Introduction à la Programmation des Algorithmes

6.1. Python – Création de types

François Fleuret
https://fleuret.org/11x001/

Comme nous l'avons vu dans 5.2. "Python – Structures de données", Python est un langage orienté objets, ce qui veut dire que la création de nouveaux types se fait en créant des **classes**.

Nous allons dans ce cours voir le strict minimum et ignorer les mécanismes orientés objets à proprement parler.

On dit qu'une variable dont le type est une classe est un **objet**, et on parle aussi de façon équivalente d'une **instance de classe**.

À chaque classe et à chaque objet sont associées deux catégories d'attributs: des fonctions appelées **méthodes**, et des variables que l'on appelle ses **attributs de données**.

Dans la majorité des cas:

- Les méthodes sont des **attributs de classes**, ce qui veut dire qu'une méthode est la même pour tous les objets de ce type.
- Les variables sont des **attributs d'instances**.

Il est malgré tout possible de définir des variables de classes et des méthodes d'instance, mais nous ne le ferons pas.

La définition d'une classe se fait à l'aide du mot-clé `class`, qui est suivi d'un identifiant et d'une succession de fonctions spécifiques au nouveau type ainsi défini, qui seront les **méthodes de la classe**.

Par convention, les noms de classes commencent en Python par une lettre majuscule.

Suivant la philosophie très dynamique de Python, les attributs peuvent être ajoutés à une classe ou à un objet existant sans devoir les spécifier dans la définition de la classe, comme c'est le cas avec `struct` en C.

Rien ne nous empêche de créer une nouvelle classe sans méthode et d'ajouter des variables d'instance à des variables de ce type.

Par exemple

```
1  class Employe:
2      pass
3
4  def affiche_employe(e):
5      print(f'{e.nom} (bureau {e.bureau})')
6
7  f = Employe()
8  f.nom = "Francois Fleuret"
9  f.bureau = 410
10
11 affiche_employe(f)
```

affiche

```
Francois Fleuret (bureau 410)
```


Une meilleure approche serait de programmer `affiche_employe` comme une méthode de la classe.

Rajouter une méthode à une classe se fait en définissant une fonction dans la définition de la classe. Cette fonction aura toujours un premier argument du type de la classe, que l'on appelle traditionnellement `self`, et qui sera l'objet auquel la méthode est appliquée.

```
1  class Employe:
2      def affiche(self):
3          print(f'{self.nom} (bureau {self.bureau})')
4
5  f = Employe()
6  f.nom = "Francois Fleuret"
7  f.bureau = 410
8
9  f.affiche()
```

affiche

```
Francois Fleuret (bureau 410)
```

La plus importante des méthodes est `__init__` qui est la méthode exécutée lorsqu'un nouvel objet de ce type est créé.

Le rôle principal de cette méthode est d'ajouter à l'instance nouvellement créée les variables d'instance qu'elle doit avoir. Ci-dessous elle est appelée pour créer `f` ligne 9.

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def affiche(self):
7          print(f'{self.nom} (bureau {self.bureau})')
8
9  f = Employe('Francois Fleuret', 410)
10
11 f.affiche()
```

affiche

```
Francois Fleuret (bureau 410)
```


Les objets en Python sont mutables.

```
1  f = Employe('Francois Fleuret', 410)
2  g = f
3  f.nom = 'Capitaine Haddock'
4  g.affiche()
```

affiche

```
Capitaine Hadock (bureau 410)
```

Une classe en Python possède des méthodes par défaut qui peuvent être re-définies.

Elles ont des identifiants de la forme `__xxx__` et sont dites **spéciales** car elles ont des rôles particuliers dans le langage même.

Il est possible d'obtenir la liste des noms des attributs d'une classe ou d'un objet avec la fonction `dir` (il y en a beaucoup par défaut!):

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def etage(self):
7          return self.bureau // 100
8
9  print(dir(Employe))
```

affiche

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'etage']
```

```python
class Employe:
    def __init__(self, nom, bureau):
        self.nom = nom
        self.bureau = bureau

    def etage(self):
        return self.bureau // 100

f = Employe('Francois Fleuret', 410)

print(dir(f))
```

affiche

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'bureau', 'etage', 'nom']
```


Donc par exemple une classe possède une méthode `__init__` par défaut qui ne prend pas d'argument.

```python
class Truc:
    pass

x = Truc()
```

Elle est remplacée par celle que l'on programme.

```python
class Truc:
    def __init__(self, v):
        self.v = v

x = Truc()
```

affiche

```
  File "test.py", line 5, in <module>
    x = Truc()
TypeError: __init__() missing 1 required positional argument: 'v'
```

Une méthode qui existe par défaut et qu'il peut être intéressant de redéfinir est `__repr__(self)` qui retourne une chaîne de caractères décrivant la variable.

C'est cette méthode qui est appelée quand un objet est explicitement converti en chaîne de caractères.

La version par défaut est peu informative:

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6  f = Employe('Francois Fleuret', 410)
7
8  print(str(f))
```

affiche

```
<__main__.Employe object at 0x7fa66e04e310>
```



```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def __repr__(self):
7          return f'{self.nom} (bureau {self.bureau})'
8
9  f = Employe('Francois Fleuret', 410)
10
11 print(str(f))
```

affiche

```
Francois Fleuret (bureau 410)
```

La fonction `print` fait cette conversion en chaîne de caractères automatiquement, on peut donc écrire simplement `print(f)` ligne 11.

Comme on l'a dit, on peut modifier dynamiquement les variables d'instances

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def __repr__(self):
7          return f'{self.nom} (bureau {self.bureau})'
8
9  f = Employe('Francois Fleuret', 410)
10
11 print(f)
12
13 f.bureau = 412
14
15 print(f)
```

affiche

```
Francois Fleuret (bureau 410)
Francois Fleuret (bureau 412)
```


Et, plus surprenant, on peut modifier dynamiquement les méthodes de classes:

```
1  f = Employe('Francois Fleuret', 410)
2  g = Employe('Raoul Volfoni', 123)
3
4  print(f, g)
5
6  def juste_le_nom(self):
7      return self.nom
8
9  Employe.__repr__ = juste_le_nom
10
11 print(f, g)
12
13 Employe.truc = lambda self: print(f'Mon nom est {self.nom}!')
14
15 g.truc()
```

affiche

```
Francois Fleuret (bureau 410) Raoul Volfoni (bureau 123)
Francois Fleuret Raoul Volfoni
Mon nom est Raoul Volfoni!
```

Modifier ou rajouter des méthodes d'instance n'est pas aussi simple. Une telle méthode ne peut pas être une simple fonction mais doit être de type `types.MethodType`. Cela sort du cadre de ce cours.

Comme nous l'avons dit, il est possible d'avoir des variables de classe, qui sont donc communes à toutes les instances.

```
1  class Vaisseau:
2      g = 9.81
3
4      def __init__(self, masse):
5          self.masse = masse
6
7      def force_de_gravite(self):
8          return self.masse * self.g
9
10 v1 = Vaisseau(1000)
11 v2 = Vaisseau(10)
12
13 print(v1.force_de_gravite(), v2.force_de_gravite())
14
15 Vaisseau.g = 1.62
16
17 print(v1.force_de_gravite(), v2.force_de_gravite())
```

affiche

```
9810.0 98.10000000000001
1620.0 16.200000000000003
```

On peut tester si une classe ou un objet possède un attribut donné avec `hasattr`:

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def __repr__(self):
7          return f'{self.nom} (bureau {self.bureau})'
8
9      def etage(self):
10         return self.bureau // 100
11
12 f = Employe('Francois Fleuret', 410)
13
14 print(hasattr(f, 'nom'))
15 print(hasattr(f, 'etage'))
16 print(hasattr(f, 'batiment'))
```

affiche

```
True
True
False
```


Il est également possible de récupérer un attribut étant donné son nom avec `getattr`:

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def __repr__(self):
7          return f'{self.nom} (bureau {self.bureau})'
8
9      def etage(self):
10         return self.bureau // 100
11
12 f = Employe('Francois Fleuret', 410)
13
14 print(getattr(f, 'nom'))
15 print(getattr(f, 'etage'))
```

affiche

```
Francois Fleuret
<bound method Employe.etage of Francois Fleuret (bureau 410)>
```

Finalement, `del` permet de supprimer un attribut.

```
1  class Employe:
2      def __init__(self, nom, bureau):
3          self.nom = nom
4          self.bureau = bureau
5
6      def __repr__(self):
7          return f'{self.nom} (bureau {self.bureau})'
8
9  f = Employe('Francois Fleuret', 410)
10 del f.bureau
11 print(f)
```

affiche

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    print(f)
  File "test.py", line 7, in __repr__
    return f'{self.nom} (bureau {self.bureau})'
AttributeError: 'Employe' object has no attribute 'bureau'
```


Exemple: Nombres complexes

```python
class NombreComplexe:
    def __init__(self, re, im = 0):
        self.re = float(re)
        self.im = float(im)

    def __repr__(self):
        if self.im == 0:
            return f'{self.re}'
        elif self.im > 0:
            return f'{self.re}+{self.im}i'
        else:
            return f'{self.re}-{-self.im}i'

z, q = NombreComplexe(2, -1), NombreComplexe(-5)
print(z, q)
```

affiche

```
2.0-1.0i -5.0
```


Certaines de méthodes spéciales sont automatiquement appelées lors de l'utilisation des opérateurs associés.

```python
    def __neg__(self):
        return NombreComplexe(-self.re, -self.im)

    def __add__(self, other):
        return NombreComplexe(self.re + other.re, self.im + other.im)

    def __sub__(self, other):
        return NombreComplexe(self.re - other.re, self.im - other.im)

    def __mul__(self, other):
        return NombreComplexe(self.re * other.re - self.im * other.im,
                              self.re * other.im + self.im * other.re)

z = NombreComplexe(2, -1)
print(z + NombreComplexe(1), z * z)
```

affiche

```
3.0-1.0i 3.0-4.0i
```

```
    def __truediv__(self, other):
        u = other.re * other.re + other.im * other.im # u = |other|^2
        return NombreComplexe(
            (  self.re * other.re + self.im * other.im) / u,
            (- self.re * other.im + self.im * other.re) / u
        )

z = NombreComplexe(2, -1)
w = z / (z + NombreComplexe(1))
print(z, w, (NombreComplexe(-1) - z) * w)
```

affiche

```
2.0-1.0i 0.7-0.1i -1.9999999999999996+1.0i
```