La programmation orienté objet

Thibault PENNING

Introduction au concept de programmation orienté objet ou POO.

La programmation objet constitue une part importante de la programmation Python que nous allons explorer ici.

Définitions

Définition 0.1 (Paradigme (programmation))**.** Un paradigme en programmation est une manière de penser et d'organiser du code.

> **Note**
>
> Programmer est comme résoudre un problème. Plusieurs solutions existent et plusieurs approches pour le résoudre existent aussi. Le paradigme ici est ce qui guide la résolution de ce problème. Le paradigme a un impact non seulement sur le code, mais aussi la manière de réfléchir et même la syntaxe. Ainsi certain paradigme sont plus adaptés (il est plus facile de réfléchir) pour certaine tache que d'autre.

La programmation objet st un paradigme de programmation où l'on manipule des objets. C'est le paradigme principal derrière Python, où tout (ou presque) est objet.

Définition 0.2 (Objet)**.** Un objet est un ensemble regroupant des données, des fonctions à appliquer a ces données, ainsi que la manière d'interagir avec ces objets.

Prenons par exemple `"Salut"`. Ici salut est un objet (de type `str`). Ainsi l'objet contient des valeurs, ici l'ensemble des caractères qui forme la structure de donnée de chaine. Il lui est aussi associé des fonctions. Ainsi la fonction `upper` (qui converti tout en majuscule) est aussi associé à cet objet. En effet, il est possible d'appeler `upper` sur l'objet précédent : `"Salut".upper()`. En revanche `upper` n'est pas définie autre par (il n'est pas possible de faire `upper("Salut")`). `upper` est donc parti intégrante de l'objet `"Salut"`. De même pour `append` sur `[1,2,3]` par exemple. Il est intéressant de constater 2 choses :

- `upper` ne prend pas en paramètre d'argument et pourtant il est capable de retourner quelque chose ayant du sens. Une fonction définie sur un objet est conscient de tout ce qui constitue l'objet. Il n'y a pas besoin ainsi de donner les caractères qui la compose car la fonction `upper` est dans l'objet elle connait donc déjà ces caractère !
- `append` est capable de modifier l'objet. Ainsi `[1,2,3].append(4)` modifie l'objet `[1,2,3]` pour contenir `[1,2,3,4]`. Les fonctions étant partie intégrante de l'objet, elles sont aussi capables de le modifier.

Enfin un objet défini aussi les règles d'interaction entre objet, par exemple `3 * "a"`, comment faire intéragir l'objet `3` avec `"a"`.

> Mise en garde
>
> Ici un objet n'est pas une variable. C'est important de faire la distinction entre les deux. Une variable peut etre vu comme une étiquette ou un tiroire, et un objet comme un objet de la vie réel (un téléphone, un voiture, …).

Définition 0.3 (Attribut)**.** Un attribut est une donnée d'un objet. L'ensemble des données d'un objet est donc l'ensemble de ses attributs.

Définition 0.4 (Méthode)**.** Une méthode est une fonction qui appartient à un objet.

`upper` est donc une méthode de `"Salut"`.

Créer ses propres objets

Classes

Afin de définir un objet nous allons utiliser un système de classe.

Définition 0.1 (Classe)**.** Une classe est une définition partagé par des objets. Il permet de définir les nom et type des attributs ainsi que le code des méthodes qui sont associé au objet.

> Analogie
>
> Pour comprendre le principe d'une classe on prend souvent l'exemple d'un plan d'une maison. Chaque maison possède un certain nombre d'objet et de mur. On définie des interupteur qui allume des lumière (comme des méthode) mais aussi des meuble et des portes (comme des attributs). Un même plan peu construire plusieurs maison, mais chaque maison partageras les même agencement de porte meuble ou compteur electrique. Chaque

maison auras en revanche un état différent (allumer la lumière d'une maison ne change que la lumière de cette maison), de même si les meubles sont au même endroit sont contenu peut varier. Un objet est chaque maison (unique) et la classe est le plan qui a servit à la construire.

Nous allons ici utilisé un exemple afin de continué notre cours. Nous allons créer un objet `Eleve` qui représenteras certaines caractéristique afin de modéliser ce qu'est un élève.

Pour créer une classe nous utilisons la syntaxe suivante :

```
class Eleve:
  pass
```

Lorsque nous voulons créer un nouvelle élève la syntaxe est la suivante :

```
e1 = Eleve()  # Un premier élève
e2 = Eleve()  # Un second élève
```

Maintenant que nous avons définie un objet nous pouvons commencer à ajouter à sa classe des méthodes et des attributs.

Pour rajouter des methodes nous définisson une fonction dans une classe de la même manière que d'habitude :

```
class Eleve:

  def presentation(self)->None:                                    ①
    print("Bonjour, je suis un élève")
```

① Voir Important 1

Afin d'appeler cette méthode, nous utilisons la manière suivante :

```
e1 = Eleve()

e1.presentation()
```

! Important 1

Remarquer un détail important. Ici `presentation` possède un argument (ici `self`) mais ne l'utilise pas. De plus, lorsque l'on appel cetméthodesion nous le faisons sans argument. En effet (quasi)**TOUTES** les methodes en Python doivent posséder au moins un argument,

car le premier arguement a toujours pour valeurs l'objet avec lequel il est appelé.
Vous pouvez voir une certaine équivalence avec les deux lignes ci dessous :

```
e1.presentation()

presentation(e1)
```

> **Pourquoi ?**
>
> Nous avons dit que les méthodes permettent de voir et changer les données des objet auquel il sont liés. Prendre en argument une référence a l'objet lui même permet de le modifier. Nous verrons cela juste après.

Maintenant voyons voir comment ajouter des arguments.

L'idée est la suivante : nous allons créer une fonction que nous appelerons afin d'intialiser les valeurs. En effet nous avons accès a la variable `self` où nous pouvons modifier (et créer) des attributs.

Ainsi le code suivant est juste :

```
class Eleve:

  def intialise(self, prenom:str, classe:str)->None:
    self.prenom = prenom                                    ②
    self.classe = classe
    self.notes = []                                         ①


  def presentation(self)->None:
    print("Bonjour, je suis", self.prenom, "élève en", self.classe)   ③




e1 = Eleve()
e1.initialise(prenom="Titeuf", classe="CE2")

print("L'élève e1 est en classe de :", e1.classe)           ④
```

① Initialisation d'un attribut avec une valeur prédéterminer
② Initialisation d'un attribut avec une valeur que l'on choisis lors de la création

③ Utilisation des attributs dans une méthode
④ Utilisation des attributs dans une fonction (*"en dehors"* de l'objet)

> **Important**
>
> Remarque que l'on utilise toujours `self.` a chaque fois que l'on veut faire référence à un attribut dans une méthode (que ce soit pour modifier ou utiliser sa valeur). Utiliser sans `self`, cela fonctionne mais n'auras pas l'effet escompté (pas d'exception pour ce cas)… Il ne faut donc pas l'oublier !
> Lorsque l'on est /à "l'extérieur"* de l'objet, on utilise la syntaxe suivante pour accéder à un attribut : `objet.nom_attribut`.

> **Note**
>
> Maintenant que vous en savez plus vous ne remarquer pas un lien avec ce que vous faisiez avant ?
> `l1.append(1)`, `"Salut".upper()`, …
> Oui, en Python les tableau, chaine de caractère et autre sont des… objets à part entière !

Il subsiste un soucis, si nous oublions l'initialisation, les attribut ne serons pas initialiser, des erreurs serons donc a suivre. Pour éviter cela, Python nous fournis un sytaxe particulièrement astucieuse :

1. On modifie le nom de la méthode `initialisation` par `__init__` (deux tiret bas, puis le mot "init" puis deux autre tiret bas).
2. Lorsque l'on crée un objet, on passe les paramètre de initialisation entre les parenthèse du nom de la classe.

Ainsi les oublie sont impossible : on initialise l'objet en même temps que sa création !

```
class Eleve:

  def __init__(self, prenom:str, classe:str)->None:
    self.prenom = prenom
    self.classe = classe
    self.notes = []


  def presentation(self)->None:
    print("Bonjour, je suis", self.prenom, "élève en", self.classe)

e1 = Eleve(prenom="Titeuf", classe="CE2")
```

Bien évidement, chaque objet étant unique, les valeurs des attributs leur son propre

```
e1 = Eleve(prenom="Titeuf", classe="CE2")
e2 = Eleve(prenom="Nadia", classe="CE2")
e3 = Eleve(prenom="Marco", classe="5ème")

e1.presentation()
#>>> Bonjour, je suis Titeuf élève en CE2
e2.presentation()
#>>> Bonjour, je suis Nadia élève en CE2
e3.presentation()
#>>> Bonjour, je suis Marco élève en 5ème
```

Synonyme

On appelle aussi un *objet* une *instance* (d'une classe).
On appelle aussi *attribut* un *champ* ou une *propriété*.
On appelle souvent l'ensemble des valeurs des attributs de l'objet *l'état* de ce dernier.

Une hygiène nécéssaire

En Python les attribut peuvent être créer "à la volée", comme nous l'avons vu en créan nous même une fonction qui ajoute des attributs. Si cela est tout à fait légal en Python, ce n'est pas vraiment dans le cadre de la POO voulus par le programme, et pire encore cela est source d'erreurs (trop courant).
Ainsi, il vous seras demander de définir **TOUTS** les attributs **SEULEMENT** dans la fonction `__init__` (ou assimilé). Un manquement à cette règle serait considéré comme une possible erreur.

Typage avec Self

Nous avons vu comment donnée un indication de type des variable en Python. Pour cela il suffit de connaitre le type de la variable. Ici le type d'un objet est simplement sa classe :

```
class Eleve:
  pass

e = Eleve()

print(type(e))
#>>> <class '__main__.Eleve'>

print(type(e) is Eleve)
#>>> True
```

On peut donc utiliser le typage ci dessous sans soucis :

```
class Eleve:
  pass

e:Eleve = Eleve()
```

Cependant lorsque nous l'utilisons dans la classe que nous somme en train de définir, cela provoque une erreur :

```
class Eleve:
    def hello(self)->Eleve:
            return self

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in Eleve
NameError: name 'Eleve' is not defined
```

Cela est du au faite que `Eleve` n'existe qu'après la définition complète de la classe. Pour pallier a ce problème, le module `typing` fournis le type `Self` qui permet justement definir que le type de retourn est le type de la classe actuellement définie.

```
from typing import Self

class Eleve:
    def hello(self)->Self:
            return self
```

Programmation objet et structure de données

A quoi ça sert ?

La programation objet fournis un terrain de jeu fantastique car il permet de définir des comportements dépendant d'un état sans a avoir a pensé à le décrire complétement. Nous avons juste besoin de pensé au donner stocker et au méthode que nous voulons appeler. Tout étant au même endroit, la POO nous permet des manipulation complexe sans y penser

La POO est relativement vieille (créer dans les années 70 avec le language SmallTalk) et est aujourd'hui largement utilisé. On peu citer par exemple les moteur de jeu vidéo qui en font un passage obligatoire. On peux aussi citer plus récement les différente librairie d'apprentissage machine, qui pour la plupart mélange POO et programation fonctionnel.

Une des utilisations que vous allez largement utiliser est celle qui permet de créer et de manipuler facilement des structures de données. En effet la POO vous offre l'abstraction suffisante pour utiliser les structure de données sans se soucier de l'implémentation (comme vous le faisiez déjà avec des tableaux (list) etc…)

Passage par référence

Un détail est important pour l'implémentation classique de structure de donnée : les passages sont par référence et non par valeur lors de l'assignation a une variable. Pour faire simple, la variable ne stoque en vérité non pas l'intégralité des données, mais simplement une référence a l'objet, qui lui est stocker autre part. Ce comportement est le même que celui des tableaux par exemple. Un exemple frapant est lorsque l'on tente de copier une variable de manière malheureuse comme cela :

```python
class Eleve:
  def __init__(self, prenom:str, classe:str)->None:
    self.prenom = prenom

e1 = Eleve("Titeuf")
e2 = e1

print(e1 is e2)                                                    ①
#>>> True


e2.prenom = "Marco"
print(e2.prenom)
#>>> Marco
```

```
print(e1.prenom)
#>>> Marco
```

① Ici `is` permet de tester si les deux objet sont les même

Si l'on change la valeur d'un seul des deux (ici `e2`), les deux sont modifié, car les deux **référence le même* objet

> **Note**
>
> Si l'on souhaite copier les valeurs, nous utiliserons la fonction `copy` ou mieux `deepcopy` de la librairie Python `copy`
>
> ```
> from copy import copy
>
> class Eleve:
>   def __init__(self, prenom:str, classe:str)->None:
>     self.prenom = prenom
>
> e1 = Eleve("Titeuf")
> e2 = copy(e1)                                                    ①
>
> print(e1 is e2)
> #>>> False
> ```
>
> ① Ici `e1` n'est pas `e2`, mais simplement une copie de la valeur initiale. Toute modification de `e2` ne se repercuteras pas sur `e1` et inversement.

Un exemple : les listes !

Rappeler vous des listes chainée, où chaque maillon possède une référence au prochain maillon. Nous pouvons grâce au classes et la POO programmer une version simplifié par rapport au implémentation précédente!. Et en utilisant le passage par référence, nous povons facilement réaliser cette chaine.

```
class Maillon:
  def __init__(self, data):
    self.data = data
    self.succ = None
```

```
def suivant_existe(self):
  return self.succ is not None

def suivant(self):
  if self.suivant_existe():
    return self.succ
  else:
    raise ValueError("Il n'y a pas de maillon suivant")

def ajoute_debut(self, data):
  nouveau = Maillon(data)
  nouveau.succ = self
  return nouveau

def valeur(self):
  return self.data

def affiche_liste(self):
  p = self
  while p.suivant_existe():
    print(p.valeur(), " -> ", end="")
    p = p.suivant()
  print(p.valeur())
```

Pour aller plus loin

Surcharge d'opérateur

La surcharge d'opérateur vous permet de réaliser une partie des opérations, comme l'addition, la multiplication, ou encore la convertion en chaine de caractère, de façon naturelle, comme vous le ferrez avec des entiers par exemple.

Pour cela nous devons redéfinir les méthode qui serons appelé par Python. Par défault, certaines de ces méthode ont un code prédéfinie, d'autre non. Chaqu'une de ces méthode seras de la forme suivante : `__nom_methode__`, que l'on appel parfois *dunder* (oui `__init__` en fait partie !).

Pour savoir quel méthode remplacer, ainsi que les arguments de ces derniers et le retour que nous devons implémenter, je vous laisse aller regarder la documentation sur le sujet :

- Généralité
- Listes des oppérateurs

On peux citer entre autre :

Operation	Nom méthode avec arguments
self == other	`__eq__(self, other) -> bool`
str(self)	`__str__(self) -> str`
not self	`__not__(self) -> Any`
self + other	`__add__(self, other) -> Self`
data in self	`__contains__(self, data) -> bool`
self[key]	`__getitem__(self, key) -> Any`
self[key]=data	`__setitem__(self, key, data)`

Par exemple, si nous créons une classe fraction

```
class Fraction:

  def __init__(self, nom, denom):
    self.nom = nom
    self.denom = denom
```

Et nous voulons que le code suivant soit correct :

```
a = Fraction(1,2)
b = Fraction(2,3)

c = a+b
```

Nous pouvons rajouter à fraction la méthode suivante :

```
def __add__(self, other)->Self:
  return Fraction(nom = self.nom * other.denom + other.nom * self.denom, denom = self.denom
```

Et tout se passeras comme prévu !

Membres privés

Jusqu'à présent, tout les membres de nos objets (méthode et attribut) pouvais être accéder de n'importe où dans le code. Cela ne pose en général pas de problème tant que nous somme les seuls utilisateur de notre classe. En revance, si nous le distribuons, ou nous reprenons notre code après quelque temps, nous pouvons pensez a de nombreuse erreurs qui peuvent arriver. En particulier, certains attributs n'ont pas d'interet d'être acceder (et en particulier modifier)

sans passez par des méthodes, soit parce qu'il n'ont pas de sens en eux même, ou parce que leur modification doit faire l'objet d'un soin particulier. Cela arrive quand plusieurs attributs sont lié (la modification d'un attribut entraine la possible modification d'un autre), ou lorsque la modification d'un attribut doit faire l'objet de nombreux test pour vérifier sa validité.

De même, certaine méthode n'ont pas toujours pour but d'être utiliser à l'extérieur. Par exemple, certaines méthode utilitaire, permettant la non répétition de code, mais dont la vérification des pré-requis n'est pas réaliser par la fonction elle même.

Dans tous ces cas, et bien d'autre, il est donc utile d'indiquer à l'utilisateur que ces membres ne sont pas déstiné à leur utilisation.

Pour cela il existe 2 convention en Python :

- Un membre dont le nom commence par un tire bas (ex: `_nom`) indique à l'utilisateur que cette variable peut-être modifié, mais que cela n'est pas un comportement prévu. En particulier cette variable peut être renomer a tout moement, ou contenir des donnée invalide. L'interpretteur n'agis en rien pour la protection.
- Un membre dont le nom commence par au moins 2 tiret bas, et finis par au plus 1 (ex : `__nom`) sont des dit membres privé, et que leur accès ne doit se faire que par les membres de la classe, et personne d'autre. Il ne doivent donc pas être toucher par l'utilisateur. En pratique l'interpretteur changeras le nom de cette variable leur de l'execution.

> **Astuce**
>
> En général il est plus simple de considérer qu'une variable est privé, sauf si elle ne doit pas l'être. Si ce n'est pas renforcer par l'idéologie de Python, c'est la route pris par de nombreux language orienté vers la sécurité comme Rust.
> Je vous conseil donc d'écrire une variable privé, a moindre de vouloir la rendre publique. Avois une varible privé ne veux pas dire qu'il est impossible de la modifier. Ainsi il est très courant de trouver des méthodes dit *getter* ou *setter* qui, respectivement, donne ou change le contenu de la variable privé. Comme cela on controlle ce qui est possible, et aussi dans quel condition cela est possible.
> Il est même possible facilement de réaliser ses propre *getter* et *setter* en Python avec le décorateur @property, dont l'utilisation dépasse le cadre du cours.

> **Note**
>
> Un dunder n'est pas une variable privé, car il finis par deux tiret bas

Variable de classe et méthode statique

Il est aussi possible de vouloir créer des membres dont le comportement ne dépend pas de l'état de l'objet.

Il existe 2 cas :

1. Pour les attributs, on parle de variable de classe. Ces derniers, ont un valeur commun (généralement constante) partagé par toute les instance d'une classe. Cela est utile pour faire par exemple référence a des valeurs utiles que peut prendre possiblement les instance de la classe. Ex : `string.ascii_lowercase` représentant les lettres possibles en miniscule. En général ces attribut ont un nom entiérement majuscule (convention indiquant une constante). Pour créer une variable de classe, rien de plus simple, il suffit de créer une variable dans une classe, sans rajouter le `self` devant.

```
class string:
  ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
```

> **Note**
>
> C'est sur ce principe que sont créer ce que l'on appel des énumération, qui permette de créer des constante, lier entre elle, ayant du sens, représentant par exemple les différent état dans lequel un objet peut se trouver.
> Les énumération ne sont pas dans le cadre du programme, mais sont très utile en programation fonctionelle.

2. Pour les méthode, il est possible de créer des méthodes dont le comportement ne dépend pas de l'état (ie des attribut) de la méthode. Cela peut être utile pour créer des fonction utilitaire, afin de cérifier la validé d'une valeur par exemple. La méthode la plus simple pour créer une fonction static est l'ajout du décorateur `@staticmethod` au dessus de la fonction. Il n'est ainsi plus nécéssaire d'avoir comme premier argument `self`

Ex :

```
class string:
  @staticmethod
  def charactere_valide(char):
    pass
```

> **Note**
>
> Il existe un dernier type qui s'appelle méthode de classe, qui au lieux de prendre en référence un objet, prennent en référence la classe de ce dernier. Cela n'est pas du tout utile dans le cadre de ce cours, car il nécéssite de solide base en héritage.

L'héritage

Il est possible de réutilise des classe, comme base pour en construire d'autre. Le concepte n'est pas dans le programme, et est assez compliqué a comprendre, car de nombreux détails s'y cache. Nous allons donc faire une présentation très en surface.

Lorsque l'on fait hérité une classe d'une autre, elle reprend tous les méthodes, sans modification du code. Il est cependant possible de réécrire (*overide*) une méthode déja définis dans une classe que l'on hérite.

Pour faire hérité une classe rien de plus simple, il suffit de mettre la classe que l'on veut hérité entre parenhèse à coté du nom de la classe :

```
class Parent:
  pass

class Fille(Parent):
  pass
```

Afin de faire référence à l'objet comme si elle apartenais à la classe mère avec la fonction `super()`. Cela permet en particulier d'appeler des méthodes qui ont été redéfinie. En particulier cela permet d'appeler la fonction d'intialisation :

```
class Parent:
  def __init__(nom):
    self.nom = nom

class Fille(Parent):
  def __init(nom, age):
    super().__init__(nom)
    self.age = age
```