

3 : Fonctions *built-in* et Surcharge d'opérateurs

Dans ce cours, on va voir comment on peut, pour une classe, **réécrire** un certain nombre de fonctions de base et d'opérateurs.

Ce sera notamment très utile pour manipuler des objets mathématiques.

3.1 : Fonctions *built-in*

3.1.1 : Définition et exemples

Une fonction *built-in* :

- existe de façon native dans le langage Python ,
- a un comportement par défaut quel que soit l'argument passé,
- peut modifier son comportement en fonction de l'argument passé.

Une liste est disponible [sur cette page](#).

Quelques exemples utiles en mathématiques sont :

- `abs ()` (calcule la valeur absolue d'un nombre),
- `sum ()` (somme les éléments d'un objet itérable, c'est-à-dire d'une liste, d'un tuple, etc.),
- `len ()` (renvoie la taille d'un itérable),
- ...

Exemple d'utilisation :

In [5]:

```
a = 2
mot = str(a)
print("mot = ", mot, " ; type de mot :", type(mot))

l = [2.3, 4.0]
mot_liste = str(l)
print("mot_liste = ", mot_liste, " ; type de mot_liste :", type(mot_liste))
print(mot_liste[0])
```

```
mot = 2 ; type de mot : <class 'str'>
mot_liste = [2.3, 4.0] ; type de mot_liste : <class 'str'>
[
```

Les fonctions `str()` et `type()` sont des ***built-in***. Elles existent pour tous les objets courants (nombres, listes, etc.).

- `str()` convertit l'objet en chaîne de caractères. Par exemple, la fonction `print()` utilise `str()`.
- `type()` renvoie le type de l'objet.

Ces fonctions existent pour les objets courants mais leur comportement peut ne pas vous plaire. Il est possible de modifier ce comportement : on parle alors de **surcharge**.

3.1.2 : Surcharge

Premier exemple : On a en fait déjà parlé de la surcharge d'une fonction *built-in* : celle de la fonction `__str__()` , qui est utilisée par la fonction `print()` .

In [6]:

```
class complex1:  
    def __init__(self, x, y):  
        self.reel = x  
        self.imag = y
```

```
c = complex1(2, 1)  
print("c = ", c)  
c_str = str(c)  
print("c_str = ", c_str)
```

```
c = <__main__.complex1 object at 0x7f7974273880>  
c_str = <__main__.complex1 object at 0x7f7974273880>
```

In [7]:

```
class complexe2:
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

    def __str__(self):
        return "partie réelle : " + str(self.reel) + " ; partie imaginaire : " + str(self.imag)

c = complexe2(2, 1)
print("c = ", c)
c_str = str(c)
print("c_str = ", c_str)
```

```
c = partie réelle : 2 ; partie imaginaire : 1
c_str = partie réelle : 2 ; partie imaginaire : 1
```

On a ainsi redéfini la fonction *built-in* `str()` pour la classe complexe. Dans chaque classe, on pourra redéfinir les *built-in*.

À l'utilisation, le *built-in* va **déterminer le type de l'objet et appeler l'implémentation correspondante**.

Il existe néanmoins des implémentations par défaut, comme on va le voir ci-dessous.

Second exemple: `int()` et `float()`, qui permettent de transformer un objet en **entier** ou en **float**.

In [5]:

```
c_i = int(c)
c_f = float(c)
print("c_i = ", c_i, " ; c_f = ", c_f)
```

```
-----
-
TypeError                                Traceback (most recent call las
t)
<ipython-input-5-93190db8fb2a> in <module>
----> 1 c_i = int(c)
      2 c_f = float(c)
      3 print("c_i = ", c_i, " ; c_f = ", c_f)

TypeError: int() argument must be a string, a bytes-like object or a numbe
r, not 'complexe2'
```

On va donc surcharger ces fonctions :

- `float()` renverra le module du nombre complexe,
- `int()` la partie entière de ce module.

In [11]:

```
import numpy as np

class complexe3:
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

    def __float__(self):
        module = np.sqrt(self.reel**2 + self.imag**2)
        return float(module)

    def __int__(self):
        return int(float(self))

c3 = complexe3(2, 1)
c3_i = int(c3)
c3_f = float(c3)
print("c3_i = ", c3_i, " ; c3_f = ", c3_f)
```

```
c3_i = 2 ; c3_f = 2.23606797749979
```

Dernier exemple: La fonction `min()` cherche le minimum dans une liste ou un tuple d'objets. Par exemple, sur des entiers :

In [7]:

```
print("minimum = ", min(3, 2, -1))  
print("minimum = ", min([3, 2, -1]))
```

```
minimum = -1  
minimum = -1
```

On essaye sur des objets de la classe `complexe3` :

In [8]:

```
c1 = complexe3(2.0, 1.0)
c2 = complexe3(1.0, 1.7)
c3 = complexe3(0.5, 2.7)
print('minimum: ', min([c1, c2, c3]))
```

```
-----
-
TypeError                                Traceback (most recent call last)
<ipython-input-8-3118894da69c> in <module>
      2 c2 = complexe3(1.0, 1.7)
      3 c3 = complexe3(0.5, 2.7)
----> 4 print('minimum: ', min([c1, c2, c3]))

TypeError: '<' not supported between instances of 'complexe3' and 'complexe3'
```


Cela ne marche pas car la *built-in* `min()` utilise l'opérateur `<`, qui n'est pas défini pour les complexes.

La surcharge du `min()` ne semble pas possible, ou est dans tous les cas très compliquée.

Tous les *built-in* ne se surchargent pas. Heureusement, il existe une autre solution : la **surcharge d'opérateurs**.

3.2 : Opérateurs

On voit que la programmation orientée objet permet de **redéfinir des fonctions générales** comme les *built-in*.

Il est assez naturel de vouloir redéfinir un maximum de fonctions ou de transformations sur des objets.

Parmi ces fonctions, les **opérateurs** occupent une place particulière.

3.2.1 : Définition et exemples

De quoi parle-t-on lorsqu'on parle d'**opérateur** ?

- Les opérateurs binaires algébriques : `+`, `*`, `-`, `/`, `%`, `**`
- les opérateurs de comparaison : `==`, `<`, `>`, `>=`, `<=`, `!=`
- les opérateurs d'identité : `is`, `is not`
- les opérateurs logiques : `and`, `or`, `not`
- les opérateurs d'appartenance : `in`, `not in`
- les opérateurs d'assignation : `+=`, `-=`, `*=`, `/=`, ...,
- et bien d'autres [sur cette page](#).

Le résultat de ces opérateurs dépend de l'objet sur lequel on les applique :

In [11]:

```
print("Opérateur + sur des float : ", 2.0 + 3.0)  
print("Opérateur + sur des chaînes de caractères : ", 'coucou, ' + 'ça va ?')
```

Opérateur + sur des float : 5.0

Opérateur + sur des chaînes de caractères : coucou, ça va ?

Pour les nombres, le `+` correspond à l'addition arithmétique. Pour les chaînes de caractères, il s'agit de la **concaténation**.

3.2.2 : Surcharge

On va maintenant regarder comment surcharger ces **opérateurs**.

Cela fonctionne comme les built-in : reprenons la problème du built-in `min()` pour les complexes.

In [12]:

```
class complexe5:
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

    def module(self): # fonction qui calcule le module
        return np.sqrt(self.reel**2 + self.imag**2)

    def __str__(self):
        return "partie réelle : " + str(self.reel) + " ; partie imaginaire : " + str(self)

    def __lt__(self, other): # opérateur <
        if self.module() < other.module():
            return True
        else:
            return False

    def __le__(self, other): # opérateur <=
        if self.module() > other.module():
            return False
        else:
            return True

    def __eq__(self, other): # opérateur ==
        if (self.reel == other.reel and self.imag == other.imag):
            return True
        else:
            return False
```


In [13]:

```
d1 = complexe5(2.0, 1.0)
print(d1.module())
d2 = complexe5(1.0, 1.7)
print(d2.module())
d3 = complexe5(0.5, 2.7)
print(d3.module())
print("d1 < d2 ? ", d1 < d2)
print('minimum : ', min([d1, d2, d3]))
```

2.23606797749979

1.972308292331602

2.7459060435491964

d1 < d2 ? False

minimum : partie réelle : 1.0 ; partie imaginaire : 1.7

Que s'est-il passé ?

- Le built-in `min()` est défini pour tout objet qui possède un opérateur `<`.
- On définit donc `<` dans notre classe : cela nous permet directement d'appliquer `min()`.

Un autre built-in, `sort()`, permet de trier une liste. Les algorithmes de tri optimaux ne sont pas faciles à implémenter : il est donc souhaitable d'utiliser ce built-in si possible.

Là encore, en re-définissant `<`, `<=` et `==`, on peut réutiliser le built-in `sort()` sur n'importe quel objet.

Les classes et les surcharges d'opérateur permettent donc de réutiliser des algorithmes compliqués, écrits de façon générale, sur une large gamme d'objets.

De la même façon, on peut redéfinir les opérateurs **arithmétiques** comme le `+` ou le `-`.

C'est très important pour les vecteurs, les matrices, etc.

Par exemple, lorsque vous utilisez les matrices de `numpy` (`numpy.array()`), ces opérateurs sont déjà surchargés et accessibles.

In [2]:

```
class complexe6:
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

    def __str__(self):
        return 'partie réelle = ' + str(self.reel) + ', partie imaginaire = ' + str(self.

    def __add__(self, other): # opérateur +
        res = complexe6(0, 0)
        res.reel = self.reel + other.reel
        res.imag = self.imag + other.imag
        return res

    def __mul__(self, other): # opérateur *
        res = complexe6(0, 0)
        res.reel = self.reel * other.reel - self.imag * other.imag
        res.imag = self.imag * other.reel + self.reel * other.imag
        return res
```

In [3]:

```
z1 = complexe6(1, 2)
z2 = complexe6(-1, 1)

print('addition :', z1 + z2)
print('addition et multiplication :', (z1 + z2) * z1)
```

addition : partie réelle = 0, partie imaginaire = 3

addition et multiplication : partie réelle = -6, partie imaginaire = 3

On peut définir des classes qui correspondent à des groupes, corps ou espaces vectoriels comme les matrices, les polynômes, les suites etc.

Le fait de pouvoir redéfinir les opérateurs comme le `=` ou le `-` simplifie considérablement les codes de calcul les utilisant.

Liste de certains opérateurs utiles :

- `__sub__` pour le `-`, `__isub__` pour le `-=`
- `__add__` pour le `+`, `__iadd__` pour le `+=`
- `__mul__` pour le `*`, `__imul__` pour le `*=`
- `__truediv__` pour le `/`, `__itruediv__` pour le `/=`
- `__pow__` pour le `**`, `__ipow__` pour le `**=`

- `__le__` pour le `<=`, `__lt__` pour le `<`

- `__ge__` pour le `>=`, `__gt__` pour le `>`
- `__eq__` pour le `==`, `__ne__` pour le `!=`

3.4 : TD

3.4.1 : Exercice 1 - Espace vectoriel

On considère l'espace vectoriel des fonctions affines $f(x) = ax + b$.

- 1) Quels seraient les attributs d'une classe représentant ces fonctions ?
- 2) Quelles méthodes pourraient être naturellement incluses dans la classe ?
- 3) Quels opérateurs arithmétiques pourraient être surchargés, et comment ?
- 4) Quel sens pourrait avoir la surcharge de l'opérateur `>` ?

3.4.2 : Exercice 2 - Trier une base de données

On se donne une base de donnée d'étudiants. Un individu est une classe contenant les attributs suivants :

- un nom,
- un prénom,
- un âge,
- le nombre d'années d'études depuis l'inscription,
- la date d'inscription.

1) Quels sont les types des quatre premiers attributs ?

2) Proposer une façon de stocker la date d'inscription.

3) Afin de trier la base de données par date d'inscription, proposer un algorithme pour l'opérateur `<` qui utilise la proposition de stockage de la question 2).