NSI Activité 5

Durée :	110 min
Classe :	Entière
Objectifs :	Aborder les notions de POO en Python de manière pratique

1. Classe simplissime

Voici la manière la plus simple en Python de définir une classe, puis de l'instancier. Nous affichons le type de l'instance qui a été créée (cette instance est un objet). Nous n'avons pas défini de constructeur, donc le constructeur par défaut est appelé. On ne peut pas utiliser de paramètres pour l'instanciation, comme le montre l'image ci-contre. On dit que v1 est une instance de Voiture.

```
1  class Voiture:
2      pass
3
4  v1 = Voiture()
5  print(type(v1))
6
7  #renvoie une erreur
8  v2 = Voiture("AIJEJKH")
```

```
<class '__main__.Voiture'>
Traceback (most recent call last):
  File "<string>", line 6, in <module>
TypeError: Voiture() takes no arguments
>
```

2. Avec un constructeur

En Python, le fait de définir un constructeur permet par la même occasion de définir les attributs des instances d'une classe. Ici, chaque voiture instanciée possèdera une couleur et une immatriculation qui lui sont propres. On aurait pu appeler i et c respectivement « immatriculation » et « couleur ». Comme dans une fonction en fait ! Le fait d'appeler un constructeur avec des paramètres ne respectant pas sa signature conduira à des erreurs, comme pour v2.

```
1  class Voiture:
2
3      def __init__(self, i, c):
4          self.immatriculation = i
5          self.couleur = c
6
7
8  v1 = Voiture("BR-653-KL", "rouge")
9  print("l'objet est de type",type(v1),"son immatriculation
       est",v1.immatriculation,"sa couleur est",v1.couleur)
10 v2 = Voiture()
```

```
l'objet est de type <class '__main__.Voiture'> son immatriculation
    est BR-653-KL sa couleur est rouge
Traceback (most recent call last):
  File "<string>", line 10, in <module>
TypeError: __init__() missing 2 required positional arguments: 'i'
    and 'c'
>
```

3. Avec d'autres méthodes

Lorsque la méthode est destinée à être appelée sur un objet (méthode d'instance), on doit le lui spécifier : il faut pour cela lui passer ***self*** en premier argument.

```
1  class Voiture:
2
3      def __init__(self, i, c):
4          self.immatriculation = i
5          self.couleur = c
6
7      #prend un entier en paramètre
8      def klaxonne(self, longueur):
9          s = "T"
10         for i in range(longueur):
11             s+="U"
12         s+="T !!!"
13         print(s)
14
15 v1 = Voiture("DF-345-YU", "vert")
16 v1.klaxonne(30)
```

```
TUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUT !!!
>
```

4. Attributs statiques : on est au niveau de la classe, pas de l'instance

Il faut bien comprendre qu'ils se rapportent à une classe : en d'autres termes, ils sont partagés par toutes les instances de cette classe. Ils sont définis avant le constructeur, dans la classe. Ils ne peuvent pas être manipulés tels quels par le

constructeur, car le constructeur s'applique à l'instance courante, du moins « en cours de création ».

```
1  class Voiture:
2
3      compteur = 0
4
5      def __init__(self, i, c):
6          self.immatriculation = i
7          self.couleur = c
8          compteur = compteur + 1
9
10 v1 = Voiture("AZ-444-BN", "jaune")
```

```
Traceback (most recent call last):
  File "<string>", line 10, in <module>
File "<string>", line 8, in __init__
UnboundLocalError: local variable 'compteur' referenced before
    assignment
>
```

Si je veux incrémenter la variable « compteur » lors de l'instanciation, je dois l'appeler sur la classe et non pas sur l'instance :

```
1  class Voiture:
2
3      compteur = 0
4
5      def __init__(self, i, c):
6          self.immatriculation = i
7          self.couleur = c
8          Voiture.compteur = Voiture.compteur + 1
9
10 print("j'ai fabriqué",Voiture.compteur,"voitures")
11 v1 = Voiture("AZ-444-BN", "jaune")
12 v2 = Voiture("DF-666-WX", "vert")
13 v3 = Voiture("FG-333-UI", "noir")
14 print("j'ai fabriqué",Voiture.compteur,"voiture(s)")
```

```
j'ai fabriqué 3 voitures
>
```

Une solution quasiment identique, dans un esprit « réutilisable » car non attachée au nom de la classe, serait d'utiliser la fonction type() à la ligne 8 :

```
type(self).compteur = type(self).compteur + 1
```

Toutes les instances partagent le même attribut. Il est tout aussi correct d'écrire la chose suivante, tout en gardant en tête qu'on fait référence à un attribut statique :

```
v1 = Voiture("AZ-444-BN", "jaune")
print("j'ai fabriqué",v1.compteur,"voitures")
v2 = Voiture("DF-666-WX", "vert")
print("j'ai fabriqué",v1.compteur,"voitures")
v3 = Voiture("FG-333-UI", "noir")
print("j'ai fabriqué",v1.compteur,"voitures")
```

```
j'ai fabriqué 1 voitures
j'ai fabriqué 2 voitures
j'ai fabriqué 3 voitures
>
```

Attention : si l'on modifie un attribut statique à partir d'une instance, il devient en quelque sorte un attribut de cette instance. L'attribut statique n'est pas modifié, il continue d'exister et est visible par toutes les instances. Attention donc à ce genre de manipulation : soit vous le faites dans un but précis avec une idée en tête, soit vous vous abstenez.

```
1  class Voiture:
2
3      compteur = 0
4
5      def __init__(self, i, c):
6          self.immatriculation = i
7          self.couleur = c
8          type(self).compteur = type(self).compteur + 1
9
10 v1 = Voiture("AZ-444-BN", "jaune")
11 print(v1.__dict__)
12 v2 = Voiture("RT-211-QS", "gris")
13 print(v2.__dict__)
14 v2.compteur = 555
15 print(v2.__dict__)
16 print(Voiture.compteur)
```

```
{'immatriculation': 'AZ-444-BN', 'couleur': 'jaune'}
{'immatriculation': 'RT-211-QS', 'couleur': 'gris'}
{'immatriculation': 'RT-211-QS', 'couleur': 'gris', 'compteur': 555}
2
>
```

En d'autres termes, là, je viens de définir une classe Voiture avec 3 attributs dont un pouvant être **None**, sauf qu'il n'est pas défini dans le constructeur mais hors de ma classe. Sauf cas très particuliers, c'est « sale » de coder ainsi, car on estime que dans 99,99% des cas une instance doit être « prête à l'emploi » lorsqu'elle est créée : on ne doit pas lui ajouter des attributs ***après*** sa création !

5. Méthodes statiques : là aussi, au niveau de la classe, pas de l'instance

Une méthode statique en Python ne peut pas accéder aux attributs statiques : souvent on l'utilise comme fonction auxiliaire, générique, car n'étant pas attachée à un objet particulier.

```
class Voiture:

    def testeKilometrage(km):
        if km<100000:
           print("ok ça roule")
        else:
          print("à la casse !")

Voiture.testeKilometrage(54231)
Voiture.testeKilometrage(150000)
```

```
ok ça roule
à la casse !
>
```

6. Visibilité des attributs et des méthodes

Ils peuvent être publics, protégés(_), ou privés (__). Un petit exemple:

```
class Voiture:

    compteur = 0

    def __init__(self, i, c):
        self.__immatriculation = i
        self.couleur = c

v1 = Voiture("AZ-123-ER", "rose")
print(v1.couleur)
print(v1.immatriculation)
```

```
rose
Traceback (most recent call last):
  File "<string>", line 11, in <module>
AttributeError: 'Voiture' object has no attribute 'immatriculation'
>
```

Même si un __dict__ me l'afficherait, l'attribut « immatriculation » reste inaccessible hors de la classe lorsqu'on l'appelle de manière « classique ». C'est parce qu'il est protégé (protected) : c'est le programmeur qui décide cela, pour plusieurs raisons, la première étant « d'imposer » aux personnes utilisant ses classes une manière de s'en servir. Par exemple, en utilisant des méthodes appropriées, comme ici un accesseur (ou *getter*)

```
class Voiture:

    compteur = 0

    def __init__(self, i, c):
        self.__immatriculation = i
        self.couleur = c

    def getImmat(self):
        return self.__immatriculation

v1 = Voiture("AZ-123-ER", "rose")
print(v1.couleur)
print(v1.getImmat())
```

```
rose
AZ-123-ER
>
>
>
>
```

Une méthode que je ne recommande absolument pas, et que je mets juste pour votre culture générale, serait la suivante : ***print(v1._Voiture__immatriculation)***

C'est entre autres pour cela que Python n'est pas réputé pour sa rigueur … ☺

7. Attributs complexes

Outre des attributs créés dans des types primitifs, une classe Python peut posséder un ou plusieurs attributs correspondant à un objet, soit natif en Python (une liste par exemple) soit que nous avons créé nous.

Dans ce cas, il faut s'assurer que l'utilisateur, lorsqu'il crée un Cercle, lui donne bien un Point comme premier argument. Une autre solution serait d'utiliser la classe Point à l'intérieur de la classe Cercle,

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

class Cercle:

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

    def info(self):
        print("Cercle de centre",self.centre.x,self.centre.y
              ,"et de rayon", self.rayon )

p = Point(15,3)
c = Cercle(p, 10)
c.info()
```

```
15 3
Cercle de centre 15 3 et de rayon 10
>
```

l'utilisateur ne passant en paramètre plus que les coordonnées du Point. On s'assure ainsi que Cercle contient bien un attribut de type Point.

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

class Cercle:

    def __init__(self, coordX, coordY, rayon):
        self.centre = Point(coordX, coordY)
        self.rayon = rayon

    def info(self):
        print("Cercle de centre",self.centre.x,self.centre.y
            ,"et de rayon", self.rayon )

c = Cercle(15, 3, 10)
c.info()
```

```
Cercle de centre 15 3 et de rayon 10
>
```

8. Héritage

Il nous permet de gagner un temps précieux et nous évite des erreurs, car grâche à lui on écrit une et une seule fois le code, on ne se concentre pas sur les points communs, mais sur les différences. Mon exemple est un peu stupide mais on dira que pédagogiquement il tient la route : un point 3D est un point 2D auquel on a ajouté une 3ème dimension. Pour ce faire, on va utiliser grâce à ***super()*** le constructeur du parent

```
class Point2D:

    def __init__(self, x, y):
        self.x = x
        self.y = y

class Point3D(Point2D):

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

a = Point2D(3, 6)
print(a.__dict__)

b = Point3D(4, 7, 20)
print(b.__dict__)
```

```
{'x': 3, 'y': 6}
{'x': 4, 'y': 7, 'z': 20}
>
```