La programmation Orienté Objet (POO) en Python

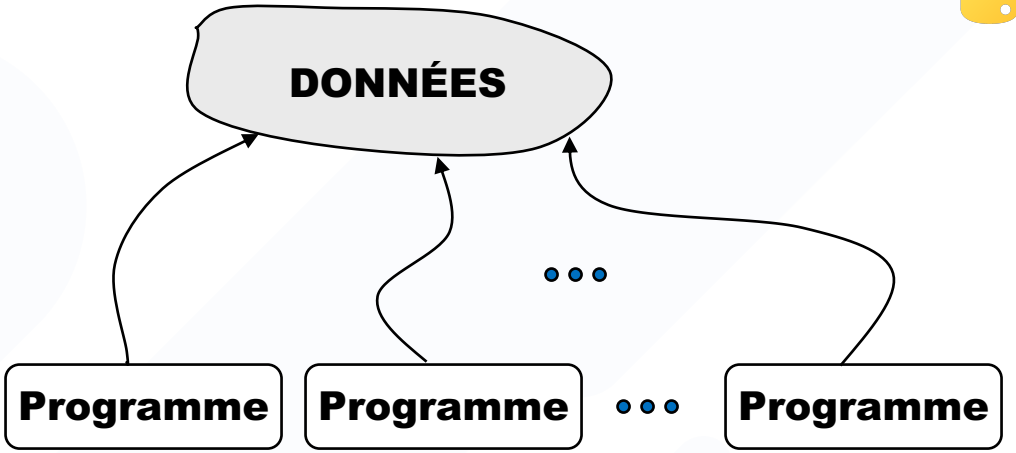
- Introduction à la POO
- Constructeur
- Attributs et méthodes
- Surcharge des opérateurs
- Héritage
- Polymorphisme

Évolution de la programmation

Programmation séquentielle :

Les instructions sont exécutées de façon linéaire.

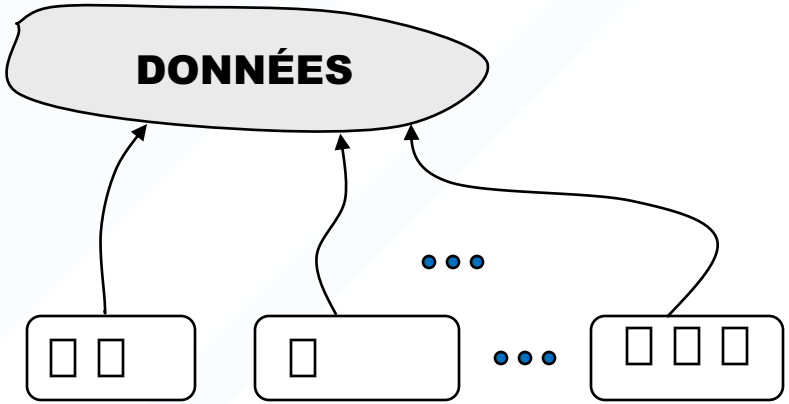
Les données sont globales et aucune réutilisation du code n'est réalisée.



Programmation procédurale :

Il s'agit de découper un programme en une série de fonctions. chaque fonction a pour but de réaliser un traitement particulier.

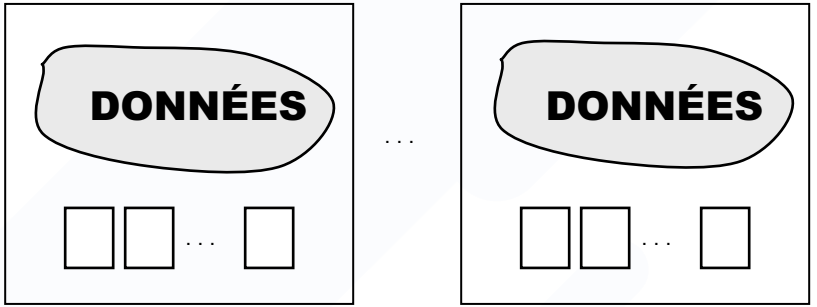
On procède à un découpage du traitement mais les données restent globales.



Évolution de la programmation

Programmation orientée objet :

Il s'agit de regrouper l'ensemble des données et les traitements qui s'appliquent à ces données. On procède à un découpage naturel des données et du traitement des données.



Exemple : Application chargée de la paye des employés d'une entreprise.

Créer une class « Employe » : toutes les données qui caractérisent un employé et les méthodes qui manipulent ces données,

POO en Python

- Une classe est équivalente à un nouveau type de données.
- On connaît déjà par exemple les classes list ou str et les nombreuses méthodes permettant de les manipuler, par exemple :

```
>>> l=[5,2,3] # l est une instance de la classe list
>>> l.sort()  # on fait appel à la méthode sort() de la classe list
>>> print(l)
[2, 3, 5]
```

- Un objet ou une instance est un exemplaire particulier d'une classe.

Les attributs et les méthodes

- On peut définir un objet comme une capsule contenant des attributs et des méthodes : **objet = attributs + méthodes**

Exemple :

- un objet de la classe complex possède :
  - deux attributs : imag et real;
  - plusieurs méthodes, comme conjugate(), abs()…

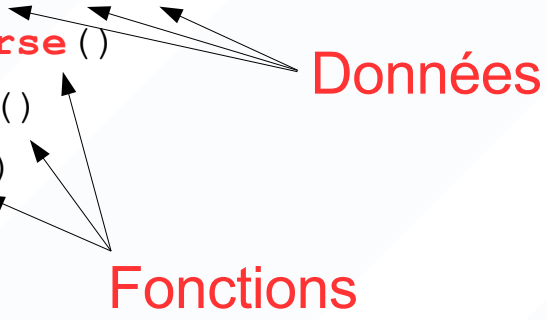
```
>>> a=3+2j # a est objet : c'est une instance de la classe complex
>>> print(a.real)  # on affiche l'attribut real de l'objet a
3.0
>>> print(a.imag)  # on affiche l'attribut imag de l'objet a
2.0
>>> print(a.conjugate())  # on appel la méthode conjugate() de l'objet a
(3-2j)
```

Les attributs et les méthodes

Les objets regroupent des **données** (attributs) et des **traitements** (méthodes) logiquement liées.

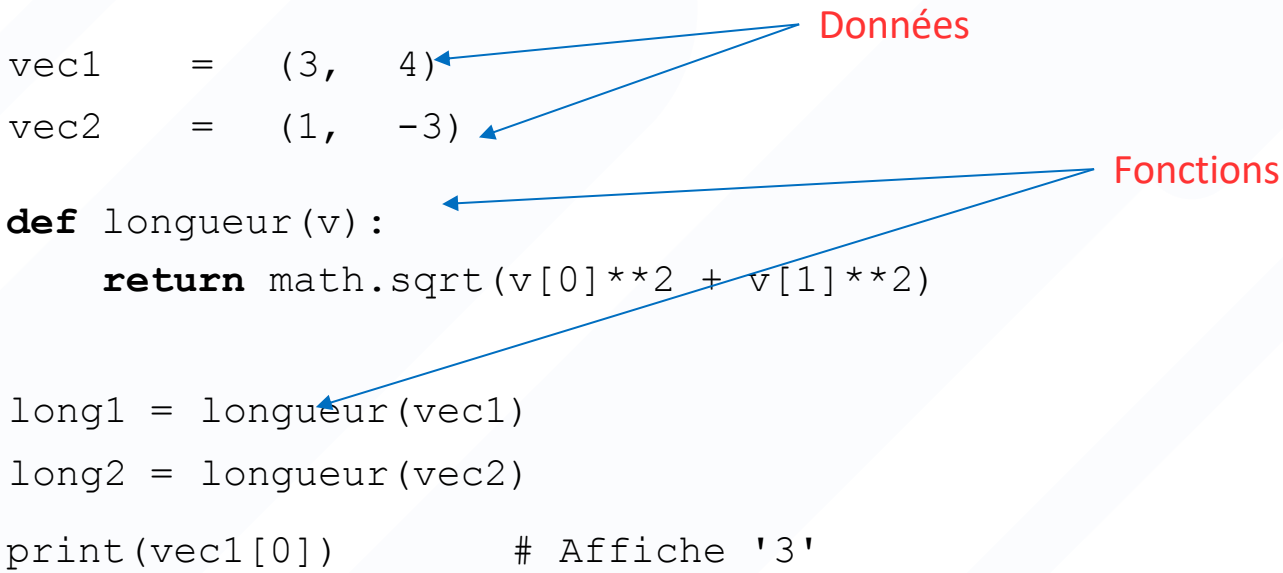
Par exemple, pour **objets** de type `list`

```
>>> nombres = [3, 1, 2]
>>> nombres.reverse()
>>> nombres.sort()
>>> nombres.pop()
```



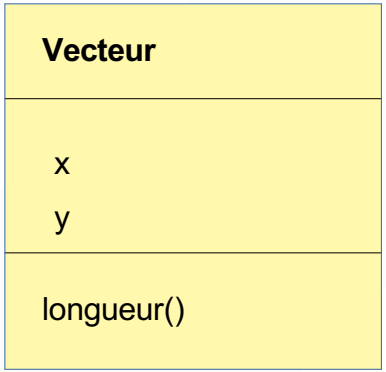
Classes

Paradigme procédural :



Classes

Paradigme OO : définir un **nouveau type `Vecteur`** pour grouper les données d'un vecteur et les fonction qui agissent dessus.



La classe **Vecteur** permet de représenter un vecteur dans le plan, composé de deux coordonnées x et y. Elle propose une méthode longueur permettant de calculer la norme du vecteur, qui vaut $\sqrt{x^2 + y^2}$

Classes

Paradigme OO : définir un **nouveau type `Vecteur`** pour regrouper les données d'un vecteur et les fonction qui agissent dessus.

```
vec1 = Vecteur(3,  4)  # création d'un objet Vecteur
vec2 = Vecteur(1,  -3) # création d'un objet Vecteur

long1 = vec1.longueur()
long2 = vec2.longueur()
print(vec1.x)          # affiche 3
print(vec1.y)          # affiche 4
```

Syntaxe de déclaration

- Les classes sont créées par mot-clé **class**.
- Les attributs sont les variables qui appartiennent à la classe.
- Les attributs sont toujours publiques et sont accessibles à l'aide de l'opérateur point (.)

L'instanciation

- Les classes sont des fabriques d'objets : on construit d'abord l'usine avant de produire des objets!
- On instancie un objet (c'est-à-dire qu'on le produit à partir de l'usine) en appelant le nom de sa classe comme s'il s'agissait d'une fonction

Exemple : déclaration des attributs et instanciation

```
class Vecteur:
    """ classe pour gérer les vecteurs """
    def __init__(self,a=0,b=0):
        self.x=a
        self.y=b
    def affichage (self):
        print( "(" + str(self.x) + ", " + str(self.y) + ")")

##########################################
v1=Vecteur()
v2=Vecteur(2,5)
v1.affichage()
v2.affichage()
```

```
(0, 0)
(2, 5)
```

Constructeur

- Les constructeurs sont généralement utilisés pour instancier un objet. La tâche des constructeurs consiste à initialiser (attribuer des valeurs) aux attributs lorsqu'un objet de la classe est créé.
- En Python, la méthode **__init __ ()** est appelée le constructeur
- la méthode **__init __ ()** est exécuté dès qu'un objet d'une classe est instancié. Cette méthode est utile pour effectuer l'initialisation que vous voulez faire avec votre objet
- En python, on ne peut définir qu'un seul constructeur
- La méthode __init__ est similaire aux constructeurs en C ++ et Java.

Constructeur

Syntaxe

```
1  def __init__(self, liste_parameters):
2      # Initialisiation
```

Exemple

```
class Time:
    " une nouvelle classe temporelle"
    def __init__(self, hh =0, mm =0, ss =0):
        self.heure = hh
        self.minute = mm
        self.seconde = ss


##########################################
t1=Time()
print("t1 = ",t1.heure, t1.minute,t1.seconde)
t2=Time(10,15,13)
print("t2 = ", t2.heure, t2.minute,t2.seconde)
```

```
t1 =  0 0 0
t2 =  10 15 13
```

Déclaration des méthodes

- Une méthode est un groupe de code destiné à effectuer une tâche particulière.
- Une méthode s'écrit comme une fonction du corps de la classe avec un premier paramètre **self** obligatoire, où self représente l'objet sur lequel la méthode sera appliquée. Autrement dit self est la référence d'instance.
- **'self'** est l'équivalent du mot-clé **'this'** utilisé pour l'objet en cours dans les autres langages OO.
- Le mot clé **'def'** est utilisé pour créer une nouvelle méthode.

Exemple : déclaration des attributs et instanciation

```
from math import *
class Vecteur:
    """ classe pour gérer les vecteurs """
    def __init__(self,a,b):
        self.x=a
        self.y=b
    def longeur(self):
        return str(sqrt(self.x**2+self.y**2))
##########################################
v=Vecteur(2,3)  # on crée un objet Vecteur
l=v.longeur()   # on appelle la méthode longeur
print(l)
```

Variable d'instance

- Un attribut est représenté par une variable particulière, appelée ***variable d'instance***, accessible à l'aide du paramètre **self**. L'idée est que le constructeur initialise ces variables d'instances, qui seront alors stockées dans l'objet et en mémoire pour toute la durée de vie de l'objet
- La classe Vecteur: x et y sont des ***variables d'instances***

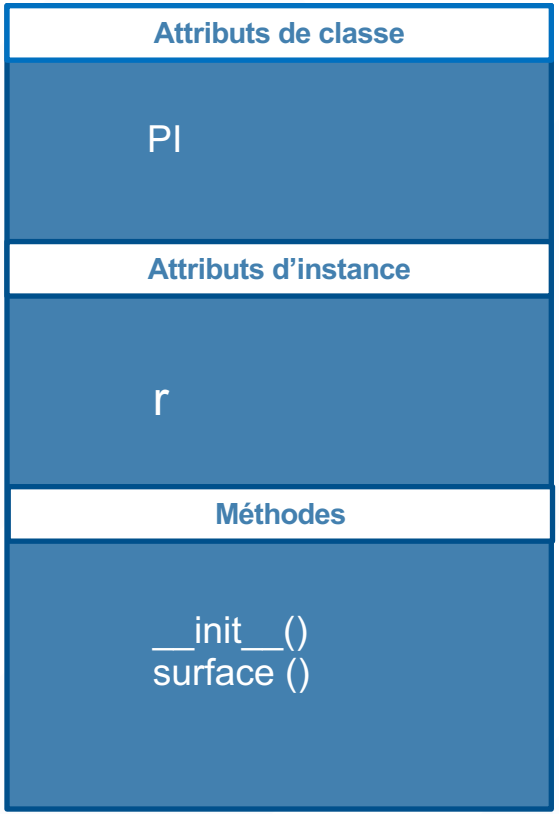
Variable de classe = attributs de classe

On peut définir des attributs pour les classes aussi (*class attributes* = **attributs de classe**), par affectation. Ils sont en commun pour toutes les objet crée par la classe. Souvent utilisés pour les « constantes » du type.

```
class Cercle:
    """ Classe pour gérer les cercles """
    PI = 3.14159 #attribut de la classe
    def __init__(self, r):
        self.rayon = r
    def surface(self):
        return Cercle.PI * self.rayon ** 2

                        self.PI
#####################################################
c = Cercle(3)
print('La valeur de PI est ' , Cercle.PI )
print('La valeur de PI est ' , c.PI )
print('le rayon est ', c.rayon)
print('la surface est ', c.surface())
```

```
La valeur de PI est  3.14159
La valeur de PI est  3.14159
le rayon est  3
la surface est  28.27431
```

Variable de classe = attributs de classe



Via __init__

hiérarchie des espaces de noms

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelés **variables de classe** ou attributs de classe.
- Chaque objet instance (crée à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées variables d'instance ou **attributs d'instance.**
- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal.

Exemple 1

```
class Cercle:
    """ Classe pour gérer les cercles """
    PI = 3.14159 #attribut de la classe
    def __init__(self, r):
        self.rayon = r
    def surface(self):
        return Cercle.PI * self.rayon ** 2
    def modifier_PI(self):
        self.PI=3.14

########################################
c1 = Cercle(3)
print('La valeur de PI est ' , Cercle.PI )
print('La valeur de PI est ' , c1.PI )
c1.modifier_PI()
c2 = Cercle(2)
print(c1.PI)
print(c2.PI)
print(Cercle.PI)
```

```
La valeur de PI est  3.14159
La valeur de PI est  3.14159
3.14
3.14159
3.14159
```

Exemple 2

```python
class Cercle:
    """ Classe pour gérer les cercles """
    PI = 3.14159  # attribut de la classe

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

    def surface(self):
        return Cercle.PI * self.rayon ** 2

    def modifier_pi(self):
        Cercle.PI = 3.14


################################################
c = Cercle(3)
print('La valeur de PI est ', Cercle.PI)
print('La valeur de PI est ', c.PI)
print('le rayon est ', c.rayon)
print('la surface est ', c.surface())
c.modifier_pi()
print(c.PI)
print(Cercle.PI)
```

```
La valeur de PI est  3.14159
La valeur de PI est  3.14159
le rayon est  3
la surface est  28.27431
3.14
3.14
```

Les attributs de classe sont souvent utilisés pour créer des données ou des actions globales à la classe, qui **ne dépendent pas** d'une instance.

Les instances peuvent cependant y accéder.

Modifier un attribut de classe dans une instance le modifiera dans **toutes les autres** !

Exemple 3

```
class Cercle:
    """ Classe pour gérer les cercles """
    PI = 3.14159 #attribut de la classe
    nb=0
    def __init__(self, r):
        self.rayon = r
        Cercle.nb += 1
    def surface(self):
        return Cercle.PI * self.rayon ** 2

##########################################
c1 = Cercle(3)
c2=Cercle(1)
c3=Cercle(1)
print("Nombre d'objets Cercle : ", Cercle.nb)
print("Nombre d'objets Cercle : ", c1.nb)
print("Nombre d'objets Cercle : ", c2.nb)
```

```
Nombre d'objets Cercle :  3
Nombre d'objets Cercle :  3
Nombre d'objets Cercle :  3
```

Méthodes de classe

- Tout comme il est possible de déclarer des attributs de classe, il est également possible de déclarer des méthodes de classe. Pour cela, on utilise le décorateur @classmethod. Comme une méthode de classe appartient à une classe, le premier paramètre correspond à la classe. Par convention, on appelle ce paramètre **cls** pour préciser qu'il s'agit de la classe et pour le distinguer de **self**.

Exemple

```python
class Cercle:
    """ Classe pour gérer les cercles """
    PI = 3.14159  # attribut de la classe
    nb_instance=0        # attribut de la classe

    def __init__(self, r):
        self.rayon = r
        Cercle.nb_instance+=1
    @classmethod
    def reinitialiser_compteur(cls):
        cls.nb_instance = 0

    def surface(self):
        return Cercle.PI * self.rayon ** 2

##############################################
c1 = Cercle(2)
c2 = Cercle(1)
c3=Cercle(2)
print("Nombre d'instances de la classe cercle :" ,Cercle.nb_instance)
Cercle.reinitialiser_compteur()
print("Nombre d'instances après l'appel de la méthode ",Cercle.nb_instance)
```

Nom de classe et nom d’attributs: Conventions

Quelques conventions.

- **Nom des classes** : en « **CamelCase** »

Exemple : Vecteur, PlanCartesien , Matrice

- **Nom des objet** : en minuscule (avec tirets bas)

```
vec = Vecteur(1, 2)

plan_cartesien = PlanCartesien()

main_win = MainWindow()
```

SURCHARGE DES OPÉRATEURS

Définition

- La surcharge d’opérateurs vous permet de redéfinir la signification d’opérateur en fonction de votre classe.
- La surcharge d’opérateurs nous a permis d’utiliser l’opérateur **+** pour faire l’addition de deux objets numériques, ainsi que pour concaténer deux objets chaîne. C’est un seul opérateur, mais qui agit différemment selon le type d’objet sur lequel il est appelé

Problème

```
class Time:
    " une nouvelle classe temporelle"
    def __init__(self, hh =0, mm =0, ss =0):
        self.heure = hh
        self.minute = mm
        self.seconde = ss
##########################################
t1=Time(1,40,55)
t2=Time(13,35,10)
t3=t1+t2
```

```
Traceback (most recent call last):
  File "/Users/oumaira/Documents/Python 2019/Orienté objet en pyt
10, in <module>
    t3=t1+t2
TypeError: unsupported operand type(s) for +: 'Time' and 'Time'
```

Solution

- La surcharge de l'opérateur est obtenue en définissant **une méthode spéciale** dans la définition de classe. Les noms de ces méthodes commencent et finissent par un double soulignement (__).

Opérateur +

- Pour surcharger l'opérateur +, nous devrons implémenter la fonction

__add __ () dans la classe

```
class Time:
    " une nouvelle classe temporelle"
    def __init__(self, hh =0, mm =0, ss =0):
        self.heure = hh
        self.minute = mm
        self.seconde = ss
    def __add__(self, other):
        somme = Time()
        somme.heure = self.heure + other.heure
        somme.minute = self.minute + other.minute
        somme.seconde = self.seconde + other.seconde
        if somme.seconde >= 60:
            somme.seconde -= 60
            somme.minute += 1
        if somme.minute >= 60:
            somme.minute -= 60
            somme.heure += 1
        return somme
##########################################
t1=Time(1,40,55)
t2=Time(13,35,10)
t3=t1+t2
print(t1.heure, t1.minute,t1.seconde)
print(t2.heure, t2.minute,t2.seconde)
print(t3.heure, t3.minute,t3.seconde)
```

```
1 40 55
13 35 10
15 16 5
```

Exemple 2

```
from math import *
class Vecteur:
    """ classe pour gérer les vecteurs """
    def __init__(self,a,b):
        self.x=a
        self.y=b
    def __add__(self,other):
        v=Vecteur(0,0)
        v.x=self.x+other.x
        v.y=self.y+other.y
        return v
```

Méthodes spéciales

+	__add__
-	__sub__
*	__mul__
/	__truediv__
//	__floordiv__
<	__lt__ (less than)
>	__gt__ (greater than)
<=	__le__ (less or equal)
>=	__ge__ (greater or equal)
==	__eq__
	...

Surcharge des opérateurs de comparaison

```
from math import *
class Vecteur:
    """ classe pour gérer les vecteurs """
    def __init__(self,a,b):
        self.x=a
        self.y=b
    def __add__(self,other):
        v=Vecteur(0,0)
        v.x=self.x+other.x
        v.y=self.y+other.y
        return v
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
##############################################
v1=Vecteur(2,3)  # on crée un objet Vecteur
v2=Vecteur(1,5)
v3=Vecteur(2,3)
print(v1==v2)
print(v1==v3)
```

```
False
True
```

Surcharge de la méthode print

```
from math import *
class Vecteur:
    """ classe pour gérer les vecteurs """
    def __init__(self,a,b):
        self.x=a
        self.y=b
    def __add__(self,other):
        v=Vecteur(0,0)
        v.x=self.x+other.x
        v.y=self.y+other.y
        return v
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
##########################################
v1=Vecteur(2,3)  # on crée un objet Vecteur
v2=Vecteur(1,5)
v3=Vecteur(2,3)
print("(" + str(v3.x) + ", " + str(v3.y) + ")")
print(v3)
```

```
 RESTART: /Users/oumaira/Documents/Python 2019/Orienté objet en python/vecteur0.py
(2, 3)
<__main__.Vecteur object at 0x7fcee8f74320>
```

Remarque

- Ce comportement par défaut n'est pas forcément des plus utiles. Ce qui serait plus intéressant serait de pouvoir obtenir une *représentation* de l'objet sous forme d'une chaine de caractères.
- En Python, il suffit d'ajouter une méthode nommée **__str__** sans paramètre (à part l'obligatoire self puisque c'est une méthode) qui renvoie une chaine de caractères.

Exemple

```
from math import *
class Vecteur:
    """ classe pour gérer les vecteurs """
    def __init__(self,a,b):
        self.x=a
        self.y=b
    def __add__(self,other):
        v=Vecteur(0,0)
        v.x=self.x+other.x
        v.y=self.y+other.y
        return v
    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"

##########################################
v1=Vecteur(2,3)  # on crée un objet Vecteur
v2=Vecteur(2,3)
v3=v1+v2
print(v3)
```

```
(4, 6)
```

L’encapsulation

Le concept d'encapsulation en POO

- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.
- L’un des grands intérêts des classes est qu’elles permettent l’encapsulation du code, c’est-à-dire le fait d’enfermer le code dans une “capsule”, dans un espace en dehors de l’espace global. Cela permet d’éviter d’utiliser des variables globales et de polluer l’espace global du script.

Les trois niveaux de visibilités

Ils définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque. Il existe trois niveaux de visibilité:

- publique: les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du plus bas niveau de protection des données
- protégée: l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées
- privée: l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

Les attributs privés en Python ?

- Il n'y a pas d'attributs privés en Python, tout est toujours accessible.
- Une convention est de préfixer avec un ou 2 tirets bas les attributs qu'on veut protéger. Mais c'est juste une « suggestion » aux programmeurs.

les conventions sont les suivantes :

- On préfixera les noms des membres qu'on souhaite définir comme "privés" avec deux underscores comme ceci : __nom-du-membre ;
- On préfixera les noms des membres qu'on souhaite définir comme "protégés" avec un underscore comme ceci : _nom-du-membre.

Exemple

```
class test:
    a="Variable Public"
    _b="Variable Protégée"
    __c="Variable Privée"

t=test()
print(t.a)
print(t._b)
print(t.__c)
```

```
Variable Public
Variable Protégée
Traceback (most recent call last):
  File "/Users/oumaira/Documents/Python 2019/Orienté
py", line 9, in <module>
    print(t.__c)
AttributeError: 'test' object has no attribute '__c'
```

Mais il y a un moyen d'y accéder !!!

```
class test:
    a="Variable Public"
    _b="Variable Protégée"
    __c="Variable Privée"

t=test()
print(t.a)
print(t._b)
print(t._test__c)
```

```
Variable Public
Variable Protégée
Variable Privée
```

Guido van Rossum : "We're all consenting adults here"

L’encapsulation

- L’idée de la notion d’encapsulation consiste à ce que :
  - une classe définit son interface, c’est-à-dire les méthodes par lesquelles on peut utiliser ce code,
  - mais reste tout à fait libre de modifier son implémentation, et tant que cela n’impacte pas l’interface, aucun changement n’est requis dans les codes utilisateurs.

La classe matrice : version 1

```
class Matrice:
    "Une implémentation sommaire de matrice carrée 2x2"
    def __init__(self,a11,a12,a21,a22):
        self.a11=a11
        self.a12=a12
        self.a21=a21
        self.a22=a22
    def determinant(self):
        "renvoie le déterminant de la matrice"
        return self.a11 * self.a22 - self.a12 * self.a21

##################################################
M=Matrice(1,3,4,9)
print("le determinant de la matrice est" ,M.determinant())
```

La classe matrice : version 2

```
class Matrice2:
    "Une implémentation sommaire de matrice carrée 2x2"
    def __init__(self,a11,a12,a21,a22):
        self.a = (a11, a12, a21, a22)
    def determinant(self):
        "renvoie le déterminant de la matrice"
        return self.a[0] * self.a[3] - self.a[1] * self.a[2]

##################################################
M=Matrice2(1,3,4,9)
print("le determinant de la matrice est" ,M.determinant())
```

Conclusion

Grâce à l’encapsulation, on peut continuer à utiliser la classe exactement de la même manière

L'encapsulation consiste à cacher les détails d'implémentation d'une classe, et ne pas les dévoiler en dehors de cette dernière. Le code qui utilise une classe doit être le plus indépendant possible de la manière avec laquelle la classe est implémentée.

L'HÉRITAGE EN PYTHON

Définition

- L'héritage est un mécanisme qui nous permet de créer une nouvelle classe (classe enfant) qui est basée sur une classe existante (la classe mère), en ajoutant de nouveaux attributs et méthodes en plus de la classe existante.
- Lorsque vous procédez ainsi, la classe enfant hérite des attributs et méthodes de la classe parent.
- L’héritage permet de construire une hiérarchie de classes. Les classes enfants héritent des champs et méthodes de la classe mère

Syntaxe

Syntaxe :

```
1  class mere:
2      # corps de la classe mère
3
4  class enfant(mere):
5      # corps de la classe enfant
```

Remarque

En Python toutes les classes héritent de la classe object "**class mere(object)**" et "**class mere**" sont identiques.

Exemple

```
Nom : Mansouri
Prénom : Ziad
Salaire : 9000
 Nom : Mansouri
 Prénom : Ziad
```

```python
class Personne:
    # constructeur de la classe
    def __init__(self):
        self.nom=""
        self.prenom=""

    def saisie(self):
        self.nom = input("Nom : ")
        self.prenom=input("Prénom : ")

    def affichage(self):
        print (" Nom : "+ self.nom +" \n Prénom : "+ self.prenom )

class Employe(Personne):
    def __init__(self):
     # appeler __init__ de la classe mère (Personne)
        super().__init__()
        self.salaire=0
    def saisie(self):   # surcharge de la méthode
         Personne.saisie(self)
         self.salaire=float(input("Salaire : "))
####### instanciation de la classe
e=Employe()
e.saisie()
e.affichage()
```

Résolution des attributs

- l'instance hérite de sa classe, et la classe va hériter de ses super-classes et les super-classes peuvent elles-mêmes hériter de leurs propres super-classes.

→ Notion d'arbre d'héritage

--> Si on cherche un attribut dans une instance, si on ne le trouve pas dans l'espace de nommage de l'instance, on remonte l'arbre d'héritage et on arrive dans la classe. Cette notion d'arbre d'héritage s'étend également aux classes. Les classes peuvent hériter d'autres classes et on peut donc avoir un arbre d'héritage qui va partir des instances, remonter aux classes, et remonter aux classes dont héritent ces classes

Héritage Multiple

- Python supporte l'héritage Multiple

Syntaxe

```
1   Class mere_1:
2       # corps de mere_1
3
4   Class mere_2:
5       # corps de mere_2
6
7   Class mere_3:
8       # corps de mere_3
9
10  Class enfant(mere_1, mere_2, mere_3):
11      # corps de la classe enfant
```

- La classe enfant est dérivée de trois classes mere_1, mere_2, mere_3. En conséquence, il héritera des attributs et des méthodes des trois classes.

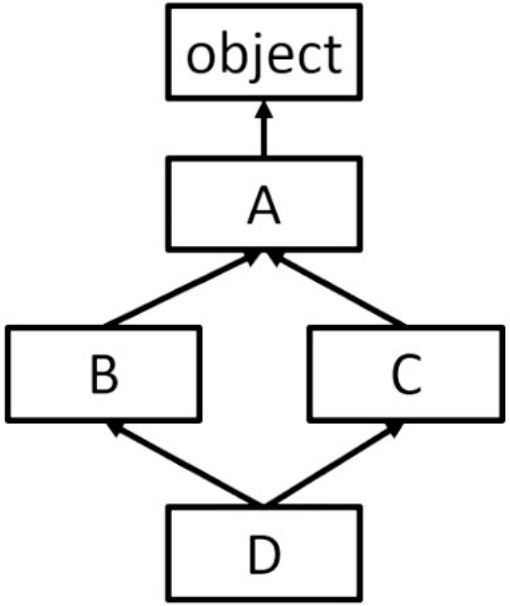
La MRO : Method Resolution Order (méthode de résolution d'attribut)

```
class SuperA:
    pass

class SuperB:
    pass

class C(SuperA,SuperB):
    pass

print(C.__mro__) #  __mro__ est un attribut special de classe.
                 # Il doit donc etre recupere depuis la classe
```

```
(<class '__main__.C'>, <class '__main__.SuperA'>, <class '__main__.SuperB'>,
<class 'object'>)
```

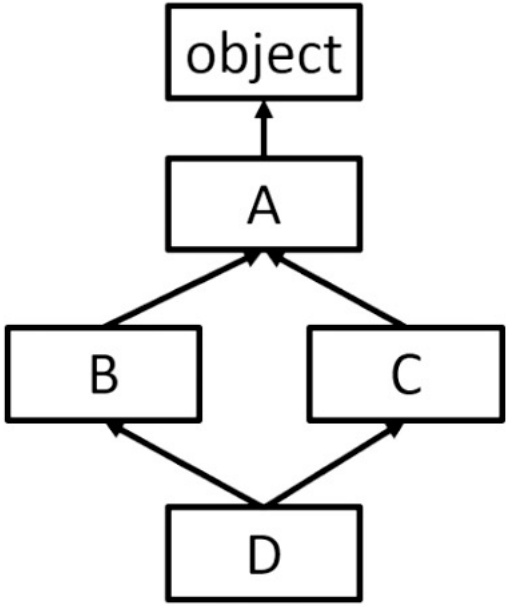
L’algorithme de résolution d’attribut

```
>>> class A: pass
>>> class B(A): pass
>>> class C(A): pass
>>> class D(B, C): pass
```



L’algorithme de résolution d’attribut

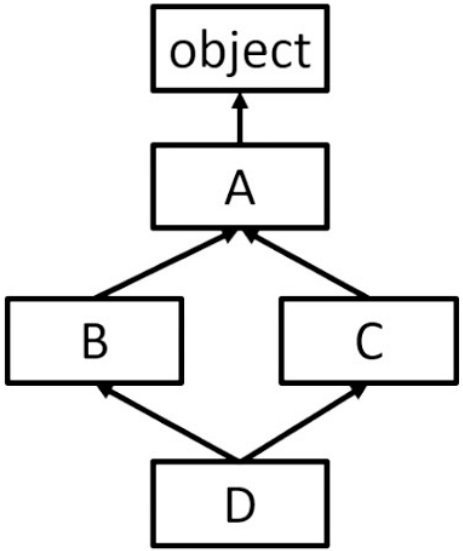
```
>>> class A: pass
>>> class B(A): pass
>>> class C(A): pass
>>> class D(B, C): pass
>>> D.mro()
```




```
D, B, A, object, C, A, object
```

L’algorithme de résolution d’attribut

```
>>> class A:pass

>>> class B(A):pass

>>> class C(A):pass

>>> class D(B,C):pass

>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__
main__.C'>, <class '__main__.A'>, <class 'object'>]
```



D, B, ~~A~~, ~~object~~, C, A, object

Polymorphisme

Définition

- Le mot **polymorphisme** signifie avoir plusieurs formes. En programmation, polymorphisme signifie que le même nom de fonction (mais des signatures différentes) est utilisé pour différents types.
- En Python, **Polymorphisme** nous permet de **définir** des méthodes dans la classe enfant portant le même nom que les méthodes de la classe mère.
- En héritage, la classe enfant hérite des méthodes de la classe mère. Toutefois, il est possible de modifier une méthode dans une classe enfant héritée de la classe mère. Ceci est particulièrement utile dans les cas où la méthode héritée de la classe mère ne convient pas tout à fait à la classe enfant. Dans ce cas, nous ré-implémentons la méthode dans la classe enfant. Ce processus de ré-implémentation d'une méthode dans la classe enfant s'appelle une **méthode redéfinie**.

Exemple 1

```
>>> l=[1,2,3]
>>> t=(1,2,3)
>>> chaine="123"
>>> len(l)
3
>>> len(t)
3
>>> len(chaine)
3
>>>
```

La même méthode **len** est appelée sur des objets de type différents

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

    def affiche(self):
        print("je suis une personne")
class Etudiant(Personne):
    def __init__(self,nom,cne):
        super().__init__(nom)
        self.cne=cne

    def affiche(self):
        print("je suis un étudiant")
class Professeur(Personne):
    def __init__(self,nom,ppr):
        super().__init__(nom)
        self.ppr=ppr

    def affiche(self):
        print("je suis un professeur")
########################################@
objet1=Etudiant('lina',123444)
objet2=Professeur('oumaira',123123123)
objet1.affiche()
objet2.affiche()
```

```
je suis un étudiant
je suis un professeur
```

Créer une classe dans un module

main.py | ModulePersonne.py

```python
class Personne:

#constructeur
    def __init__(self):
        #lister les champs
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
#fin constructeur
#saisie des infos
    def saisie(self):
        self.nom = input("Nom : ")
        self.age = int(input("Age : "))
        self.salaire = float(input("Salaire : "))
#fin saisie
#affichage des infos
    def affichage(self):
        print("Son nom est ", self.nom)
        print("Son âge : ", self.age)
        print("Son salaire : ", self.salaire)
#fin affichage
#fin définition
```

```python
import ModulePersonne as MP

# instanciation
p = MP.Personne()
# saisie
p.saisie()
# méthode affichage
p.affichage()

```

```
Nom : Zaimi
Age : 24
Salaire : 11000
Son nom est  Zaimi
Son âge :  24
Son salaire :  11000.0
```

COLLECTION D'OBJETS

Python propose des outils pour la gestion des collections d'objets hétérogènes (tuple, liste, dictionnaire). Ils sont opérationnels pour les instances de nos classes.

```
# -*- coding: utf -*-

#appel du module
import ModulePersonne as MP

#liste vide
liste = []

#nb. de pers ?
n = int(input("Nb de pers : "))

#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)

#affichage
print("*** début affichage 1")
for p in liste:
    print("-------")
    p.affichage()
```

Créer l'objet référencé par a, effectuer la saisie. Ajouter la référence dans la liste.

Le typage est automatique, p est bien de type Personne.

Exemple d'exécution

```
Nb de pers : 2
Nom : Toto
Age : 35
Salaire : 1200
Nom : Tata
Age : 36
Salaire : 3000
*** début affichage 1
-------
Son nom est  Toto
Son âge :  35
Son salaire :  1200.0
-------
Son nom est  Tata
Son âge :  36
Son salaire :  3000.0
```