de typeCompteur, l'attribut de classeobjets\_creess'incrémente de 1. Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques. Nous aurons l'occasion d'en reparler par la suite.

**Les méthodes, la recette**

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions, comme nous l'avons vu dans la partie précédente, agissant sur l'objet. Par exemple, la méthodeappendde la classelistpermet d'ajouter un élément dans l'objetlistmanipulé.

Pour créer nos premières méthodes, nous allons modéliser… un tableau. Un tableau noir, oui c'est très bien.

Notre tableau va posséder une surface (un attribut) sur laquelle on pourra écrire, que l'on pourra lire et effacer. Pour créer notre classeTableauNoiret notre attributsurface, vous ne devriez pas avoir de problème :

class TableauNoir:

"""Classe définissant une surface sur laquelle on peut écrire,

que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié

est 'surface'"""

def \_\_init\_\_(self):

"""Par défaut, notre surface est vide"""

self.surface = ""

Nous avons déjà créé une méthode, aussi vous ne devriez pas être trop surpris par la syntaxe que nous allons voir. Notre constructeur est en effet une méthode, elle en garde la syntaxe. Nous allons donc écrire notre méthodeecrirepour commencer.

class TableauNoir:

"""Classe définissant une surface sur laquelle on peut écrire,

que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié

est 'surface'"""

def \_\_init\_\_(self):

"""Par défaut, notre surface est vide"""

self.surface = ""

def ecrire(self, message\_a\_ecrire):

"""Méthode permettant d'écrire sur la surface du tableau.

Si la surface n'est pas vide, on saute une ligne avant de rajouter

le message à écrire"""

if self.surface != "":

self.surface += "\n"

self.surface += message\_a\_ecrire

Passons aux tests :

>>> tab = TableauNoir()

>>> tab.surface

''

>>> tab.ecrire("Coooool ! Ce sont les vacances !")

>>> tab.surface

"Coooool ! Ce sont les vacances !"

>>> tab.ecrire("Joyeux Noël !")

>>> tab.surface

"Coooool ! Ce sont les vacances !\nJoyeux Noël !"

>>> print(tab.surface)

Coooool ! Ce sont les vacances !

Joyeux Noël !

>>>

Notre méthodeecrirese charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.

On retrouve ici notre paramètreself. Il est temps de voir un peu plus en détail à quoi il sert.

**Le paramètreself**

Dans nos méthodes d'instance, qu'on appelle également des **méthodes d'objet**, on trouve dans la définition ce paramètreself. L'heure est venue de comprendre ce qu'il signifie.

Une chose qui a son importance : quand vous créez un nouvel objet, ici un tableau noir, les attributs de l'objet sont propres à l'objet créé. C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface. Donc les attributs sont contenus dans l'objet.

En revanche, les méthodes sont contenues dans la classe qui définit notre objet. C'est très important. Quand vous tapeztab.ecrire(…), Python va chercher la méthodeecrirenon pas dans l'objettab, mais dans la classeTableauNoir.

>>> tab.ecrire

<bound method TableauNoir.ecrire of <\_\_main\_\_.TableauNoir object at 0x00B3F3F0>>

>>> TableauNoir.ecrire

<function ecrire at 0x00BA5810>

>>> help(TableauNoir.ecrire)

Help on function ecrire in module \_\_main\_\_:

ecrire(self, message\_a\_ecrire)

Méthode permettant d'écrire sur la surface du tableau.

Si la surface n'est pas vide, on saute une ligne avant de rajouter

le message à écrire.

>>> TableauNoir.ecrire(tab, "essai")

>>> tab.surface

'essai'

>>>

Comme vous le voyez, quand vous tapeztab.ecrire(…), cela revient au même que si vous écrivezTableauNoir.ecrire(tab, …). Votre paramètreself, c'est l'objet qui appelle la méthode. C'est pour cette raison que vous modifiez la surface de l'objet en appelantself.surface.

Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer parself.

Le nomselfest une très forte convention de nommage. Je vous déconseille de changer ce nom. Certains programmeurs, qui trouvent qu'écrireselfà chaque fois est excessivement long, l'abrègent en une unique lettres. *Évitez ce raccourci*. De manière générale, évitez de changer le nom. Une méthode d'instance travaille avec le paramètreself.

N'est-ce pas effectivement plutôt long de devoir toujours travailler avecselfà chaque fois qu'on souhaite faire appel à l'objet ?

Cela peut le sembler, oui. C'est d'ailleurs l'un des reproches qu'on fait au langage Python. Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement. Mais c'est moins clair et cela peut susciter la confusion. En Python, dès qu'on voitself, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

Bon, voyons nos autres méthodes. Nous devons encore coderlirequi va se charger d'afficher notre surface eteffacerqui va effacer le contenu de notre surface. Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples. Sinon, n'hésitez pas à relire, jusqu'à ce que le déclic se fasse.

class TableauNoir:

"""Classe définissant une surface sur laquelle on peut écrire,

que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié

est 'surface'"""

def \_\_init\_\_(self):

"""Par défaut, notre surface est vide"""

self.surface = ""

def ecrire(self, message\_a\_ecrire):

"""Méthode permettant d'écrire sur la surface du tableau.

Si la surface n'est pas vide, on saute une ligne avant de rajouter

le message à écrire"""

if self.surface != "":

self.surface += "\n"

self.surface += message\_a\_ecrire

def lire(self):

"""Cette méthode se charge d'afficher, grâce à print,

la surface du tableau"""

print(self.surface)

def effacer(self):

"""Cette méthode permet d'effacer la surface du tableau"""

self.surface = ""

Et encore une fois, le code de test :

>>> tab = TableauNoir()

>>> tab.lire()

>>> tab.ecrire("Salut tout le monde.")

>>> tab.ecrire("La forme ?")

>>> tab.lire()

Salut tout le monde.

La forme ?

>>> tab.effacer()

>>> tab.lire()

>>>

Et voilà ! Avec nos méthodes bien documentées, un petit coup dehelp(TableauNoir)et vous obtenez une belle description de l'utilité de votre classe. C'est très pratique, n'oubliez pas les docstrings.

**Méthodes de classe et méthodes statiques**

Comme on trouve des attributs propres à la classe, on trouve aussi des méthodes de classe, qui ne travaillent pas sur l'instanceselfmais sur la classe même. C'est un peu plus rare mais cela peut être utile parfois. Notre méthode de classe se définit exactement comme une méthode d'instance, à la différence qu'elle ne prend pas en premier paramètreself(l'instance de l'objet) maiscls(la classe de l'objet).

En outre, on utilise ensuite une fonction *built-in* de Python pour lui faire comprendre qu'il s'agit d'une méthode de classe, pas d'une méthode d'instance.

class Compteur:

"""Cette classe possède un attribut de classe qui s'incrémente à chaque

fois que l'on crée un objet de ce type"""

objets\_crees = 0 # Le compteur vaut 0 au départ

def \_\_init\_\_(self):

"""À chaque fois qu'on crée un objet, on incrémente le compteur"""

Compteur.objets\_crees += 1

def combien(cls):

"""Méthode de classe affichant combien d'objets ont été créés"""

print("Jusqu'à présent, {} objets ont été créés.".format(

cls.objets\_crees))

combien = classmethod(combien)

Voyons d'abord le résultat :

>>> Compteur.combien()

Jusqu'à présent, 0 objets ont été créés.

>>> a = Compteur()

>>> Compteur.combien()

Jusqu'à présent, 1 objets ont été créés.

>>> b = Compteur()

>>> Compteur.combien()

Jusqu'à présent, 2 objets ont été créés.

>>>

Une méthode de classe prend en premier paramètre non passelfmaiscls. Ce paramètre contient la classe (iciCompteur).

Notez que vous pouvez appeler la méthode de classe depuis un objet instancié sur la classe. Vous auriez par exemple pu écrirea.combien().

Enfin, pour que Python reconnaisse une méthode de classe, il faut appeler la fonctionclassmethodqui prend en paramètre la méthode que l'on veut convertir et renvoie la méthode convertie.

Si vous êtes un peu perdus, retenez la syntaxe de l'exemple. La plupart du temps, vous définirez des méthodes d'instance comme nous l'avons vu plutôt que des méthodes de classe.

On peut également définir des méthodes statiques. Elles sont assez proches des méthodes de classe sauf qu'elles ne prennent aucun premier paramètre, niselfnicls. Elles travaillent donc indépendamment de toute donnée, aussi bien contenue dans l'instance de l'objet que dans la classe.

Voici la syntaxe permettant de créer une méthode statique. Je ne veux pas vous surcharger d'informations et je vous laisse faire vos propres tests si cela vous intéresse :

class Test:

"""Une classe de test tout simplement"""

def afficher():

"""Fonction chargée d'afficher quelque chose"""

print("On affiche la même chose.")

print("peu importe les données de l'objet ou de la classe.")

afficher = staticmethod(afficher)

Si vous vous emmêlez un peu avec les attributs et méthodes de classe, ce n'est pas bien grave. Retenez surtout les attributs et méthodes d'instance, c'est essentiellement sur ceux-ci que je me suis attardé et c'est ceux que vous retrouverez la plupart du temps.

Rappel : les noms de méthodes encadrés par deux soulignés de part et d'autre sont des **méthodes spéciales**. Ne nommez pas vos méthodes ainsi. Nous découvrirons plus tard ces méthodes particulières. Exemple de nom de méthode à éviter :\_\_mamethode\_\_.

**Un peu d'introspection**

Encore de la philosophie ?

Eh bien… le terme d'**introspection**, je le reconnais, fait penser à quelque chose de plutôt abstrait. Pourtant, vous allez très vite comprendre l'idée qui se cache derrière : Python propose plusieurs techniques pour explorer un objet, connaître ses méthodes ou attributs.

Quel est l'intérêt ? Quand on développe une classe, on sait généralement ce qu'il y a dedans, non ?

En effet. L'utilité, à notre niveau, ne saute pas encore aux yeux. Et c'est pour cela que je ne vais pas trop m'attarder dessus. Si vous ne voyez pas l'intérêt, contentez-vous de garder dans un coin de votre tête les deux techniques que nous allons voir. Arrivera un jour où vous en aurez besoin ! Pour l'heure donc, voyons plutôt l'effet :

**La fonctiondir**

La première technique d'introspection que nous allons voir est la fonctiondir. Elle prend en paramètre un objet et renvoie la liste de ses attributs et méthodes.

class Test:

"""Une classe de test tout simplement"""

def \_\_init\_\_(self):

"""On définit dans le constructeur un unique attribut"""

self.mon\_attribut = "ok"

def afficher\_attribut(self):

"""Méthode affichant l'attribut 'mon\_attribut'"""

print("Mon attribut est {0}.".format(self.mon\_attribut))

>>> # Créons un objet de la classe Test

... un\_test = Test()

>>> un\_test.afficher\_attribut()

Mon attribut est ok.

>>> dir(un\_test)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_g

e\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_',

'\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_

\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'affich

er\_attribut', 'mon\_attribut']

>>>

La fonctiondirrenvoie une liste comprenant le nom des attributs et méthodes de l'objet qu'on lui passe en paramètre. Vous pouvez remarquer que tout est mélangé, c'est normal : pour Python, les méthodes, les fonctions, les classes, les modules sont des objets. Ce qui différencie en premier lieu une variable d'une fonction, c'est qu'une fonction est exécutable (*callable*). La fonctiondirse contente de renvoyer tout ce qu'il y a dans l'objet, sans distinction.

Euh, c'est quoi tout cela ? On n'a jamais défini toutes ces méthodes ou attributs !

Non, en effet. Nous verrons plus loin qu'il s'agit de **méthodes spéciales** utiles à Python.

**L'attribut spécial\_\_dict\_\_**

Par défaut, quand vous développez une classe, tous les objets construits depuis cette classe posséderont un attribut spécial\_\_dict\_\_. Cet attribut est un dictionnaire qui contient en guise de clés les noms des attributs et, en tant que valeurs, les valeurs des attributs.

Voyez plutôt :

>>> un\_test = Test()

>>> un\_test.\_\_dict\_\_

{'mon\_attribut': 'ok'}

>>>

Pourquoi « attribut spécial » ?

C'est un attribut un peu particulier car ce n'est pas vous qui le créez, c'est Python. Il est entouré de deux signes soulignés\_\_de part et d'autre, ce qui traduit qu'il a une signification pour Python et n'est pas un attribut « standard ». Vous verrez plus loin dans ce cours des **méthodes spéciales** qui reprennent la même syntaxe.

Peut-on modifier ce dictionnaire ?

Vous le pouvez. Sachez qu'en modifiant la valeur de l'attribut, vous modifiez aussi l'attribut dans l'objet.

>>> un\_test.\_\_dict\_\_["mon\_attribut"] = "plus ok"

>>> un\_test.afficher\_attribut()

Mon attribut est plus ok.

>>>

De manière générale, ne faites appel à l'introspection que si vous avez une bonne raison de le faire et évitez ce genre de syntaxe. Il est quand même plus propre d'écrireobjet.attribut = valeurqueobjet.\_\_dict\_\_[nom\_attribut] = valeur.

Nous n'irons pas plus loin dans ce chapitre. Je pense que vous découvrirez dans la suite de ce livre l'utilité des deux méthodes que je vous ai montrées.

**En résumé**

* On définit une classe en suivant la syntaxeclass NomClasse:.
* Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe.
* Les méthodes d'instance prennent en premier paramètreself, l'instance de l'objet manipulé.
* On construit une instance de classe en appelant son constructeur, une méthode d'instance appelée\_\_init\_\_.
* On définit les attributs d'une instance dans le constructeur de sa classe, en suivant cette syntaxe :self.nom\_attribut = valeur.

**Les propriétés**

[Connectez-vous](https://openclassrooms.com/fr/login) ou [inscrivez-vous](https://openclassrooms.com/fr/register) gratuitement pour bénéficier de toutes les fonctionnalités de ce cours !

Au chapitre précédent, nous avons appris à créer nos premiers attributs et méthodes. Mais nous avons encore assez peu parlé de la philosophie objet. Il existe quelques confusions que je vais tâcher de lever.

Nous allons découvrir dans ce chapitre les propriétés, un concept propre à Python et à quelques autres langages, comme le Ruby. C'est une fonctionnalité qui, à elle seule, change l'approche objet et le principe d'encapsulation.

**Qu'est-ce que l'encapsulation ?**

L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet. Dans la plupart des langages orientés objet, tels que le C++, le Java ou le PHP, on va considérer que nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe. Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe,mon\_objet.mon\_attribut.

Mais c'est stupide ! Comment fait-on pour accéder aux attributs ?

On va définir des méthodes un peu particulières, appelées des **accesseurs** et **mutateurs**. Les accesseurs donnent accès à l'attribut. Les mutateurs permettent de le modifier. Concrètement, au lieu d'écriremon\_objet.mon\_attribut, vous allez écriremon\_objet.get\_mon\_attribut(). De la même manière, pour modifier l'attribut écrivezmon\_objet.set\_mon\_attribut(valeur)et non pasmon\_objet.mon\_attribut = valeur.

getsignifie « récupérer », c'est le préfixe généralement utilisé pour un accesseur.  
setsignifie, dans ce contexte, « modifier » ; c'est le préfixe usuel pour un mutateur.

C'est bien tordu tout cela ! Pourquoi ne peut-on pas accéder aux attributs directement, comme on l'a fait au chapitre précédent ?

Ah mais d'abord, je n'ai pas dit que vous ne *pouviez* pas. Vous pouvez très bien accéder aux attributs d'un objet directement, comme on l'a fait au chapitre précédent. Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages. En Python, c'est un peu plus subtil.

Mais pour répondre à la question, il peut être très pratique de sécuriser certaines données de notre objet, par exemple faire en sorte qu'un attribut de notre objet ne soit pas modifiable, ou alors mettre à jour un attribut dès qu'un autre attribut est modifié. Les cas sont multiples et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

L'inconvénient de devoir écrire des accesseurs et mutateurs, comme vous l'aurez sans doute compris, c'est qu'il faut créer deux méthodes pour chaque attribut de notre classe. D'abord, c'est assez lourd. Ensuite, nos méthodes se ressemblent plutôt. Certains environnements de développement proposent, il est vrai, de créer ces accesseurs et mutateurs pour nous, automatiquement. Mais cela ne résout pas vraiment le problème, vous en conviendrez.

Python a une philosophie un peu différente : pour tous les objets dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait au chapitre précédent. On peut y accéder et les modifier en écrivant simplementmon\_objet.mon\_attribut. Et pour certains, on va créer des propriétés.

**Les propriétés à la casserole**

Pour commencer, une petite précision : en C++ ou en Java par exemple, dans la définition de classe, on met en place des principes d'accès qui indiquent si l'attribut (ou le groupe d'attributs) est privé ou public. Pour schématiser, si l'attribut est public, on peut y accéder depuis l'extérieur de la classe et le modifier. S'il est privé, on ne peut pas. On doit passer par des accesseurs ou mutateurs.

En Python, il n'y a pas d'attribut privé. Tout est public. Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez. Pour faire respecter l'encapsulation propre au langage, on la fonde sur des conventions que nous allons découvrir un peu plus bas mais surtout sur le bon sens de l'utilisateur de notre classe (à savoir, si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, je ne vais pas chercher à y accéder depuis l'extérieur de la classe).

Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela ». De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.

Mais ces propriétés, c'est quoi ?

Hum… eh bien je pense que pour le comprendre, il vaut mieux les voir en action. Les propriétés sont des objets un peu particuliers de Python. Elles prennent la place d'un attribut et agissent différemment en fonction du contexte dans lequel elles sont appelées. Si on les appelle pour modifier l'attribut, par exemple, elles vont rediriger vers une méthode que nous avons créée, qui gère le cas où « on souhaite modifier l'attribut ». Mais trêve de théorie.

**Les propriétés en action**

Une propriété ne se crée pas dans le constructeur mais dans le corps de la classe. J'ai dit qu'il s'agissait d'une classe, son nom estproperty. Elle attend quatre paramètres, tous optionnels :

* la méthode donnant accès à l'attribut ;
* la méthode modifiant l'attribut ;
* la méthode appelée quand on souhaite supprimer l'attribut ;
* la méthode appelée quand on demande de l'aide sur l'attribut.

En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit nos accesseur et mutateur d'objet.

Mais j'imagine que ce n'est pas très clair dans votre esprit. Considérez le code suivant, je le détaillerai plus bas comme d'habitude :

class Personne:

"""Classe définissant une personne caractérisée par :

- son nom ;

- son prénom ;

- son âge ;

- son lieu de résidence"""

def \_\_init\_\_(self, nom, prenom):

"""Constructeur de notre classe"""

self.nom = nom

self.prenom = prenom

self.age = 33

self.\_lieu\_residence = "Paris" # Notez le souligné \_ devant le nom

def \_get\_lieu\_residence(self):

"""Méthode qui sera appelée quand on souhaitera accéder en lecture

à l'attribut 'lieu\_residence'"""

print("On accède à l'attribut lieu\_residence !")

return self.\_lieu\_residence

def \_set\_lieu\_residence(self, nouvelle\_residence):

"""Méthode appelée quand on souhaite modifier le lieu de résidence"""

print("Attention, il semble que {} déménage à {}.".format( \

self.prenom, nouvelle\_residence))

self.\_lieu\_residence = nouvelle\_residence

# On va dire à Python que notre attribut lieu\_residence pointe vers une

# propriété

lieu\_residence = property(\_get\_lieu\_residence, \_set\_lieu\_residence)

Vous devriez (j'espère) reconnaître la syntaxe générale de la classe. En revanche, au niveau du lieu de résidence, les choses changent un peu :

* Tout d'abord, dans le constructeur, on ne crée pas un attributself.lieu\_residencemaisself.\_lieu\_residence. Il n'y a qu'un petit caractère de différence, le signe souligné \_ placé en tête du nom de l'attribut. Et pourtant, ce signe change beaucoup de choses. La convention veut qu'on n'accède pas, depuis l'extérieur de la classe, à un attribut commençant par un souligné \_. C'est une convention, rien ne vous l'interdit… sauf, encore une fois, le bon sens.
* On définit une première méthode, commençant elle aussi par un souligné \_, nommée\_get\_lieu\_residence. C'est la même règle que pour les attributs : on n'accède pas, depuis l'extérieur de la classe, à une méthode commençant par un souligné \_. Si vous avez compris ma petite explication sur les accesseurs et mutateurs, vous devriez comprendre rapidement à quoi sert cette méthode : elle se contente de renvoyer le lieu de résidence. Là encore, l'attribut manipulé n'est paslieu\_residencemais\_lieu\_residence. Comme on est dans la classe, on a le droit de le manipuler.
* La seconde méthode a la forme d'un mutateur. Elle se nomme\_set\_lieu\_residenceet doit donc aussi être inaccessible depuis l'extérieur de la classe. À la différence de l'accesseur, elle prend un paramètre : le nouveau lieu de résidence. En effet, c'est une méthode qui doit être appelée quand on cherche à modifier le lieu de résidence, il lui faut donc le nouveau lieu de résidence qu'on souhaite voir affecté à l'objet.
* Enfin, la dernière ligne de la classe est très intéressante. Il s'agit de la définition d'une propriété. On lui dit que l'attributlieu\_residence(cette fois, sans signe souligné \_) doit être une propriété. On définit dans notre propriété, dans l'ordre, la méthode d'accès (l'accesseur) et celle de modification (le mutateur).

Quand on veut accéder àobjet.lieu\_residence, Python tombe sur une propriété redirigeant vers la méthode\_get\_lieu\_residence. Quand on souhaite modifier la valeur de l'attribut, en écrivantobjet.lieu\_residence = valeur, Python appelle la méthode\_set\_lieu\_residenceen lui passant en paramètre la nouvelle valeur.

Ce n'est pas clair ? Voyez cet exemple :

>>> jean = Personne("Micado", "Jean")

>>> jean.nom

'Micado'

>>> jean.prenom

'Jean'

>>> jean.age

33

>>> jean.lieu\_residence

On accède à l'attribut lieu\_residence !

'Paris'

>>> jean.lieu\_residence = "Berlin"

Attention, il semble que Jean déménage à Berlin.

>>> jean.lieu\_residence

On accède à l'attribut lieu\_residence !

'Berlin'

>>>

Notre accesseur et notre mutateur se contentent d'afficher un message, pour bien qu'on se rende compte que ce sont eux qui sont appelés quand on souhaite manipuler l'attributlieu\_residence. Vous pouvez aussi ne définir qu'un accesseur, dans ce cas l'attribut ne pourra pas être modifié.

Il est aussi possible de définir, en troisième position du constructeurproperty, une méthode qui sera appelée quand on feradel objet.lieu\_residenceet, en quatrième position, une méthode qui sera appelée quand on ferahelp(objet.lieu\_residence). Ces deux dernières fonctionnalités sont un peu moins utilisées mais elles existent.

Voilà, vous connaissez à présent la syntaxe pour créer des propriétés. Entraînez-vous, ce n'est pas toujours évident au début. C'est un concept très puissant, il serait dommage de passer à côté.

**Résumons le principe d'encapsulation en Python**

Dans cette section, je vais condenser un peu tout le chapitre. Nous avons vu qu'en Python, quand on souhaite accéder à un attribut d'un objet, on écrit tout bêtementobjet.attribut. Par contre, on doit éviter d'accéder ainsi à des attributs ou des méthodes commençant par un signe souligné \_, question de convention. Si par hasard une action particulière doit être menée quand on accède à un attribut, pour le lire tout simplement, pour le modifier, le supprimer…, on fait appel à des propriétés. Pour l'utilisateur de la classe, cela revient au même : il écrit toujoursobjet.attribut. Mais dans la définition de notre classe, nous faisons en sorte que l'attribut visé soit une propriété avec certaines méthodes, accesseur, mutateur ou autres, qui définissent ce que Python doit faire quand on souhaite lire, modifier, supprimer l'attribut.

Avec ce concept, on perd beaucoup moins de temps. On ne fait pas systématiquement un accesseur et un mutateur pour chaque attribut et le code est bien plus lisible. C'est autant de gagné.

Certaines classes ont besoin qu'un traitement récurrent soit effectué sur leurs attributs. Par exemple, quand je souhaite modifier un attribut de l'objet (n'importe quel attribut), l'objet doit être enregistré dans un fichier. Dans ce cas, on n'utilisera pas les propriétés, qui sont plus utiles pour des cas particuliers, mais plutôt des méthodes spéciales, que nous découvrirons au prochain chapitre.

**En résumé**

* Les propriétés permettent de contrôler l'accès à certains attributs d'une instance.
* Elles se définissent dans le corps de la classe en suivant cette syntaxe :nom\_propriete = property(methode\_accesseur, methode\_mutateur, methode\_suppression, methode\_aide).
* On y fait appel ensuite en écrivantobjet.nom\_proprietecomme pour n'importe quel attribut.
* Si l'on souhaite juste lire l'attribut, c'est la méthode définie comme accesseur qui est appelée.
* Si l'on souhaite modifier l'attribut, c'est la méthode mutateur, si elle est définie, qui est appelée.
* Chacun des paramètres à passer àpropertyest optionnel.