Compléments en Python :
Remarque sur le typage dans une classe et introduction aux *dunder methods*

/home/phil/Documents/LMG/STS_A_2025-2027/PYTHON_STS2/C_classe_(heritage)/03_introduction_dunder_methods

Table des matières

I. Comment utiliser une classe dans la définition de cette même classe ?....1
II. Afficher les informations sur un objet ; *dunder methods*?....2
1. Qu'est-ce qu'une dunder method ?....2
2. À quoi servent-elles ?....2
3. Exemples courants....2
a. `__init__` — initialisation....2
b. `__str__` et `__repr__` — affichage....3
c. `__add__` — opérateur `+`....3
4. Liste de dunder methods fréquentes....3

I. Comment utiliser une classe dans la définition de cette même classe ?

Typer les paramètres la valeur retournée par une fonction constitue une bonne pratique. Cela permet l'utilisation de l'autocomplétion dans l'écriture de la fonction et cela clarifie le programme. Par exemple si l'on déclare une classe `Panier` et une fonction `fusionner` pour réunir le contenu de deux paniers dans un troisième, le code de test pourrait ressembler à :

```
1   class Panier:
2       def __init__(self, articles = None): # le panier est vide par défaut
3           if articles is None:
4               self.articles = []
5           else :
6               self.articles = list(articles) # ou articles.copy()
7
8       def fusionner(self, panier2: Panier) -> Panier:
9           """ créer un troisième Panier qui fusionne les deux autres"""
10          return Panier(self.articles + panier2.articles)
11
12  panier_1 = Panier(["oranges", "citron"])
13  panier_2 = Panier(["tomates"])
14  panier_3 = panier_1.fusionner(panier_2)
```

Mais à l'exécution il y a un petit problème :

```
 File "<string>", line 1, in <module>
```
Pb dans la définition de la classe Panier

```
  File "<string>", line 8, in Panier
```
Pb sur la ligne def fusionner(...

```
NameError: name 'Panier' is not defined
```
le type Panier n'est pas défini

Et oui ! A ce stade la classe Panier n'a pas encore été complètement définie ; elle est encore en cours d'écriture. Heureusement il y a une solution simple qui tient en une seule ligne :

```
from __future__ import annotations # A placer en tête du fichier
```

Remarque : ce problème est résolu depuis les versions 3.14+ de Python (sortie le 7 octobre 2025). Vous pouvez néanmoins laisser cette ligne pour assurer la compatibilité avec les versions moins récentes, versions 3.7+ à 3.13 de python.

Remarque : en C ou C++ ces problèmes sont résolus simplement car on distingue la déclaration de la définition d'une structure, d'une classe,…

class Vecteur ; // déclaration anticipée d'une classe Vecteur

class Vecteur { ….. } ; // définition de la classe Vecteur

Ce problème illustre bien la différence d'approche entre Python et C,C++ :

En Python : beaucoup de choses sont implicites par défaut (on minimise le volume de code).

En C, C++ : Tout est fixé, tout est très structuré. Et l'exemple ci-dessus justifie pourquoi.

II. Afficher les informations sur un objet ; *dunder methods*?

On aimerait bien pouvoir appliquer directement la fonction print sur n'importe quel objet pour avoir les informations qui nous interessent à son sujet. Bien sur on pourrait écrire pour chaque classe une fonction *afficher_informations()* comme nous l'avons fait jusqu'à présent et appeler *obj.voir_information()* Mais il y a une solution plus *pythonic* : l'utilisation d'une *dunder method* .

1. Qu'est-ce qu'une dunder method ?

Le terme **dunder** est une contraction de **"double underscore"**. Une *dunder method* est une méthode spéciale dont le nom est entouré de **deux underscores de chaque côté**, comme `__init__`, `__str__`, `__add__`, etc. On les appelle aussi **magic methods** ou **special methods**.

2. À quoi servent-elles ?

Elles permettent de définir le comportement de vos objets vis-à-vis des **opérations natives** du langage :

- création et destruction d'objets
- affichage (`print`, `str`, `repr`)
- comparaisons (`==`, `<`, `>`)
- opérations arithmétiques (`+`, `-`, `*`)
- accès aux éléments (`[]`), itération, appel (`()`)
- gestion du contexte (`with`)

Elles ne sont généralement **pas appelées directement** : Python les invoque automatiquement.

3. Exemples courants

a. `__init__` — initialisation

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

appelée automatiquement lors de la création d'un objet : `pt_A = (4 , 5)`

b. `__str__` et `__repr__` — affichage

```
    def __str__(self):
        return f"({self.x}, {self.y})"

    def __repr__(self):
        return f"Point({self.x!r}, {self.y!r})"
```

- `__str__` → affichage à destination de l'utilisateur (`print`) : `print(pt_A)`
- `__repr__` → affichage à destination du développeur (débogage) : `print(repr(pt_A))`

Les deux méthodes retournent une chaîne de caractères mais __repr__ fournit généralement une ligne de code qui permettrait de créer l'objet représenté ; nom_de_classe(Param,...). Notez aussi la présence de !r dans __repr__. Il s'agit encore d'un raccourci de python ; !r est un convertisseur de format. En fait self.x!r est équivalent à __repr__(self.x)

c. `__add__` — opérateur `+`

```
    def __add__(self, other):
        return Vecteur(self.x + other.x, self.y + other.y)
```

vec = Vecteur(1, 2) + Vecteur(3, 4) # vec = Vecteur(4, 6)

4. Liste de dunder methods fréquentes

Méthode	Rôle
`__new__` , `__init__`	Création / Initialisation de l'objet
`__str__` / `__repr__`	Représentation textuelle
`__eq__`, `__lt__`, `__gt__`	Comparaisons == , < , >
`__add__`, `__sub__`, `__mul__`	Opérations arithmétiques
`__len__`	`len(obj)`
`__getitem__` / `__setitem__`	Accès par indice
`__iter__` / `__next__`	Itération
`__call__`	Rendre l'objet appelable
`__enter__` / `__exit__`	`with`
`__hash__`	Utilisation comme clé de dictionnaire

Points importants :

- Il ne faut pas les inventer : leur nom est défini par Python.
- On peut les surcharger (override) dans ses propres classes.
- Elles rendent les objets personnalisés plus naturels à utiliser, comme s'ils étaient des types natifs.
- `__init__` initialise un objet ; ce n'est pas le constructeur à proprement parler (c'est `__new__`), mais en Python la création de l'objet s'effectue presque toujours implicitement (avec une méthode __new__() exécutée avec des paramètres par défaut).

Pour aller plus loin :

- La liste officielle complète est dans la documentation Python, section **"Data Model"** : https://docs.python.org/3/reference/datamodel.html#special-method-names
- En C++ la surcharge d'opérateur (redéfinition d'opérateur +, == , ….) existe aussi. Sans developper ici, le but est de souligner que s'il y a des différences entre les deux langages, il y a surtout beaucoup de similitudes.