# Programmation Objet en Python

## La programmation objet, c'est quoi ? Ca sert à quoi ?

En Python, on doit considérer que tous les objets que l'on manipule sont des objets au sens de la programmation objet telle que vous avez pu la voir en C++, c'est-à-dire des entités regroupant:

- des données appelées attributs ou variables d'instance de l'objet
- des fonctions appelées métjodes de l'objet.

Par exemple, les entiers, les flottants, les complexes, des listes, les tuples, les chaînes de caractères sont des objets... Parmi les attributs d'un objet de la classe list, on peut citer ses éléments, parmi les mathodes de cette classe, les fonctions append(), extend(), copy() etc

Autre exemple : les complexes ! Les attributs d'un complexe sont sa partie réelle et sa partie imaginaire, parmi les méthodes associées à cet objet, on a module(), conjugate() etc

On va ainsi définir des classes ! C'est un type permettant de regrouper dans la même structure d'une part les informations liées à cette sutructure (les attributs) et d'autre part les opérations que l'on va pouvoir faire sur les éléments de la structure (méthodes)... En fait, une classe permet en quelque sorte d'introduire un nouveau type décrivant les opérations disponbiles sur une famille d'objets.

Termes techniques :

- "Classe" est la structure
- "objet" est un élément d'une classe
- "instanciation" correspond à la création d'un objet de la classe.

Un exemple avec le type List

In [1]:

```
L1=[4,2,1,3] #L1 est un objet liste (instance de la classe liste)
type(L1)
```

Out[1]:

```
list
```

In [2]:

```
dir(list)
```

Out[2]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__delslice__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getslice__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__setslice__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

In [3]:

```
L1.sort(); print("L1 apres sort :",L1) #objet.methode
L1.append(5); print('L1 apres append(5) :',L1) #objet.methode
L1.reverse() ; print("L1 apres reverse :",L1) #objet.methode
print len(L1)
```

```
('L1 apres sort :', [1, 2, 3, 4])
('L1 apres append(5) :', [1, 2, 3, 4, 5])
('L1 apres reverse :', [5, 4, 3, 2, 1])
5
```

# Comment définit-on une classe ?

Quel classe veut-on construire ? On propose de construire une classe Personne et de choisir comme attribut d'une personne son Nom, son Prénom, son age et son lieu de résidence

Pour définir une nouvelle classe, on résume ce dont on a besoin :

- le nom de la classe : Personne
- les attributs : nom, prénom, age, lieu de résidence

Pour cela, on utilise

- le mote-clé class, la syntaxe est simple : class NomDeLaClasse
- un constructeur (une methode de notre objet qui se charge de créer nos attributs. C'est la même méthode qui sera appelée quand on voudra créer notre objet

Treve de suspens, voici un exemple de code pour créer cette classe

In [4]:

```
class Personne: #Définition de la classe Personne
        """Classe définissant une personne caractérisée par :
        -son nom
        -son prenom
        -son age
        -sa ville"""

        def __init__(self,nm,pm,ag,lieu): #Notre methode constructeur
            """On ne définit qu'un attribut nom pour le moment"""
            self.nom=nm
            self.prenom=pm
            self.age=ag
            self.ville=lieu
```

Commentaires :

- class est le mot clé de création de la classe, Personne le nom d ela classe
- entre """ """ on a commenté la classe pour la documenter. Si on fait help(Personne) on aura cette description (comme pour les méthodes et fonctions de l'aide classique de Python)
- Attention au role des ":" et de l'indentation, comme d'habitude
- **init** est une méthode standard appelée constructeur
- self représente l'objet qui est en train d'être crée. Il doit apparaître en première position dans la définition de toutes les méthodes, mais il n'est pas nécessaire lors de l'appel
- on utilise le constructeur pour énumerer les champs de la classe

- Le constructeur peut prendre des paramètres en entrée (initialisation des champs par exemple)
- Noter le rôle de '.' dans l'accès aux champs

In [5]:

```
FD=Personne('Delebecque','Fanny',35,'Toulouse')
```

In [6]:

```
FD
```

Out[6]:

```
<__main__.Personne instance at 0x103d11440>
```

In [7]:

```
print FD.nom
print FD.prenom
print FD.age
print FD.ville

#Fanny déménage
FD.ville='Montpellier'
```

```
Delebecque
Fanny
35
Toulouse
```

In [8]:

```
print FD.ville
```

```
Montpellier
```

In [9]:

```
VL=Personne()
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-9-3fed9719bdaa> in <module>()
----> 1 VL=Personne()

TypeError: __init__() takes exactly 5 arguments (1 given)
```

Variable de classe : On peut aussi ajouter aux attribut de chaque objet de la classe une variable de classe, qui ne dépend pas de l'objet crée ("self") mais de la classe globale... Exemple : on veut ajouter un compteur pour savoir combien de personnes on a crées

In [10]:

```
class Personne: #Définition de la classe Personne
        """Classe définissant une personne caractérisée par :
        -son nom
        -son prenom
        -son age
        -sa ville"""

        #variable de classe
        compteur = 0

        def __init__(self,nm,pm,ag,lieu): #Notre methode constructeur
            """On définit les attributs un par un"""
            self.nom=nm
            self.prenom=pm
            self.age=ag
            self.ville=lieu
            Personne.compteur += 1
```

In [11]:

```
FD=Personne('Delebecque','Fanny',36,'Toulouse')
```

In [12]:

```
JD=Personne('Dupont', 'Jerome', 24, 'Mulhouse')
```

In [13]:

```
DV=Personne('Skywalker','Anakin',8,'Tatooine')
```

In [14]:

```
print Personne.compteur
```

```
3
```

Une autre façon de faire : on définit la classe Personne mais sans parametres en entrée

In [15]:

```
class Personne:

    compteur=0

    def __init__(self):
        self.nom=""
        self.prenom=""
        self.age=0
        self.ville=""
        Personne.compteur += 1
```

In [16]:

```
FD=Personne()
```

In [17]:

```
FD.nom='Delebecque'
FD.prenom='Fanny'
FD.ville='Toulouse'
```

In [18]:

```
print FD.nom
print FD.prenom
print FD.age
print FD.ville
```

```
Delebecque
Fanny
0
Toulouse
```

Un peu fastidieux... On aimerait bien fabriquer une méthode spécifique pour la saisie et l'affichage... Voyons comment faire

## Comment définit-on les méthodes agissant sur la classe ?

Un exemple de méthode Saisie qui permet la saisie au clavier et d'une méthode affichage qui affiche tou sles attributs d'un coup pour une Personne crée.

In [19]:

```
class Personne:
    """Classe définissant une personne caractérisée par :
        -son nom
        -son prenom
        -son age
        -sa ville"""

    compteur=0

    def __init__(self):#Notre methode constructeur
            """On définit les attributs un par un"""
            self.nom=""
            self.prenom=""
            self.age=0
            self.ville=""
            Personne.compteur += 1

    def saisie(self):#methode de saisie clavier
        """Methode de saisie au clavier"""
        self.nom=input("Nom :")
        self.prenom=input("Prenom :")
        self.age=input("Age :")
        self.ville=input("Ville :")

    def affichage(self): #method ed'afficchage de tous les attributs
        """Méthode d'affichage des attributs de l'objet"""
        print("Son nom est :",self.nom)
        print("Son prenom est :",self.prenom)
        print("Son age est :",self.age)
        print("Il/Elle habite a :",self.ville)
```

In [20]:

```
FD=Personne()
```

In [24]:

```
FD.saisie()
```

```
Nom :'Delebecque'
Prenom :'Fanny'
Age :33
Ville :'Paris'
```

In [25]:

```
print FD.nom
```

```
Delebecque
```

In [26]:

```
FD.affichage()
```

```
('Son nom est :', 'Delebecque')
('Son prenom est :', 'Fanny')
('Son age est :', 33)
('Il/Elle habite a :', 'Paris')
```

Et si je déménage ? Et si je fete mon anniversaire ?

In [27]:

```
class Personne:
    """Classe définissant une personne caractérisée par :
        -son nom
        -son prenom
        -son age
        -sa ville"""

    compteur=0

    def __init__(self):#Notre methode constructeur
            """On définit les attributs un par un"""
            self.nom=""
            self.prenom=""
            self.age=0
            self.ville=""
            Personne.compteur += 1

    def saisie(self):#methode de saisie clavier
        """Methode de saisie au clavier"""
        self.nom=input("Nom :")
        self.prenom=input("Prenom :")
        self.age=input("Age :")
        self.ville=input("Ville :")

    def affichage(self): #methode d'afficchage de tous les attributs
        """Méthode d'affichage des attributs de l'objet"""
        print("Son nom est :",self.nom)
        print("Son prenom est :",self.prenom)
        print("Son age est :",self.age)
        print("Il/Elle habite a :",self.ville)

    def demenage(self,newlieu): #methode demenagement
        self.ville=newlieu

    def anniversaire(self): #methode anniversaire
        self.age += 1
```

In [28]:

```
FD=Personne()
```

In [29]:

```
FD.saisie()
```

```
Nom :'Delebecque'
Prenom :'Fanny'
Age :34
Ville :'Mulhouse'
```

In [30]:

```
FD.demenage('Colomiers')
```

In [31]:

```
FD.affichage()
```

```
('Son nom est :', 'Delebecque')
('Son prenom est :', 'Fanny')
('Son age est :', 34)
('Il/Elle habite a :', 'Colomiers')
```

In [32]:

```
FD.anniversaire()
```

In [33]:

```
FD.affichage()
```

```
('Son nom est :', 'Delebecque')
('Son prenom est :', 'Fanny')
('Son age est :', 35)
('Il/Elle habite a :', 'Colomiers')
```

# Les méthodes spéciales

Les méthodes spéciales sont des méthodes que Python reconnaît et sait utiliser, dan scertains contextes. Elles peuvent servir à indiquer à Python ce qu'il doit faire s'il se trouve face à une expression comme

- mon_objet_1+mon_objet_2
- mon_objet[indice] Elles permettent de controler comment un objet se crée, ainsi qu el'acces à ses attributs

Une méthode spéciale voit son nom entouré d epart et d'autre par deux caractères "souligné" _ Le nom d'une telle méthode prend la forme __methodespeciale__

NB : Bien sur, le constructeur **intit** est une telle méthode spéciale !

- représentation : __repr__
- affichage "joli" : __str__
- objet[index] : accéder à un indice __getitem__
- objet[index]=valeur : modifier une valeur __setitem__
- objet in container : __contains__
- taille(objet) : __len__

Les méthodes mathématiques spéciales :

- __sub__ surcharge l'opérateur -
- __mul__ surcharge l'operateur +
- __truediv__ surcharge l'operateur /
- __mod__ surcharge l'operateur %
- __power__ surcharge l'operateur **
- __add__ surcharge l'operatuer +

on peut aussi surcharger +=, -=, *=, an ajoutant un i devant le nom habituel des surchages

De même on peut surcharger les opérateurs de comparaison

- __eq__ surcharge l'opérateur ==
- __ne__ surcharge l'opérateur !=
- __gt__ surcharge l'opérateur >
- __ge__ surcharge l'opérateur >=
- __lt__ surcharge l'opérateur <
- __le__ surcharge l'opérateur <=

L'exemple des complexes

On peut évidemment revenir sur la definition du type complexe à la lumière de ce que l'on vient de voir... et le redéfinir !

In [34]:

```
class Complexe:
    def __init__(self,r,i):
        self.re=r
        self.im=i
    def __add__(self,c):
        return Complexe(self.re+c.re,self.im+c.im)
    def __iadd__(self,c):
        self.re += c.re
        self.im += c.im
        return self
    def __str__(self):
        return str(self.re)+"+"+str(self.im)+"j"
```

In [35]:

```
c=Complexe(2,1)
print c
```

```
2+1j
```

In [36]:

```
str(c)
```

Out[36]:

```
'2+1j'
```

In [37]:

```
print c
```

```
2+1j
```

In [38]:

```
d=Complexe(0,2)
```

In [39]:

```
print c+d
```

```
2+3j
```

In [40]:

```
c+=d
print c
c-=d
```

```
2+3j

---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-40-091395a4bf2d> in <module>()
      1 c+=d
      2 print c
----> 3 c-=d

TypeError: unsupported operand type(s) for -=: 'instance' and 'instanc
e'
```

L'exemple des durées !

On fabrique une classe capable de contenir des durées exprimées en minutes et secondes, et de faire des opérations dessus.

In [41]:

```
class Duree:

    def __init__(self,min=0,sec=0):
        self.min=min
        self.sec=sec
    def __str__(self):
        return "{0:02}:{1:02}".format(self.min,self.sec)
```

In [42]:

```
d1=Duree(3,5)
```

In [43]:

```
print d1
```

```
03:05
```

In [44]:

```
d1+4
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-44-9f28c6c99e11> in <module>()
----> 1 d1+4

TypeError: unsupported operand type(s) for +: 'instance' and 'int'
```

In [45]:

```
class Duree:

    def __init__(self,min=0,sec=0):
        self.min=min
        self.sec=sec
    def __str__(self):
        return "{0:02}:{1:02}".format(self.min,self.sec)
    def __add__(self, objet_a_ajouter):
        """l'objet à ajouter est un entier : le nombre de secondes"""
        nduree=Duree()
        #on copie self dans nouvelle_duree
        nduree.min=self.min
        nduree.sec=self.sec
        #on ajoute la duree
        nduree.sec += objet_a_ajouter
        #si le nombre d esec est > 60
        if nduree.sec >= 60:
            nduree.min += nduree.sec//60
            nduree.sec = nduree.sec % 60
        return nduree
```

In [46]:

```
d1=Duree(12,8)
print d1
```

```
12:08
```

In [47]:

```
d2=d1+54
print d2
```

```
13:02
```