



Python

Cours : les bases

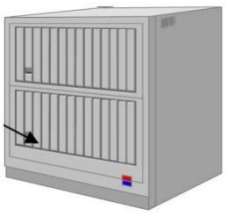


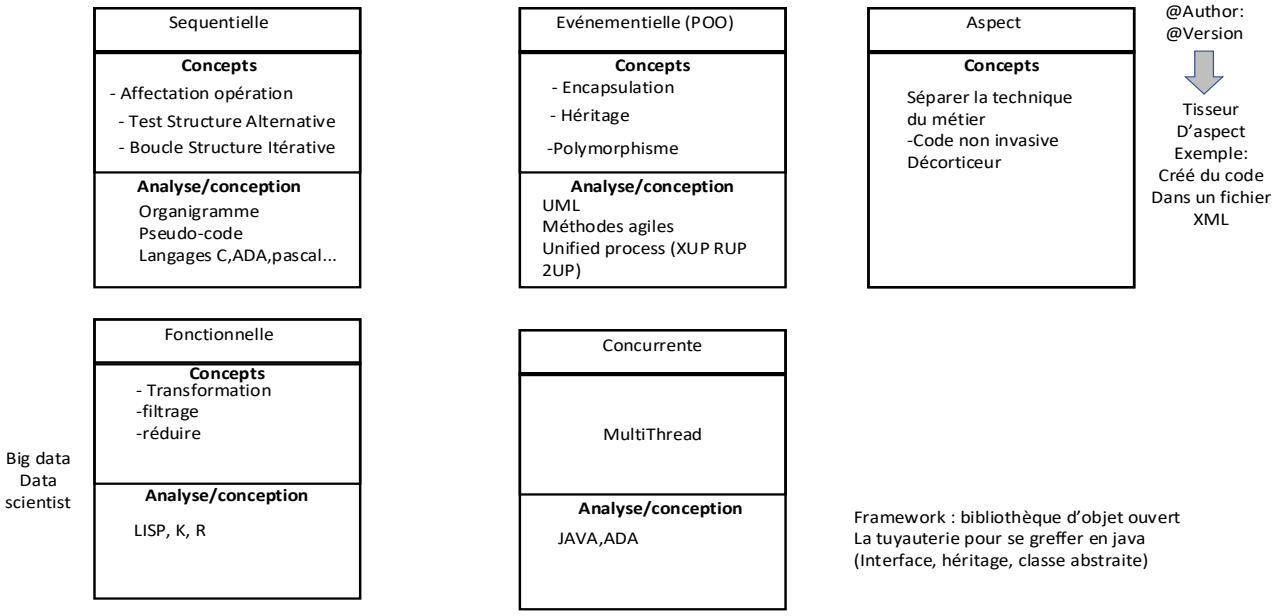
Table des matières

1 Introduction 4
Les cinq paradigmes de la programmation 4
2 Les scripts et les programmes 4
3 Les variables ou objet en camelcase 5
4 Les scripts en python exemple simple sur IDLEX.exe 5
5 Les opérateurs de base conditionnelles ou non 6
5.1 La priorité 6
5.2 Les affectations 6
5.3 Opérateurs d'auto affectation et binaires 6
5.4 Affectation multiple simultanée 7
5.5 Les opérateurs de relation 7
5.6 L'opérateur d'identité is 8
5.7 Opérateur d'appartenance in 8
5.8 Les commentaires 8
5.9 L'opérateur * et ** 8
6 Objet non mutable (objet qui ne peut pas changer d'état) 9
7 Les instructions en python 10
7.1 Structure alternatives ou de test : 10
7.2 Structures itératives ou boucles 11
7.3 L'instruction with as 13
8 Les fonctions de saisie et d'affichage en console 14
9 Les expressions régulières sur les chaines (patron ou pattern) 15
10 Les collections ou les structures de données 18
10.1 Les tuples 18
10.2 Les Listes 21
10.3 Les dictionnaires 26
11 Les listes, les dictionnaires les tuples en compréhension 29
12 Les fonctions 31
12.1 La fonction numérique simplifiée avec l'opérateur lambda 31
12.2 La fonction universelle 31
12.3 Nombre de paramètre non fixés *args et **kwargs 32
13 Les décorateurs 35
13.1 Le principe 35
13.2 Le décorateur @classmethod 35

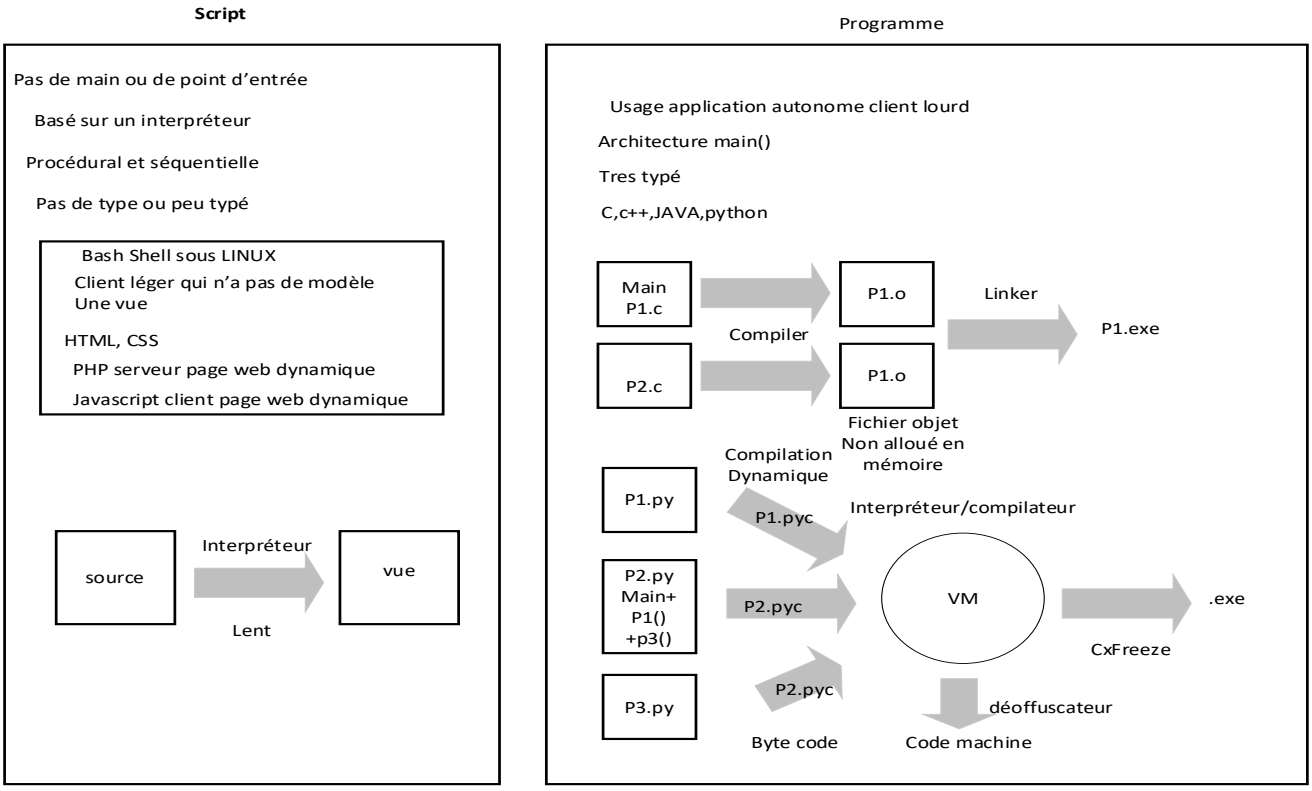
13.3 Le décorateur @staticmethod 36
13.4 Le décorateur @property, @getter, @setter 36
14 Les modules ou Les bibliothèques 38
15 Le programme et la fonction principale 38
16 Les paquetages 40
17 Les Classes 42
18 L'Héritage 48
19 La surcharge de méthodes en Python (Overloading) 49
20 Les modèles de conception (pattern design) pour les classes 51
20.1 La classe singleton 51
20.2 les décorateurs de classe 51
20.3 Les observateurs de classe 52
21 méthodes dunder 53
22 L'héritage multiple des classes en Python 56
23 Les Exceptions 61
24 La sérialisation 63
25 Les fonctions génératrices et les iterables 64
25.1 les fonctions iter() et next() 64
25.2 Les fonctions génératrices avec yield 64
26 Les fonctions les plus usuels en python 66
27 Les principaux formats de fichiers de données et de configuration 69
27.1 Le format XML 69
27.2 Le format CSV 69
27.3 Le format INI 69
27.4 Le format JSON 69
27.5 Le format YAML 69

1 Introduction

Les cinq paradigmes de la programmation



2 Les scripts et les programmes



2013 : version 3 beaucoup + professionnelle
2022 : version 3.10.5 package pip équivalent à apt-get
Dépôt repository officielle PYpi
La PEP permet de vérifier la syntaxe (python enhancement proposal)

3 Les variables ou objet en camelcase

maVariable
snake case : ma_super_variable

Les constantes

MACSTE

Une classe

MaClasse

Les packages

Courbe graphisme -→ Matplotlib
Calcul scientifique → numpy
Stat → scipy
Big data → pandas ML
Traitement d'image → Pil
Vision → Singlecv
GUI → TKinder wxpython pyqt

Workspace IDE
Anaconda
Winpython avec Spyder

4 Les scripts en python exemple simple sur IDLEX.exe

Math est une classe
>>> import math
>>>help(math)
>>>dir(math)
>>>math.__doc__
>>>4/3 donne un réel
>>>8/3 partie entière de la division
>>>4%3 modulo reste de la division
>>>4**3 puissance

5 Les opérateurs de base conditionnelles ou non

+,-,*,/,%,//,,<,>,<=,>=,==, !=,and,or,is,not**

5.1 La priorité

PEMDAS parenthese→ exposant→ multiplication →division→addition→sosutraction

Opérateur particulier le is → (est ce que c'est), opérateur is === égalité de valeur de type et d'adresse

5.2 Les affectations

En python pas de typage statique mais dynamique

>>>maVariable='4' ou maVariable= "4" → chaine de caractère ' ' ou " "

pour obtenir le type :

>>>type(maVariable)
>>>maVariable=3.5 → float
>>>maVariable=4 → entier
>>>maVariable=True ou False → booléen
>>>maVariable=4+3j → complexe
>>>maVariable=(4,5) → collection de type tuple
>>>0.1+0.1+0.1-0.3 → me donne 5.551115123125783e-17 résolution des float
>>> ((0.7+0.1)*10) → me donne 7.99999999 résolution des float

Affectation multiple

>>>V1=V2=4
>>>V1,V2=(4,4) affectation d'un tuple en V1et V2
>>> not True
>>> "45"*3 surcharge des opérateurs me donne "454545"
>>> "45" + str(3) me donne "453" en chaine
number = '12345'
result = int(number) + 10
print(result)
>>> exit() pour sortir de l'éditeur
>>> 2**3 → ** opérateur de puissance
8
>>> 2==3 → les booléens
False

5.3 Opérateurs d'auto affectation et binaires

+=		x += 2
-=		x -= 2
*=		x *= 2 x = x * 2
/=	La divivion	x /= 2
%=	reste de la division	x %= 2
//=	La divivion entière	x //= 2
**=	La puissance	x **= 2 x = x ** 2
&=	Et logique	x &= 2

|=	Ou logique	x |= 2
^=	Ou exclusif	x ^= 2
>>	Décalage à droite	x >>= 2
<<	Décalage à gauche	x <<= 2
&	Et logique	
|	Ou logique	
^	xor	
~	Complément à 1	
>>	Décalage à droite	
<<	Décalage à gauche	

5.4 *Affectation multiple simultanée*

```
a,b=b,a  # permutation
a,b=b,a+b
a=2
b=3
a,b=b,a+b
print(a,b) # donne 3 et 5 affectation simultanée
id(a)                    # donne l'adresse de a
```

Pour afficher le type d'une variable on utilise la méthode type()

type(a)

Longue chaîne de caractères

s1 = """Hello! My name is Nicolas

I wrote a text with lot of lines.

I can write everything here! I'm free with special char \ """

print(s1)

5.5 *Les opérateurs de relation*

Les opérateurs de comparaison (ou relationnels) sont utilisés pour comparer les valeurs de deux opérandes.
L'expression s'écrit sous la forme d'une relation (de comparaison) : elle vaut True si les valeurs des opérandes vérifient l a relation et False sinon.

==	égalité
!=	différent
>	supérieur
>=	Supérieur ou égal
<	inférieur
<=	Inférieur ou egal
and	Et booléen
or	Ou booléen
not	Complément booléen

5.6 L'opérateur d'identité is

Il vérifie s'il s'agit du même objet en mémoire

a = [1, 2, 3] b = [1, 2, 3]
print(a == b) # True vérifie si les valeurs sont identiques
print(a is b) # False

5.7 Opérateur d'appartenance in

in et not in sont les opérateurs d'appartenance en Python. Ils sont utilisés pour vérifier si une valeur est présente (ou non) dans une séquence (chaîne, liste, tuple, set et dictionnaire).

in	appartenance
Not in	Non appartenance

tableau = [1, 2, 3]
a = 2
a in tableau

True

5.8 Les commentaires

Le #
python n'est pas typé de manière statique mais fortement typé de manière dynamique

*5.9 L'opérateur * et **

L'opérateur * de Python permet de décomposer une séquence en arguments individuels pour les fonctions. Ceci est particulièrement utile pour gérer des entrées dynamiques ou de tailles inconnues.

```
def multiply(x, y, z):
   return x * y * z
A_tuple = (2, 3, 4)
A_list = [5, 6, 7]
result = multiply(*A_tuple)
print(result)
result = multiply(*A_list)
print(result)
```

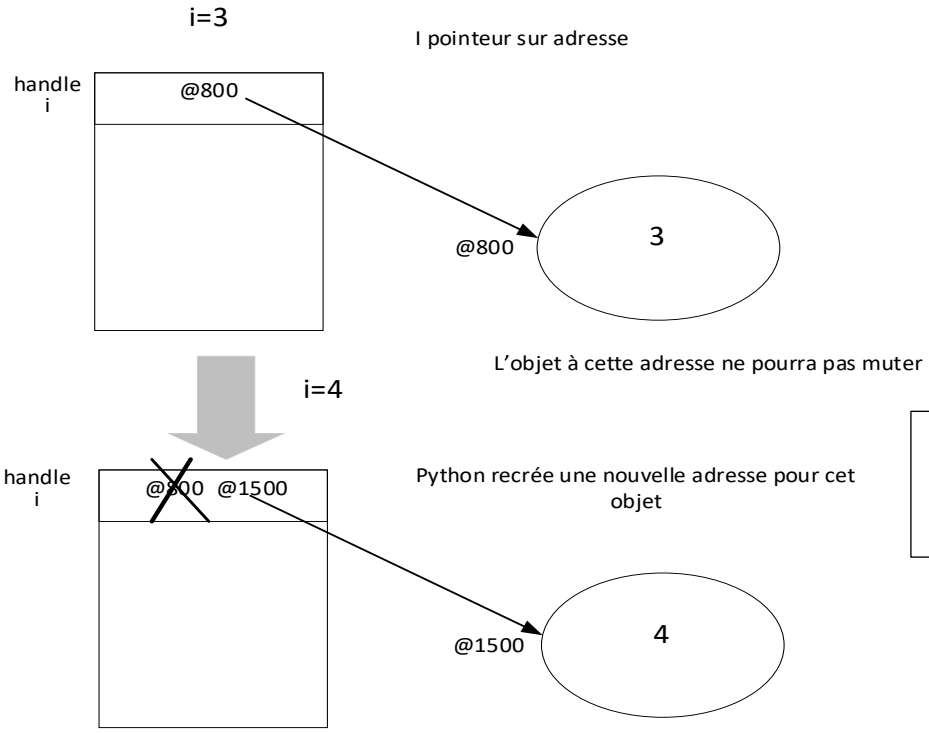
Et l'opérateur ** vous permet même de décompresser et de passer des dictionnaires en tant qu'arguments.

```
def add_numbers(a, b):
   return a+b
dict = {"a":2, "b":3}
result = add_numbers(**dict)
print(result)
```

6 Objet non mutable (objet qui ne peut pas changer d'état)

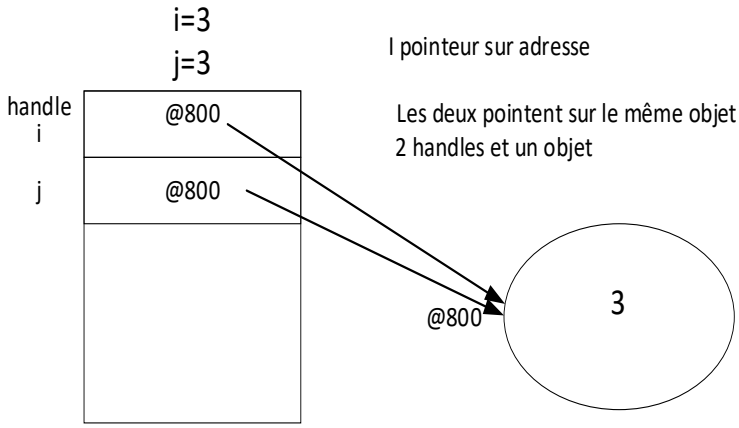
Objet de base : int,str,float,complex,bool,tuple

Exemple de non mutabilité :



En faisant i+=1 il va créer un nouvel objet intermédiaire pour l'opération

Autre curiosité du non mutable sur l'optimisation de la mémoire

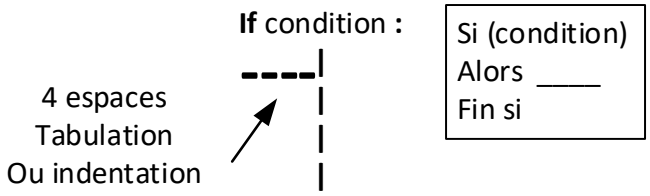


Très peu de méthodes pour les objets non mutables. On utilise essentiellement les opérateurs

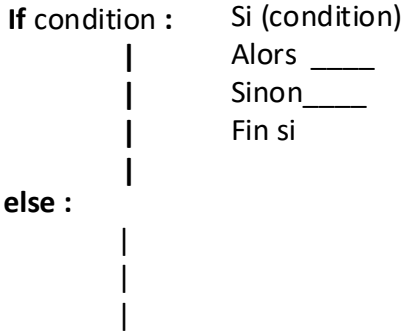
Une fois initialisé on ne peut plus modifier les éléments dans un tuple

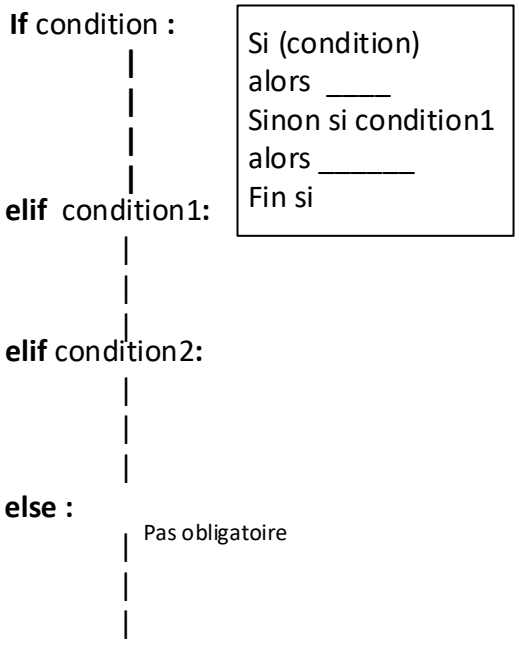
7 Les instructions en python

7.1 ***Structure alternatives ou de test :***



Si (condition)
Alors ____
Fin si





Si (condition)
alors ____
Sinon si condition1
alors _____
Fin si

if 10<b<3 :
écriture impossible pour la condition en java possible en python

```
match operation:
    case '+':
        return a + b
    case '-':
        return a - b
    case '*':
        return a * b
    case '/':
        return a / b
    case _:
        return "Invalid operation"
```

suivant variable **faire**
valeur1 **:** action1 ;
valeur n **:** action n ;
sinon action n+1 ;
fsuivant ;

TD : Créer un script qui saisit la valeur de votre age et qui vous indique jeune si vous avez moins de 40 ans et vieux si vous avez plus de 40 ans

7.2 ***Structures itératives ou boucles***

While condition **:**
____|
|
|
|

Tant que (condition)
Faire -----
Fin faire

TD :

Le script demande à l'user son N° secu 13 chiffres et aussi la clef en deux saisies
validation du numéro de secu
Le N° est valide si cle =97 - N°secu modulo 97
le script boucle tant que le numéro n'est pas bon et tant que l'utilisateur veut continuer en répondant oui à la demande qui lui est faite

La suite de fibonacci et l''affectation simultanée

```
a, b, c = 1, 1, 1
print(b)
while c < 15:
        a, b, c = b, a + b, c + 1
print(b)
```

for element in range(debut,fin) **:**
|
|
|
|

Pour i de debut à fin-1 faire
-----
i=i+1
fin faire

attention fin exclue

for element in collection **:**
|
|
|
|

Attention debut inclus fin exclu **[debut fin [**

On balaye à chaque itération les valeurs des éléments de la collection. Exemple pour une liste

```
>>> for element in ["pomme","poire",3]:
        print(element)

pomme
poire
3
```

La fonction enumerate : pour obtenir l'indice d'un iterable **on prend la fonction enumerate** qui renvoie un tuple (un couple (indice,element))pour chaque itération

```
a = ["Geeks", "for", "Geeks"]

# Iterating list using enumerate to get both index and element
for i, name in enumerate(a):
   print(f"Index {i}: {name}")

# Converting to a list of tuples
print(list(enumerate(a)))
```

Syntaxe de la méthode enumerate()

enumerate(itérable, début=0)

Paramètres d'entrées:

- Itérable : tout objet qui prend en charge l'itération
- Start : la valeur de l'index à partir de laquelle le compteur doit être démarré, par défaut elle est 0

Exemples:

```
noms = ['Bob', 'Alice', 'Guido']
a= enumerate(noms)
print (a)
enumerate object at 0x1057f4120
```

Comme vous pouvez le voir, il s'agit d'un itérable : <enumerate object at 0x1057f4120>. C'est l'itérateur réel. Et comme je l'ai dit, il génère ses éléments de sortie un par un et lorsqu'ils sont demandés (à la volée).

Afin de récupérer ces éléments pour que nous puissions les inspecter, je vais donc utiliser la fonction native list() sur l'itérateur :

```
b= list(enumerate(noms))
print (b)
```

La fonction range est une fonction qui renvoie un élément. Elle dispose de trois arguments

```
def range (début :int, fin :int, pas :int)->object:
```

debut est inclus, fin est exclus

```
for i in range(0,5):
        if i==2:
                continue
        if i==4:
                break
        print(i) # result 0 1 3
```

Continue : permet de passer à l'itération suivante

Break : permet de sortir de la boucle

Le in du for veut dire vient piocher chaque élément de la collection

```
for _ in range(5):
   print("Hello")
```

Dans ce cas, le trait de soulignement est utilisé pour indiquer que les valeurs individuelles produites par le for ne sont pas nécessaires dans la boucle. C'est une façon de dire : « Je vais boucler 5 fois, mais je ne me soucie pas des valeurs réelles du for.

```
B= range(10)

print(B)  #range(0, 10)
```

L'objet renvoyé par range() se comporte presque comme une liste, mais ce n'en est pas une. Cet objet génère les éléments de la séquence au fur et à mesure de l'itération, sans réellement produire la liste en tant que telle, économisant ainsi de l'espace.

On appelle de tels objets des itérables, c'est-à-dire des objets qui conviennent à des fonctions ou constructions qui s'attendent à quelque chose duquel elles peuvent tirer des éléments, successivement, jusqu'à épuisement.

TD : créez une boucle qui permet d'afficher les chiffres de 0 à 42

7.3 L'instruction with as

Par conséquent, nous pouvons également utiliser l'instruction with avec la fonction open() en Python pour ouvrir un fichier, effectuer des opérations dessus et le fermer. La syntaxe d'utilisation de l'instruction with est la suivante.

```
with open(filename, mode) as myFile:

   //read from the file
   // write to the file.
```

Dans la syntaxe ci-dessus, le fichier est ouvert et le pointeur de fichier est affecté à la variable myFile. Ensuite, nous pouvons effectuer des opérations de lecture et d'écriture sur le fichier à l'aide de l'objet myFile. Lorsque toutes les instructions à l'intérieur du bloc with sont exécutées, le fichier est automatiquement fermé.

8 Les fonctions de saisie et d'affichage en console

La saisie **input**

Cette fonction renvoie toujours un string

e=input("veuillez saisir un entier ")

l'affichage deux formes

la forme habituelle

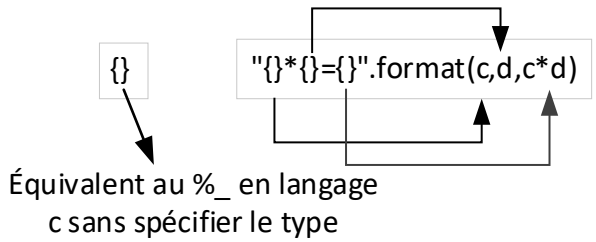
print("la valeur de V1 ",V1)

la forme python

```
print("la valeur de V1 vaut {} et la valeur de V2 vaut {}".format(V1,V2))
print(f"la valeur de V1 vaut {V1} et la valeur de V2 vaut {V2}") # f abrégé de la fonction format
```

Le formatage consiste à écrire une chaine de caractère en y insérant des variables dont nous donnerons la valeur par la suite

```
>>> c=2
>>> d=3
>>> "{}*{}={}".format(c,d,c*d)
'2*3=6'
```



Autre exemple vieux format :

```
myStr = "The color is {} and the temperature is {}°C"
print(myStr.format("red", 30.6)) # The color is red and the temperature is 30.6°C
```

9 Les expressions régulières sur les chaines (patron ou pattern)

Les expressions régulières, également appelées RE, ou regex, ou modèles d'expression régulière, fournissent un moyen puissant pour rechercher et manipuler des chaînes de caractères.
En fait, les expressions régulières utilisent une séquence de caractères et de symboles pour définir un modèle de texte. Un tel modèle est utilisé pour localiser un morceau de texte dans une chaîne de caractères en faisant correspondre le modèle aux caractères.
Pour rechercher un modèle dans une chaîne de caractère, on utilise les fonctions du module re.

```
import re
modele = r"bonjour"
texte = "bonjour les amis"
if re.match(modele, texte):
        print("Oui")
else :
        print("Non")
```

match() : recherche le modèle et retourne la première occurrence. Cette méthode vérifie une correspondance uniquement au début de la chaîne.

search() : recherche la première occurrence du modèle. Contrairement à match(), elle vérifie toutes les lignes de la chaîne.

findall(): recherche toutes les occurrences qui correspondent à un modèle. En revanche, search() ne retournera que la première occurrence.

group() : renvoie un ou plusieurs sous-groupes qui correspondent à notre modèle.
Les caractères spéciaux

La puissance des expressions régulières vient lorsque nous ajoutons **des caractères spéciaux**, ce qui permet de contrôler plus précisément quelles lignes correspond à la chaîne. L'ajout de ces caractères spéciaux à notre expression régulière permet de faire des correspondances et des extractions sophistiquées avec peu de code.

Une expression régulière se compose de trois entités :

Une classe
Un joker
Une cardinalité

La classe :
^ chaine qui commence par
$ chaine qui finit par
[] un intervalle
(|) un regroupement

Exemple :
^ [GG ou TT] commence par GG ou TT
^ [0-9] commence par n'importe quel chiffre
^ [A B C] commence par A ou B ou C
^ [a-z] commence par une minuscule
^ ATG commence par ATG
^ T[A B C]G commence par TAG ou TBG ou TCG
^ T[ABC]G commence par TABCG ou TABCG ou TABCG

Le joker :
\ : échappement
. : n'importe quel caractère
\d n'importe quel chiffre (d signifie digit), équivalent à [0-9]
\w n'importe quel caractère alphanumérique et le caractère souligné
\s les espaces

La cardinalité (nombre de fois)

Elle s'applique à la classe qui précède
+ : 1à n
?: O ou 1
* :0 à n fois le caractère précédent ou l'expression entre parenthèses précédente
{min,max} : de min à max d'occurrence

{n} : n fois le caractère précédent ou l'expression entre parenthèses précédente

{n,m} n à m fois le caractère précédent ou l'expression entre parenthèses précédente

{n,} Au moins n fois le caractère précédent ou l'expression entre parenthèses précédente

{,m} Au plus m fois le caractère précédent ou l'expression entre parenthèses précédente

Exemple :
^ [a e i o u y]+ Commence par une voyelle répétée de 1 à n fois (aab ou ab mais pas ba)
^ [a e i o u y]{2} Commence par une voyelle répétée 2 fois

Exemple d'utilisation Le téléphone

```
pattern = "^0[67]([-. / ]?[0-9]{2}){4}$" # Numero  qui commence par un 0 puis un 6 ou un 7 puis une série de 4 x 2 chiffres séparé par des -./ ou

telephone=input("veuillez saisir votre numéro")
if (re.search(pattern,telephone)is not None ):
        valide = True
        print("numero valide")
else :
        valide = False
```

```
import re
animals = "cat swift dog"
match = re.search("swift", animals)
print(match)
#<re.Match object; span=(4, 9),match='swift'>
if match:
    print("Found")
# Found
```

```
import re
strings = ["cat swift dog swift",
"swift cat swift dog"]
regex = re.compile("^swift")
for s in strings:
    if regex.search(s):
    print(f"{s}: True")
```

```
    else:
    print(f"{s}: False")
```

```
import re
s = "Le nombre pi est 3.14 et j'aime beaucoup le chiffre 42"
regex = re.compile("[0-9]+\.?[0-9]*")
results = regex.findall(s)# renvoie une liste des éléments cherchés
print(results)
# ['3.14', '42']
```

TD:

le script demande à l'user son numéro de téléphone
le script vérifie le numéro on boucle si il n'est pas bon pas et si l'utilisateur le veut bien en répondant oui à la question voulez vous continuer
un numéro est valide si il commence par 06 ou 07
et doit contenir les 8 chiffres supplémentaires organisés comme suit:
2 chiffres séparés ou non par soit - espace ou /
puis encore 2 chiffres ...

Le type hint

C'est une façon de devenir plus explicite sur les types lors de l'écriture du code
Il n'est pas obligatoire et ne change pas le comportement du Python à runtime

```
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
x: list[int] = [1]
```

10 Les collections ou les structures de données

Les list les tuples ou les String sont des structures de données qui forment des séquences. Une séquence est un ensemble d'éléments ordonnés un ensemble dans lequel il y a un ordre et une indexation. On peut donc atteindre chaque élément par [indice]
C'est donc un itérable un objet qui contient une séquence d'éléments sur lesquels on peut itérer.
On peut boucler sur tous les itérables grâce à une boucle for.
Parmi les objets natifs itérables, on retrouve les listes, les dictionnaires, les tuples et même les chaînes de caractères (qui ne sont en fait que des séquences de caractères individuels)

10.1 Les tuples

Objets non mutables très rapide d'accès et très petit (très utiles pour renvoyer plusieurs paramètres dans une fonction)
Déclaration initialisation (les parentheses)

monTuple= (3,5, "coucou ",3,3.8) # parenthèse pour tuple

```
t=[1,2,3]
tuple1=tuple(t)          # on ne peut pas donner comme nom tuple à un tuple
tuple1[0]                # me donne la valeur de 1
>>> tuple1
(1, 2, 3)
```

Sans rien préciser ni () ni [] ni {} on a un tuple

```
montuple=1,2,3
type(montuple)
<class 'tuple'>
```

l'Accès

On peut lire directement un tuple

```
a=monTuple[1]
```

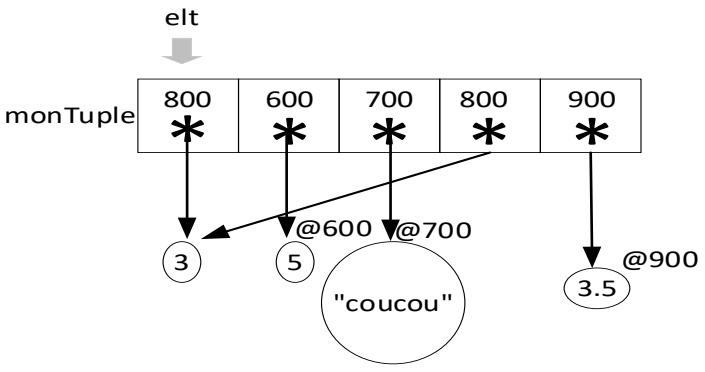
on ne peut pas écrire directement dans un élément du tuple

~~monTuple[1]=3~~

s'il contient un élément de type mutable, comme une liste, vous pouvez modifier cet élément particulier.

```
mon_tuple = ("a", 3, [5,9])
mon_tuple[2][1] = "c"
mon_tuple
('a', 3, [5, 'c'])
```

Le parcours d'un tuple




```
for elt in monTuple :
        print(elt) # renvoie chaque élément du tableau à chaque itération
```

pour obtenir l'indice du tuple en plus **on prend la fonction enumerate** qui renvoie un tuple (un couple (indice,element)) pour chaque itération

```
for i,elt in enumerate(monTuple) : # on appele cela l'éclatement
        print(i) # renvoie chaque élément du tuple à chaque itération
        print(elt) # renvoie l'indice de chaque élément du tuple à chaque itération
```

si l'on fait :

```
for elt in monTuple :
        print(elt)
        elt*=3 # surcharge des opérateurs suivant le type
```

on ne modifie rien dans monTuple car à chaque itération python crée un objet intermédiaire qui sera détruit par la ramasse miette. Une fois initialisé on ne peut plus modifier les éléments dans un tuple
Pour tester si un tuple contient un élément spécifique, vous pouvez utiliser les opérateurs d'appartenance in et not in.

```
mon_tuple = ("a", 3, [1,2])
1 in mon_tuple
False
```

Pour supprimer un tuple

```
del mon_tuple
```

L'agrandissement d'un tuple

monTuple += (4,6) # ajout de deux éléments du type entier à la fin de mon tuple

La copie d'un tuple

```
monTuple= (3,5, "coucou ",3,3.8)
t2=monTuple # 2 handlers qui pointent sur le même objet
t2 +=(1,2) # 2 handlers (monTuple,t2) qui pointent sur deux objets différents cette fois ci MonTuple 5 éléments et t2 7 éléments
```

L'indexing

```
A= monTuple[0]
```

Le découplage de tuple (le slicing)

t=monTuple[debut :fin :pas] on garde les éléments à partir de l'indice debut jusqu'à l'indice fin-1 par pas de pas c'est [debut fin[

```
monTuple= (3,5, "coucou ",3,3.8)
t= monTuple[0 :4 :2]  # t contient les éléments 3 "coucou "
```

le tuple permet de faire une copie parfaite dans un autre handle ou objet

Les méthodes de tuple

count() : renvoie le nombre de fois où l'élément apparaît dans le tuple.

```
mon_tuple = ("bonjour", 1, 2, 1, 2)
mon_tuple.count(1)              #renvoie la valeur 2
```

index() : renvoie l'index du premier élément qui est égal à l'élément donné.

```
mon_tuple = ("a", 1, 5, 8,"a")
```

```
mon_tuple.index("a")
0
```

Quel peut être I ‘intérêt d'utiliser ce type plutôt qu'une liste ? Tout d'abord, les données ne peuvent pas être modifiées par erreur mais surtout, leur implémentation en machine fait qu'elles occupent moins d'espace mémoire et que leur traitement par L’interpréteur est plus rapide que pour des valeurs identiques mais stockées sous forme de listes.
De plus de par leur aspect non modifiable, les valeurs contenues dans un tuple peuvent être utilisées en tant que clé pour accéder à une valeur dans un dictionnaire (alors que c'est beaucoup plus dangereux avec les valeurs contenues dans une liste)

10.2 Les Listes

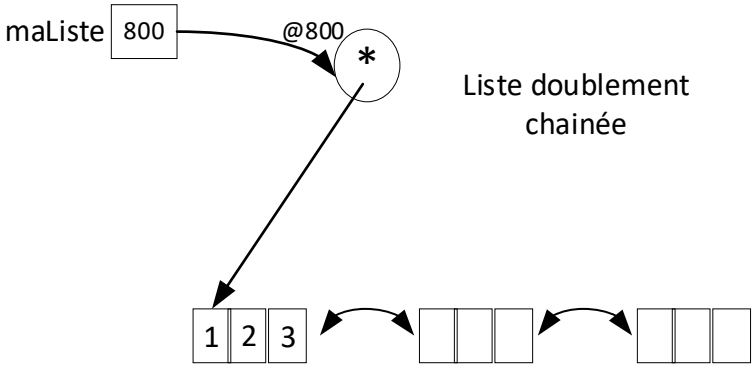
Objet mutable va faire appel à des méthodes ou des comportements

Déclaration (les crochets)

maListe = [1,2,3]
maListe1=[]
maListe2=list() #appel du constructeur

Le remplissage

maListe = [[1,2,3], [1,2,3],] [1,2,3]] # liste de sous liste une collection dans une autre collection



Pour ajouter un élément à la fin de ma liste append

maListe.append(34)

Pour ajouter une collection à la fin de ma liste extend

maListe.extend(("a","b","c")) on rajoute une collection de tuple

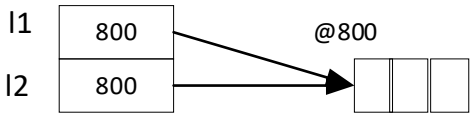
Pour insérer un élément à un indice de la liste insert

maListe.insert(3,4444444)*# insertion en quatrième position d'une valeur*

La longueur d'une liste fonction len

Longueur=len(maliste)

Une liste est un objet mutable
l2 = l1



Si je modifie l1 je modifie l2 (mutable = même objet)

Pour supprimer un élément de la liste remove ou del

l=[1,2,3,4,5]
l.**remove**(3) # supprime l'élément à l'indice 3 de la liste me donne [1, 2, 4, 5]
del l[3] # idem que précédemment

c = [1, 2, 3]
d = c * 2
d # Retour: [1, 2, 3, 1, 2, 3]

Pour savoir si un élément appartient à une liste le in

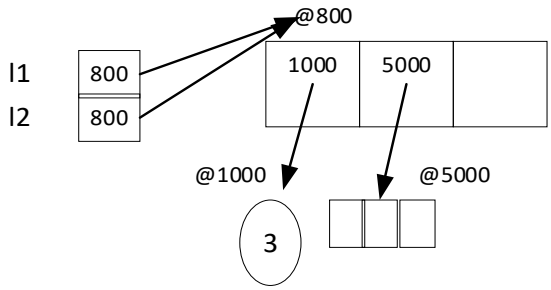
distrib=["debian","ubuntu","redhat"]
"debian" in distrib #retoune la valeur true

Pour obtenir d'indice de la première occurrence d'un élément d'une liste index

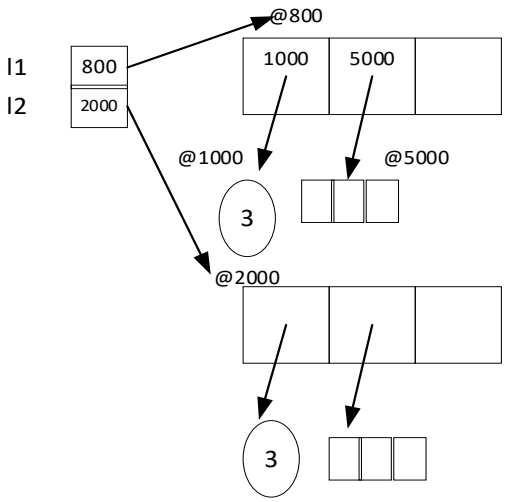
distrib.index("redhat") #retoune la valeur 2

La copie ou le clonage

l1=[3,[1,2,3]]
l2=l1 # pas de copie ou de clonage ici c'est le même objet



l2=l1.copy()



copy() : copie superficielle du premier niveau
Si au premier niveau on a des non mutables plus possibles

deepcopy() : copie profonde avec copie des valeurs non mutables
la liste est utile quand son contenu est homogène on perd la sémantique dans ce cas (qui fait qui et qui fait quoi)

indexing

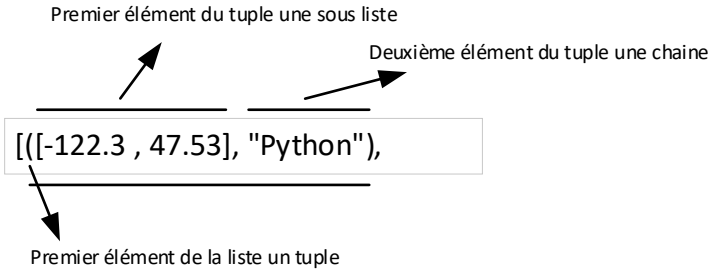
ville=['paris','berlin','londres','bruxelles']
ville [-1] donne le dernier élément
ville [-2] donne l'avant dernier élément

Le slicing

ville =['paris','berlin','londres','bruxelles']
ville [:3] # donne paris berlin londres debut=0
ville [2 :] # donne londres bruxelles fin = dernier
ville [: :2] # donne paris londres debut =premier fin=dernier
ville [: :-1] # donne bruxelles londres berlin paris debut=dernier fin=premier
le in peut aussi servir à trouver un élément sans utiliser la boucle itérative

```
if ‘paris ‘ in ville :
        print(‘oui’)
else
        print(‘non’)
```

Liste de tuple chaque tuple est constitué d’une sous liste et d’une chaine de caractère




```
cities = [([-122.3 , 47.53], "Python"),      # Seattle
([-96.85, 32.85], "Java"),                   # Austin
([-89.33, 43.13], "R")                       # Madison
]
```

```
print(f" on affiche le premier" ,cities[0]) # premier élément de la liste qui est un tuple
```

on peut directement dissocier les tuples avec un for :

```
for (longitude, latitude), language in cities:
   print(longitude)
   print(latitude)
   print(language)
```

on récupére chaque élément de la liste

```
for elt in cities:     #elt est un tuple je balaye ma liste et renvoie chaque élément
      print(elt)     #affichage du tuple
      print(elt[0])   #affichage de la liste
      print(elt[0][0]) #affichage du premier éléments de la liste
      print(elt[1])   #affichage de la chaine
      for i, j in enumerate(elt):
         print(f"mon tuple[{i}]={j}")#
```

Une chaine de caractère est une liste particulière ne contenant que des caractères

```
>>> chaine="toto"
>>> chaine[0]              # lecture du premier caractère 't'
```

Les sous listes

```
l=[1,2,['a','b',['000','001','010']]]#troisième élément = sous-liste cette sous-liste contient aussi une sous liste
len(l)              # me donne trois les sous listes ne sont pas comptabilisés
>>> l[2]
['a', 'b', ['000', '001', '010']]
>>> l[2][2][0]
'000'
```

La fonction de combinaison zip

associe des éléments de plusieurs listes

```
names = ["Alice", "Bob"]
scores = [95, 85]
combined = list(zip(names, scores))
print(combined)  # Output: [('Alice', 95), ('Bob', 85)]
```

La fonction map

applique une fonction à chaque élément d'une liste

```
nums = [1, 2, 3]
squared = list(map(lambda x: x**2, nums))
print(squared)  # Output: [1, 4, 9]
```

la fonction filter

ne conserve que les éléments qui satisfont à une condition.

```
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)  # Output: [2, 4]
```

la fonction de tri

sorted().sort()

- **sorted()**: Crée une nouvelle liste triée sans modifier l'original.
- **.sort()**: Trie la liste en place (modifie la liste d'origine).

```
nums = [3, 1, 4]
print(sorted(nums))  # Output: [1, 3, 4]
print(nums.reverse() )# Output: [4, 3, 1]
print(nums)        # Output: [3, 1, 4]
nums.sort()
print(nums)        # Output: [1, 3, 4]
```

Tri d'une liste de dictionnaires

```
people = [ {"name": "Kuldeep", "age": 25}, {"name": "Mike", "age": 30}, {"name": "Peter", "age": 20}]
sorted_people = sorted(people, key=lambda x: x["age"])
print(sorted_people) # [{'name': 'Peter', 'age': 20}, {'name': 'Kuldeep', 'age': 25}, {'name': 'Mike', 'age': 30}]
```

Tri d'une liste de tuple

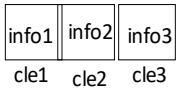
```
people = [("Kuldeep", 25), ("Mike", 30), ("Peter", 20)]
sorted_people = sorted(people, key=lambda x: x[1])
print(sorted_people)# [('Peter', 20), ('Kuldeep', 25), ('Mike', 30)]
```

`liste.append(x)`	ajoute l'élément `x` à la fin de la liste `liste`
`liste.index(x)`	renvoie l'index de l'élément `x` dans la liste `liste` *Si l'élément n'est pas dans la liste, renvoie un message d'erreur. Si l'élément apparaît plusieurs fois, renvoie le plus petit index.*
`liste.count(x)`	renvoie le nombre d'occurrences de l'élément `x` dans la liste `liste`
`liste.remove(x)`	retire l'élément `x` de la liste `liste` *S'il y a plusieurs occurrences de `x` dans la liste, seule la première est enlevée. Renvoie un message d'erreur si `x` n'est pas dans la liste.*
`liste.insert(i,x)`	insère l'élément `x` à l'index `i` *Si cet index dépasse la taille de la liste, l'élément est ajouté à la fin.*
`liste.sort()`	ordonne la liste `liste`

10.3 Les dictionnaires

Objet mutable

Les éléments sont non ordonnés dans le dictionnaire, les valeurs n'ont pas d'index mais des clefs. La clef est toujours une chaine. C'est un élément qui reste encore itérable même si il n'y a pas d'index apparent



Chaque clef est unique. On ne peut pas avoir deux fois la même clef. Un dictionnaire contient des association clef valeurs. Etant que les éléments sont non ordonnés On peut donc atteindre les éléments par [valeur de la clef]

Déclaration les accolades

monDico={clef1 :valeur1,clef2 :valeur2,clef3 :valeur3}

La clef est séparée de la valeur par :

monDico1={"C1" :3, "C2" :2, "C3" :3}
monDico=dict()

L'accès

a=monDico["C1"] # on récupère la valeur c'est-à-dire 3
monDico["C1"]=2 # on modifie la valeur de la clef "C1"

Le balayage d'un dictionnaire values(),keys(),items()

```
for valeur in monDico.values(): # on récupère les valeurs
        print(valeur)

for clef in monDico.keys(): # on récupère les clefs
        print(clef)

for cle,valeur in monDico.items(): # on récupère les clefs et les valeurs
        print(cle,valeur)
for i, element in enumerate(d):
   print(element) # la clef
   print(d[element])# la valeur associé à la clef
   print(i)# indice de la clef
```

Enumération d'un dictionnaire

```
d = {"clef": "valeur", "clef2": "valeur2"}
for i, element in enumerate(d):
   print(element)
   print(d[element])
   print(i)
```

copie d'un dictionnaire

d1=d2
d1=d2.copy()
d1=d2.deepcopy()

Pour créer une nouvelle association :

monDico["C2"]=3 # ont créé une nouvelle association

Des méthodes renvoient des objets itérables : ce ne sont pas des listes et ils ne consomment donc pas autant de place en mémoire que pour stocker une liste, on ne stocke qu'un pointeur vers l'élément courant. Avec ces structures vous pourrez toujours utiliser les boucles for classiques. par contre vous n'aurez plus un accès direct aux valeurs en spécifiant leur indice.
Dans un dictionnaire pour obtenir la valeur correspondant à une clé on peut bien entendu utiliser la notation classique dictionnaire[cle] mais on sera aussi utiliser la methode get() qui renvoie la valeur associée à la clé passée en premier paramètre ou, si elle n'existe pas, la valeur passée en second paramètre

Recherche d'une clef

A= monDico.get("cle ",None)

Si clef n'existe pas on renvoie un objet par défaut None par exemple ou ce qu'on l'on veut

Dictionnaire qui a comme valeur un autre dictionnaire

```
config = {"database": {"host": "localhost", "port": 5432}}
# Safe access using get()
db_host = config.get("database", {}).get("host", "default_host")
print(db_host) # Output: localhost
```

La méthode counter ()

Elle permet de trier le nombre d'occurrence d'éléments d'une liste en fournissant le résultat sous la forme d'un dictionnaire :

```
from collections import Counter
```

```
words = ["apple", "banana", "apple"]
word_count = Counter(words)
print(word_count)#  Counter({'apple': 2, 'banana': 1})
print(dict(word_count))# {'apple': 2, 'banana': 1}
```

le dictionnaire par default defaultdict

Le defaultdict du module collections simplifie la gestion des clés manquantes dans les dictionnaires en fournissant automatiquement des valeurs par défaut.

```
from collections import defaultdict
fruit_count = defaultdict(int)
fruits = ["apple", "banana", "apple", "cherry"]
for fruit in fruits:
   fruit_count[fruit] += 1
print(fruit_count)     # defaultdict(<class 'int'>, {'apple': 2, 'banana': 1, 'cherry': 1})
print(dict(fruit_count))   #{'apple': 2, 'banana': 1, 'cherry': 1}
```

```
from collections import defaultdict
text = "hello world hello"
word_count = defaultdict(int)
for word in text.split():
   word_count[word] += 1
print(dict(word_count))
```

Création d'un dictionnaire à partir d'une liste :

Liste1=['paris','londres','berlin']
monDico.fromkeys(Liste1) # les valeurs auront par défaut None
monDico.fromkeys(Liste1,1000) # les valeurs auront par défaut 1000

Pour supprimer nous connaissons déjà la méthode del

monDico1={"C1" :3, "C2" :2, "C3" :3}
del (monDico1["C2"]]

Suppression d'une association :

nbreHabitant=monDico.fromkeys('paris') # efface l'association dans le dictionnaire et renvoie la valeur

Vous pouvez vérifier l'existence d'une clé à l'aide de in :

grades = { "Joel" : 80, "Tim" : 95 }
joel_has_grade = "Joel" in grades # Vrai
kate_has_grade = "Kate" in grades # Faux

Les conversions

De tuple à liste

L=list(t) # avec t un tuple et L une liste

De liste à tuple

T= tuple(l) # avec T un tuple et l une liste

De dico en tuple

T=tuple(choice(monDico)) choice est une méthode qui retourne un dico de un élément au hasard (clef,valeur) d'un dictionnaire

De float en str

pi = 3.1415
print(type(pi)) # float
piInString = **str(pi)** # float -> str
print(type(piInString)) # str

De String to float

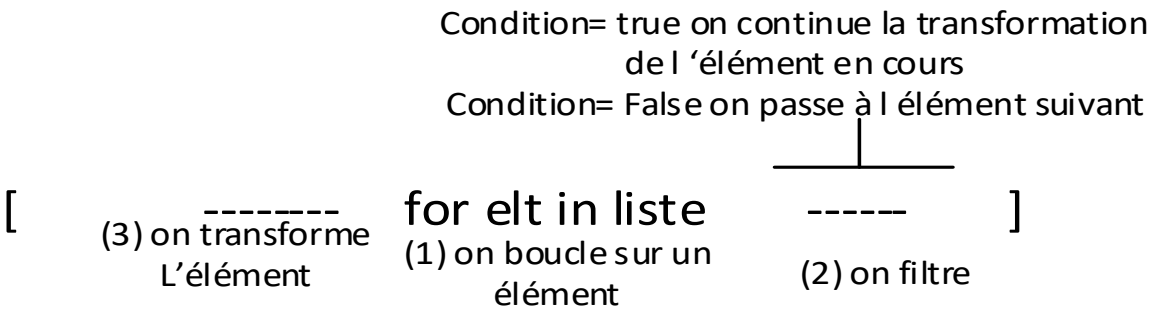
num = "3.1415"
print(num)
print(type(num)) # str
pi = **float(num) # convert str to float**
print(pi)
print(type(pi)) # float

De String to int

num = "1234"
print(num)
print(type(num)) # str
num1 = **int(num) # convert str to float**
print(num1)
print(type(num1)) # float

11 Les listes, les dictionnaires les tuples en compréhension

On insère la boucle for à l'intérieur de la liste elle-même. On simplifie l'écriture et le processus de création se fait plus rapidement



Attention à l'ordre

Séquentiellement: (1) on boucle (2) on filtre – (3)on transforme

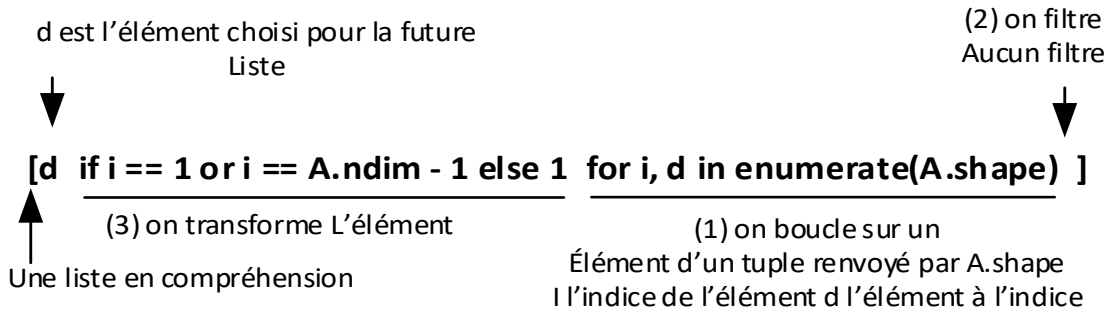
Les listes en compréhension

```
listePair = [elt for elt in range(50,10000) if not(elt%2) ]    # pas de tranformation  filtrage (if elt%2 ==0) retourne une liste

ret
listeTuples1 = [f"{indice}:{valeur}" for indice,valeur in enumerate(maListe1) if 100< valeur <200 ]# retourne une liste de chaine dont les valeurs sont comprises entre 100 et 200 exclues
```

Une liste dans une liste

```
Liste3= [[i for i in range(3]] for j in range(3)]
[[0,1,2], [0,1,2], [0,1,2]]
A = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])# tableau de dimension 3
print (A)
print(A.shape)# B.shape renvoie un tuple nbre de ligne pour chaque dimension (2,2,2)
print(A.ndim)# retourne la dimension d'un tableau 3 ici
shape = [d  if i == 1 or i == A.ndim - 1 else 1  for i, d in enumerate(A.shape)  ]
```



Shape est une liste # [1, 2, 2]

Les dictionnaires en compréhension

```
Dictionnaire = {"0" : "banane",
"1" : "cerise",
"2" : "pomme",
"3" : "kiwi",
}
Peremption = [ 2,3,2,1]
Liste =["banane","cerise","pomme","kiwi"]

Dico ={ k:v for k,v in enumerates(Liste)}
Dico = {fruit:age for fruit,age in zip(Liste,preemption)}

sentence = "Awareness is power in a world where knowledge is everywhere"
words = sentence.split()
word_embeddings = {word: [ord(char) - 96 for char in word.lower()] for word in words}
```

```
[ord(char) – 96        for char in word.lower()        ]
```

(3) on traite on prend l'entier du carac ASCII -96

(1) on boucle On prend tous les caractères d'un mot

(2) aucun filtrage

{'Awareness': [1, 23, 1, 18, 5, 14, 5, 19, 19], 'is': [9, 19], 'power': [16, 15, 23, 5, 18],…….

Les tuples en compréhension

```
T1= tuple((i**2 for I in range(5))          # creation d'un tuple en comprehension
```

Td :

Créez une liste maliste de chiffre qui vont de 50 à 9999 en utilisant les listes en compréhension

En utilisant les liste en compréhension créez une nouvelle liste listepair qui ne contient que les chiffres pairs de 50 à 9999

```
 even_numbers = [x for x in range(5) if x % 2 == 0] # [0, 2, 4]
squares = [x * x for x in range(5)] # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers] # [0, 4, 16]
```

12 Les fonctions

Ce sont des sous programmes

12.1 La fonction numérique simplifiée avec l'opérateur lambda

La syntaxe de base de la fonction lambda :

```
Lambda arguments: expression
```

f= lambda x,y : x**2+y

Equivaut à faire :

$$f(x,y) = x^2 + y$$

print(f(3,4)) ;

Pour filtrer les données

Par exemple, si nous avons une liste de nombres et que nous voulons n'avoir que des nombres pairs. Ensuite, au lieu d'écrire une autre fonction complète, nous pouvons simplement utiliser la fonction lambda

```
num = [1, 2, 3, 4, 5, 6]# We want to keep only even numbers
evens = list(filter(lambda x: x % 2 == 0, num))
print(evens)
```

Output: [2, 4, 6]

12.2 La fonction universelle

Déclaration

```
def  mafonction() :
        |
        |
        |
        |
```

Une fonction complète par exemple

```
def f1(param1:int, param2:int=10) -> tuple:
        -----
        ------
        return (2,3)
```

Les arguments sont param1, param2. Ceux sont des prêtes nom (keyword) qui nous permettent de faire passer des valeurs ou des adresses à notre fonction

le nom de la fonction est f1 les types sont donnée à titre indicatif

=10 et le paramètre par défaut avec une valeur par défaut de 10 si ce paramètre n'est pas appelé

les variables par défaut sont toujours à droite

def f1(p1, p2=10,p3=11)

on ne peut pas mélanger les variables par défaut en plein milieu des autres
~~def f1(p1=0, p2,p3=11)~~

Il n' y a pas de surcharge de fonction en python les variables par défaut peuvent s'y apparenter

Une variable globale à une visibilité dans tous le code sauf dans les fonctions

Une Variable locale à une visibilité dans les fonctions

```
def f2(uneListe:list)->None:          # passage par référence
```

Exemple :

def fct(p1=0 ,p2=0,p3=0)

Utilisation appel

Fct() ok avec 0 comme valeur pour tous les paramètres
Fct(1) p1=1 p2=0 p3=0
Fct(1,5) p1=1 p2=5
Fct(p3=8) p1=0 p2=0 p3=8 on peut nommer et initialiser les paramètres
B,C=f1(3) on récupère les deux variables du tuple dans B et C

Le mot clé **pass** permet de continuer par la suite la fonction ou la classe

```
def maFonction()->None
    pass
```

En python l'ordre des paramètres n'est pas fixé du moment où ceux-ci sont nommés .

```
def premier(a,b) :
    return a
```

l'appel

c= premier**(b=3,a=2)**

*12.3 Nombre de paramètre non fixés *args et **kwargs*

args pour un tuple et kwargs pour un dictionnaire

```
def exemple(*args):
    print(args)
exemple(1,2,3)          # renvoie le tuple (1,2,3)
```

```
def sum(*argv):
    total = 0
    for value in argv:
    total = total + value
    print("The sum is equal to", total)
```

```
def exemple1(**kwargs):
        print(kwargs)

exemple1(i=2,z=3,w=4) # renvoie le dictionnaire {'i': 2, 'z': 3, 'w': 4}

def doSomething(**kwargs):

        for key, value in kwargs.items():

        print("The value for {0} is {1}".format(key, value))

doSomething(username = 'Nicolas', age = 35, email ='nicolas@craftedbits.io')
```

en python tous les paramètres sont passés par adresse

Par contre comme certains types ne sont pas modifiables (entier, flottant, complexe) les variables associées à ces types ne pourront être modifiées quand on les fait passer comme paramètre de la fonction

```
>>> def incremente(a):
        a=a+1
>>> b=3
>>> incremente(b)       #b reste à 3 non modifiable
>>> b
3
>>> def modifliste(liste,val):
        liste.append(val)
>>> l=[1,2,3]
>>> modifliste(l,4)       # liste est un objet modifiable
>>> l
[1, 2, 3, 4]
```

Les variables définies à l'extérieur d'une fonction sont **globales**

Les variables globales sont visibles partout mais ne sont pas modifiables

```
def namespace():
        a = 21
        print(a, b)
a, b = 42, 99
namespace()    # 21 et 99
```

Il est néanmoins possible de modifier dans une méthode une variable globale. Pour cela nous devons l'indiquer en utilisant le mot clef global

```
def stepup():
        global a
a = a + 1
        print(a)
a = 42
stepup()        # donne 43 pour a
```

Python a une fonction **filter** prédéfinie qui prend deux arguments, une fonction (ici une fonction lambda appelée element), et une liste numbers et retourne une liste filtered_numbers. La fonction passée comme premier argument à filter doit elle-même prendre un argument numbers et la liste que filter retournera contiendra tous les éléments de la liste passée en argument pour lesquels la fonction passée à filter retourne vrai.

```
numbers = [1,2,3,4,5,6]
filtered_numbers = filter(lambda element: element > 3, numbers)
for n in filtered_numbers:
        print(n, end=',')
```

```
def odd(n):
    return n % 2

li = [1, 2, 3, 5, 9, 10, 256, -3]
filter(odd, li)
```

13 Les décorateurs

13.1 Le principe

Un décorateur est une fonction qui prend une autre fonction comme argument, étend son comportement et renvoie une nouvelle fonction. Cela permet d'ajouter des fonctionnalités supplémentaires de manière dynamique sans modifier la fonction d'origine elle-même.

Un exemple de base sur la façon dont les décorateurs sont créés en Python :

```
def say_hello():
        print("Hello!")
say_hello()

def simple_decorator(func):
        def wrapper():
                print("Starting...")
                func()
                print("Done!")
        return wrapper



# Without @
hello = simple_decorator(say_hello)
hello()
# With @
@simple_decorator
def new_hello():
        print("Hi!")
new_hello()
```

Donc, c'est juste une façon plus courte de dire . C'est la façon dont vous appliquez un décorateur à une fonction.
@simple_decorator
hello = simple_decorator (new_hello)

13.2 Le décorateur @classmethod

Un autre concept avancé dans la POO Python est l'utilisation des décorateurs classmethod et staticmethod. Ces décorateurs modifient les méthodes d'une classe pour changer la façon dont elles peuvent être appelées et les données auxquelles elles peuvent accéder.
Le décorateur est utilisé pour créer une méthode qui est liée à la classe et non à l'instance de la classe. Cela signifie qu'il peut accéder aux attributs de classe, mais pas aux attributs d'instance. Le premier paramètre d'une méthode de classe est généralement nommé **@classmethod,** ce qui fait référence à la classe elle-même.

```
class Person:
  def __init__(self, name):
    self.name = name

  @classmethod
```

```
  def from_full_name(cls, full_name):
    name = full_name.split()[0]
    return cls(name)


# Usage
john = Person.from_full_name("John Doe")
print(john.name)  # Output: John
```

À utiliser lorsque vous avez besoin d'accéder aux attributs de classe ou lorsque vous avez besoin d'instancier la classe dans la méthode(comme les méthodes de fabrique.

13.3 Le décorateur @staticmethod

Le décorateur est utilisé pour créer une méthode qui n'accède pas aux données d'instance ou de classe. Les méthodes statiques ne reçoivent pasde premier argument implicite (ni ni).@staticmethodselfcls

```
class TemperatureConverter:
  @staticmethod
  def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32


# Usage
fahrenheit = TemperatureConverter.celsius_to_fahrenheit(0)
print(fahrenheit)  # Output: 32.0
```

À utiliser lorsque la méthode n'interagit pas avec la classe ou ses instances, et qu'elle sert de fonction utilitaire

13.4 Le décorateur @property, @getter, @setter

Le décorateur .@property transforme une méthode en « getter » pour un attribut en lecture seule portant le même nom. Pour ajouter les fonctionnalités« setter » et « deleter » correspondantes, vous pouvez décorer des méthodes supplémentaires avec @<attribute>.setter@<attribute>.deleter

```
class Temperature:
  def __init__(self, celsius):
    self._celsius = celsius

  @property
  def celsius(self):
    return self._celsius


# Usage
```

```
temp = Temperature(100)
print(temp.celsius)  # 100
```

Une méthode setter est utilisée pour définir la valeur d'un attribut. Il fournit un moyen contrôlé de définir les attributs, permettant la validationou le calcul.

```
class Temperature:
    # ... (previous code)

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value


# Usage
temp = Temperature(0)
temp.celsius = -300  # Raises ValueError
```

Une méthode deleter est utilisée pour supprimer un attribut. C'est utile lorsque la suppression d'un attribut nécessite plus que sa simple suppression de l'objet .

```
class Temperature:
    # ... (previous code)

    @celsius.deleter
    def celsius(self):
        print("Deleting celsius")
        del self._celsius


# Usage
temp = Temperature(50)
del temp.celsius  # "Deleting celsius"
```

14 Les modules ou Les bibliothèques

Certaines fonctionnalités de Python ne sont pas chargées par défaut. C'est le cas de toutes les fonctionnalités incluses à la fois dans le langage et comme fonctionnalités tierces à télécharger à part. Pour utiliser ces dernières, vous devez importer les modules correspondants.
En python les fonctions peuvent être organisées en bibliothèques appelées module (ensemble de fonctions). Pour avoir accès aux fonctions dans le module il faut utiliser la commande **import**

from permet de donner le nom du module

import permet de déclarer les fonctions que l'on veut importer

from math import * # on demande d'importer toutes les fonction du module math

Si on manipule des modules ayant des noms très longs on peut définir un alias avec le mot clé

as **Import** <<fichier>> **as e**

fichier.f1()
e.f1()

from <<fichier>> **import** f1,f2

f1()
f2() l' étoile * (toutes les fonctions) est à éviter après l'import on peut définir un alias

Le chemin de recherche des modules

Par défaut il commence à rechercher par le répertoire courant puis dans le ou les répertoires spécifiés par la variable d'environnement PYTHONPATH et dans le répertoire usr/lib/pythonversion

15 Le programme et la fonction principale

En Python, chaque module a une variable native spéciale appelée __name__ . Lorsque vous exécutez un script Python directement (c'est-à-dire pas en tant que module importé), Python définit __main__. Cela constitue le point d'entrée de votre programme. Python exécute le code sous le bloc if __name__ == "__main__". Mais si vous importez dans un autre module, le code à l'intérieur du bloc __main__ ne s'exécutera pas

Cette ligne vérifie si votre script est exécuté directement ou importé en tant que module. S'il est exécuté directement, __name__ est défini à "__main__", ce qui vous permet de contrôler le code qui s'exécute.

Pour lancer un programme principale dans un module on doit déclarer la fonction main

```
def main() :
        ghghghgh
```

puis rajoutez dans le module lanceur :

```
if __name__ == '__main__':
   main()
   exit(0)
```

pour tester tous les tests de la docstring dans notre module

```
if __name__ == '__main__':
   import doctest as d
   d.testmod(verbose=True)
```

On n' est pas oblige de créer la fonction main(), on peut directement à la fin du module donnez la suite d'instruction suivante :

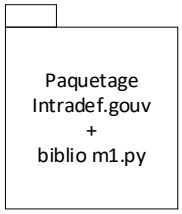
```
if __name__ == '__main__':
    print(sys.argv[1])    # 0:nom du fichier 1 premier argument
    #print(lireTexte("exemple.txt"))
    print(lireCsv(sys.argv[1]))
   # [elt for elt in lireCsv(sys.argv[1])]
    tracerCourbe([i for i in range(0,4)],[4,7,8,9],"les x","les y")
```

16 Les paquetages

Pour regrouper des classes vous pouvez créer des paquetages en plaçant vos fichiers sources dans des répertoires contenant un fichier spécial **__init__.py**

En termes simples :
Marque un paquet :
L'ajout d'un fichier **__init__.py** dans un dossier indique à Python que le dossier est un paquet.

Paquetage ensemble de bibliothèque ou un paquet de différents modules
Nom du paquetage en minuscule pas de caractère spéciaux. Le paquetage est basé sur les espaces de noms

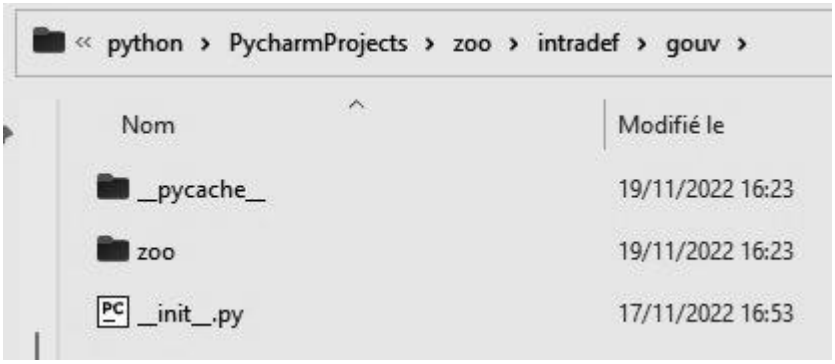


Import intradef.gouv.m1

Le nom du paquetage se lit à l'envers

Pour zoo.intradef.gouv

Lire gouv.intradef.zoo



Le fichier de dossier privé __init__.py permet de mettre en place la compilation à ne pas effacer

Lorsque vous importez le package, Python exécute d'abord le code à l'intérieur __init__.py.

Soit le paquetage appelé mypackage

À l'intérieur de mypackage , vous pouvez avoir deux fichiers :

module1.py
module2.py

Si vous ajoutez un fichier __init__.py au dossier mypackage, vous pouvez contrôler ce qui se passe lorsque quel qu'un importe mypackage,. Par exemple, vous pouvez ajouter le code suivant à __init__.py:

```
# __init__.py
from .module1 import MyClass
from .module2 import useful_function

__all__ = ['MyClass', 'useful_function']
```

Importations pour l'utilisateur :

Au lieu de demander aux utilisateurs d'importer des choses à partir de module1 et module2 séparément, l'utilisateur peut simplement faire :

```
From mypackage
import MyClass, useful_function
```

Sans __init__.py , vous devrez peut-être importer des parties de votre package comme suit:

```
From mypackage.module1 import MyClass
From mypackage.module2 import useful_function
```

17 Les Classes

En programmation structurelle les données ne sont pas corrélés aux codes

~~poids,age= f1(age,poids)~~

une collection permet de factoriser les données

Une fonction permet de factoriser du code

L'objet permet d'associer le code et les données

En événementielle l'objet à les caractéristiques suivantes :

@ mémoire	→	unique
Données invisible	→	Etat
code	→	Comportement
x	→	autonome
En mémoire	→	Cycle de vie (naissance vie mort

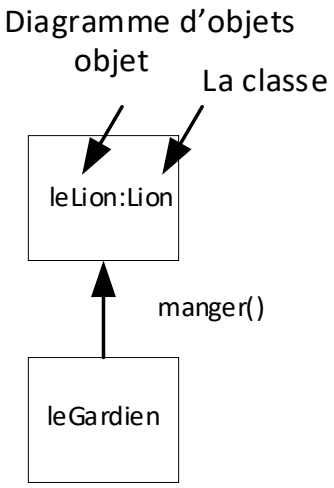
L'handle ou le nom de l'objet est une variable sur l'adresse de l'objet

Une classe est un concept c'est un type

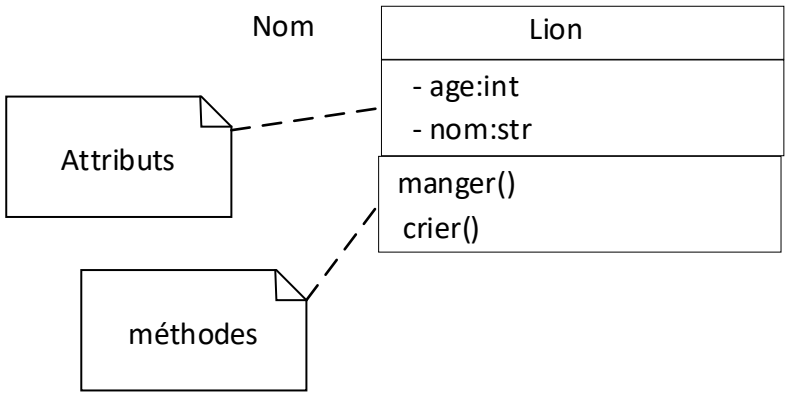


Instanciation
Représentant d'un type

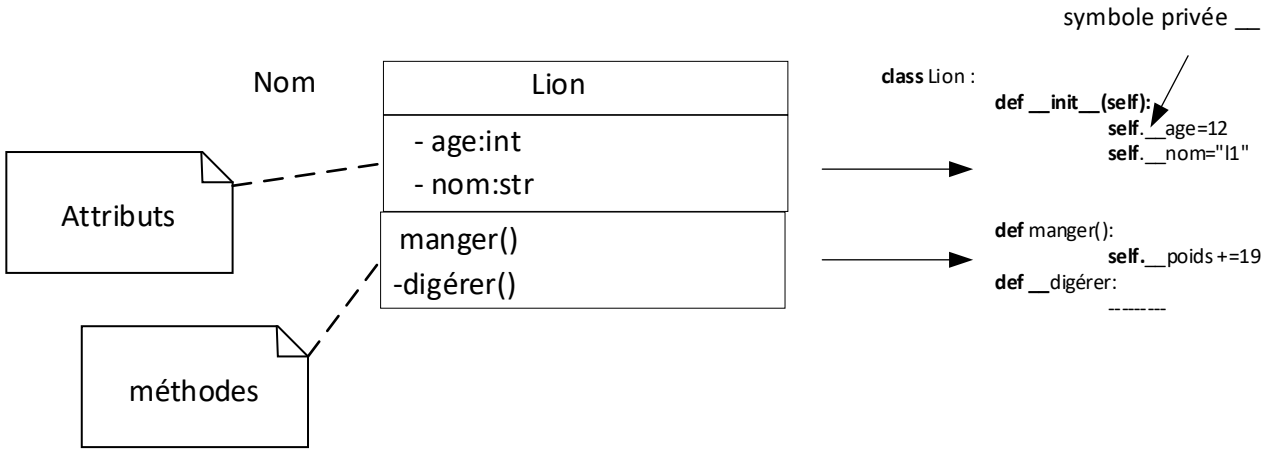
Les méthodes vont permettre de symboliser le comportement



En UML représentation de la classe



Le mot clé self veut dire moi-même (this en java)



Les attributs d'instance se déclarent dans la méthode constructeur __init

La méthode __init permet d'initialiser les attributs d'instance

monLion = Lion()
monLion.age crée un nouveau membre dans l'objet
~~monLion.__age=12~~ attribut privée

2 méthodes d'administration pour les tests une méthode accesseur pour lire les attributs getAge()

- ✓ Une méthode successeur pour écrire dans les attributs __setAge)__()
- ✓ Méthode accesseur public méthode accesseur privée

Les quelques concepts à retenir :

Le nom des classes est donné en camel case nom commençons par une majuscule et séparation entre les mots indiquées par une majuscule

Exemple :

Class MaClasse(object) :
    Définition de la classe MaClasse
Contrairement à java ou le nom de la classe doit porter le nom du fichier ici aucune nécessité.

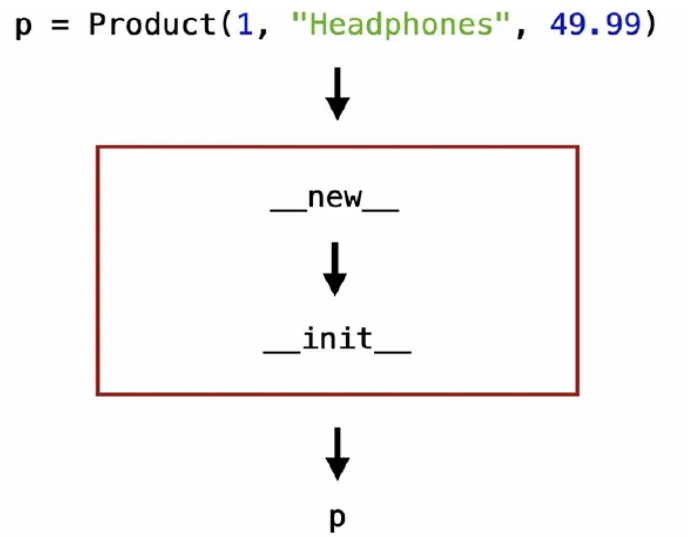
Les attributs d'instance seront déclarés dans la méthode constructeur de la classe pour les initialiser par le constructeur.

Tous les objets héritent d'une classe mère **object**

Les attributs sont attachés à l'objet courant ils commenceront par le mot clef **self** pour les distinguer des variables locales ou attribut de classe.
La méthode constructeur est définie en utilisant un nom de méthode spéciale et réservé :

__init__

Un constructeur a les tâches importantes suivantes. Il indique à Python de créer un objet à un emplacement mémoire. Il prend les arguments nécessaires au bon fonctionnement de l'objet et les stocke dans des attributs de données.
deux méthodes sont appelées dans l'ordre suivant. Tout d'abord, __new__. __new__ alloue de la mémoire à un objet et renvoie une référence à son adresse mémoire. Lorsque la classe définit __init__, la référence lui est alors transmise. Il peut maintenant initialiser l'objet. Enfin, l'instance est renvoyée au code appelant.



Le constructeur d'une classe Python est la combinaison de __new__ et de __init__. Comme il n'y a que quelques raisons d'implémenter __new__, dans la plupart des cas, vous n'implémenterez que __init__ et c'est pourquoi de nombreux tutoriels désignent __init__ comme le constructeur.
Les méthodes de la classe sont définies comme des fonctions ou **le premier argument est obligatoirement self**. cet argument fait référence à l'instance de l'objet sur lequel la méthode a été appelée.
Une méthode appelée depuis sa classe de définition **devra être précédée de self**

Par défaut les attributs d'instance et méthodes sont publics il faudra préfixer leur nom avec les **deux Under score __ pour les rendre privées (**attention Python ne prend pas en charge l'encapsulation comme d'autre langage le __ permettra d'appeler le membre ou la méthode privée dans la classe mais pas à l'extérieur de la classe **)**

Les attributs de classe ou attributs statiques sont déclarés en dehors de toute méthode. Pour déclarer une méthode statique il faudra utiliser le décorateur **@staticmethod**

Les méthodes de classe ou méthodes statiques n'ont accès qu'aux attributs statiques ces méthodes sont appelées sans instance de la classe.

```
class Clavier(object):
   attributdeclasse=42            #membre jamais initialisé par le constructeur attributs statiques
   __membreprivee=3                        # attributs statiques privées
   def __init__(self,c,dispo,n):  # ne pas oubliez l'argument self méthode constructeur
      self.__couleur = c    # membre ou attribut privé
      self.disposition = dispo    # membre ou attribut public
      self.nb_touche = n
      self.marque = None
```

```
  def appuyersurtouche(self,touche):
     print(f"l'utilisateur a appuyé sur la touche {touche}")
      print("la valeur de couleur est {}".format(self.__couleur))
     print(f"la valeur de nb_touche est {self.nb_touche}")
     print(f"la valeur de attributdeclasse est {self.attributdeclasse}")
     print(f"la valeur de __membreprivee est {self.__membreprivee}")
     self.__mamethodeprivee()

  def __mamethodeprivee(self):
     print("je suis une methode privee")


  @staticmethod     # methode static
  def affiche():
     print(Clavier.attributdeclasse)
     print(self.nb_touche) # n'accède qu'aux membres statiques

monclavier=Clavier("noir","azerty",115)
monclavier.appuyersurtouche(2)
e=monclavier.nb_touche
print(e)
e=monclavier.attributdeclasse
print(e)
Clavier.attributdeclasse=43
#clavier.__membreprivee=2  attribut de classe ou statique privée
#monclavier.__mamethodeprivee()        méthode privée
#d=monclavier.self.__couleur            membre privée
Clavier.affiche()
```

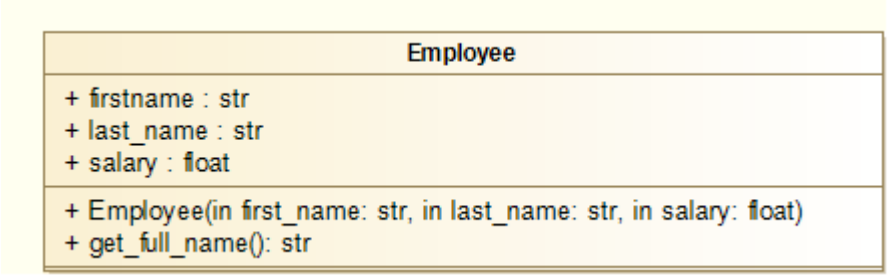
Il n'est pas nécessaire de mettre le décorateur @staticmethod pour Les méthodes d'instance La différence repose sur la présence ou pas du mot clef self

Quand Python détecte un attribut avec __ il modifie le nom pour y accéder (mangling)
d=monclavier._Clavier__couleur # on peut y accéder

```
class Employee :
        def __init__(self, first_name, last_name, salary):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary

        def get_full_name ( self ):
        return f" {self.first_name} {self.last_name} " # retourne une chaine formatée

e1 = Employee( "Vera" , "Schmidt" , 2000 )
e2 = Employee( "Chuck" , "Murphy" , 1800 )
e3 = Employee( "Dave" , "Dreissing" , 1900 )
print (e1.get_full_name())
print (e2.get_full_name())
print (e3.get_full_name())
```



L'attribut protégé

Un attribut protected signifie qu'il ne peut être lu ou modifié que par une classe enfant par le concept d'héritage.

• La convention est de préfixer l'attribut par un _ pour signifier qu'il est protected

• Cependant Python ne restreint pas l'accès à ces attributs protégés. Ce sont les développeurs, par convention, qui s'interdiront l'accès à ces attributs

```
class Employee:
        def __init__(self, name, sal):
        self._name=name # protected
        self._salary=sal # protected
```

```
#La création d'objets à partir d'une classe s'appelle l'instanciation de classe

e1=Employee("Nicolas", 10000)
e1._salary
# 10000
e1._salary=20000
e1._salary
# 20000
```

Les attributs de classe

Python fournit des attributs de classe intégrés que nous pouvons utiliser pour obtenir plus d'informations sur une classe.

__dict__: que vous connaissez certainement, c'est un dictionnaire constitué de l'espace de noms de la classe.

```
print(Clavier.__dict__)
```

{'__module__': '__main__', 'attributdeclasse': 43, '_Clavier__membreprivee': 3, '__init__': <function Clavier.__init__ at 0x00000174F90EA3B0>, 'appuyersurtouche': <function Clavier.appuyersurtouche at 0x00000174F90EA440>, '_Clavier__mamethodeprivee': <function Clavier.__mamethodeprivee at 0x00000174F90EA560>, 'affiche': <staticmethod(<function Clavier.affiche at 0x00000174F90EA680>)>, '__dict__': <attribute '__dict__' of 'Clavier' objects>, '__weakref__': <attribute '__weakref__' of 'Clavier' objects>, '__doc__': None}

__doc__ : c'est la classe docstring, mais vaut None s'il n'y a pas de docstring ou un commentaire dans la classe.

__name__ : c'est le nom de la classe.

```
print(Clavier.__name__)
'Fruit'
```

__module__ : représente le nom du module dans lequel la classe est définie. Et comme nous somme en mode interactif, c'est main.

```
print(Clavier.__module__)
```

```
'__main__'
```

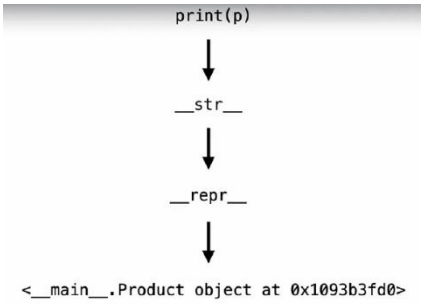
Exemple d'utilisation par défaut des méthodes de classe

```
class Product :
        def __init__(self, product_id, name, price):
                self.product_id = product_id
                self.name = name
                self.price = price

products = [Product(1,"Headphones",49.99),Product(2,"Monitor speaker",150.0),Product(3,"Soundcard",78.5)]

for p in products:
print (p)
```

Python se replie sur l'implémentation de la méthode de classe par défaut, qui renvoie le type et l'adresse de l'objet.



La destruction

Tout comme le constructeur il est possible de surcharger le destructeur

- C'est la méthode __del__
- Elle est appelée automatiquement lors de l'utilisation de del sur un objet

```
class Dog(object):
        "This is a Dog class"
        def __init__(self, name: str):
        self.name = name
        def __del__(self):
        print("Delete the dog")
dog = Dog('Buddy')
del dog
# Delete the dog
```

18 L'Héritage

L'héritage est spécifié dès la création de la classe **entre parenthèse**. **La fonction super()** est utilisée pour faire référence à l'objet parent. Nous pouvons appeler la méthode super() pour accéder aux attributs et aux méthodes de la classe parente.
En effet, super() est une fonction prédéfinie en python. Par conséquent, elle permet à la classe enfant d'appeler le constructeur, les variables et les méthodes de la classe de base ou la super-classe.
En faisant référence à la classe de base, nous n'avons pas besoin d'écrire explicitement son nom.

```
class A(object):
  def __init__(self):
     print("Le constructeur de la class de base A")
  def mamethode(self):
     print("A est la classe mere")


class B(A):
  def __init__(self):
     print("Le constructeur de la class enfant B")
     super().__init__()

  def mamethode(self):
     print("B est la classe fille")

objb=B()
objb.mamethode()
```

on obtient :

```
Le constructeur de la classe enfant B
Le constructeur de la classe de base A
B est la classe fille
```

Si on oublie d'appeler le constructeur de la classe parent les attributs d'instance du ou des parents ne seront pas créés

Overriding

La classe B a hérité de la classe A et nous avons la même méthode mamethode dans les deux classes. La méthode de la classe A a été remplacée par la méthode de la classe B.
En réalité, les recherches dans un héritage vont des instances aux sous-classes aux superclasses, s'arrêtant à la première apparition du nom d'attribut trouvé. Par conséquent, le nom méthodes de la classe B a été trouvé avant celui de la classe A. En terme de programmation, on dit que la classe B a surchargé la méthode de la classe A.
De plus, sachez que la personnalisation introduite dans B est complètement externe à A. Autrement dit, cela n'affecte pas les objets existants ou futurs de A.
Tout ce dont vous avez besoin pour surcharger une méthode est de créer une méthode portant le même nom dans la sous-classe.

```
Class Employee :
        Def __init__ (self, name, salary):
                self.name = name
                self.salary = salary

        def get_info ( self ):
                return f"Employee {self.name} , salary {self.salary}"

class Manager (Employee):
        def get_info(self):
                return f"Manager{self.name}, salary{self.salary}"

employees = [Manager("Vera",2000),Employee("Chuck",1800),Employee("Dave",1900),]
for e in employees:
        print (e.get_info())
```

la classe Manager a surchargé la méthode get_info de la classe Employee.

En java C++ la méthode get_info doit être abstraite ou virtuelle

19 La surcharge de méthodes en Python (Overloading)

Contrairement à d'autres langages de programmation comme C, C ++ ou Java, **deux méthodes ne peuvent pas avoir le même nom en Python**. Cependant, la surcharge de méthode en Python est une fonctionnalité qui permet au même opérateur d'avoir des significations différentes.
Par conséquent, vous pouvez avoir une méthode **et une seule du même nom** qui a zéro, un ou plusieurs paramètres. Un nombre de paramètre variable pour une seule méthode n'est pas vrai en C, C ++ ou Java

Créer deux fonctions avec le même nom.

```
class B(A):

  def __init__(self):
    print("Le constructeur de la class enfant B")
    super().__init__()

  def mamethode(self):
    print("B est la classe fille")

  def mamethode(self,ch:str): # impossible en python
    print(f"ma chaine est {ch}")

objb=B()
objb.mamethode()
objb.mamethode("moi")
```

En fait, ces deux derniers concept sont connus sous le nom (overriding et overloading) de polymorphisme. Ce qui signifie, plusieurs formes ou lorsque la même méthode est déclarée plusieurs fois, à des fins multiples et dans différentes classes Python.
On peut résoudre le manque de polymorphisme par :

```
def mamethode(self,ch=None):
  if ch is not None:
    print(ch)
```

```
else:
    print("aucun parametre")
```

```
class Point(object):
    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add(self,p2):
        return Point(self.x+p2.x,self.y+p2.y)

    def __str__(self):
        print(f"{self.x},{self.y}")
```

```
class Polygon(object):

    def __get_pts(self):
        return self.__pts

    def __set_pts(self,lst):
        self.__pts=lst
        #print(self.__pts)
    pts = property(__get_pts, __set_pts)

    def __init__(self,*Points):     #*points veut dire nombre d'arguments non connus renvoie un tuple de un triangle + 3point
        self.__set_pts(Points)
        print(self.__pts)


    def display(self):
        for i in self.__get_pts():
            print(f"{i}")
```

```
class Triangle(Polygon):

    def __init__(self,p1,p2,p3):
        #Polygon.__init__(self,p1,p2,p3)
        super().__init__(self, p1, p2, p3)

    def display(self):
        print("redefinition de display() dans triangle")
        #Polygon.display()
        super().display()
```

```
p1=Point(1,2)
#p1.__str()    # méthode privée impossible à appeler
t=Triangle(Point(1,5),Point(3,4),Point(5,6))
t.display()
```

20 Les modèles de conception (pattern design) pour les classes

20.1 La classe singleton

le motif Singleton garantit que, quel que soit le nombre de fois que vous essayez de créer une instance d'une classe, une seule instance sera créée. Si vous essayez d'en créer une autre, il vous remet simplement l'instance existante.

```
class Singleton:
   _instance = None  # This is where the one-and-only instance lives

   def __new__(cls, *args, **kwargs):
      if cls._instance is None:
         print("Creating the singleton instance... because one is enough. 😎")
         cls._instance = super(Singleton, cls).__new__(cls)
      return cls._instance

   def __init__(self, value):
      if not hasattr(self, 'initialized'):  # Prevents reinitialization
         self.value = value
         self.initialized = True

# Testing it out:
s1 = Singleton("First Instance")
s2 = Singleton("Second Instance")

print(s1.value)  # Output: First Instance
print(s2.value)  # Output: First Instance
```

20.2 les décorateurs de classe

Le motif Decorator vous permet d'ajouter de nouvelles fonctionnalités à un objet sans modifier son code existant. Vous « enveloppez » votre objet original avec un objet décorateur qui implémente la même interface. Chaque décorateur ajoute ses propres fonctionnalités, et vous pouvez les empiler selon vos besoins.

```
class Coffee:
   def cost(self):
      return 5

   def description(self):
      return "Simple coffee"

# Decorator to add milk
class MilkDecorator:
   def __init__(self, coffee: Coffee):
      self.coffee = coffee

   def cost(self):
      return self.coffee.cost() + 1

   def description(self):
      return self.coffee.description() + " with milk"
```

```
# Decorator to add sugar
class SugarDecorator:
    def __init__(self, coffee: Coffee):
        self.coffee = coffee

    def cost(self):
        return self.coffee.cost() + 0.5

    def description(self):
        return self.coffee.description() + " with sugar"

# Start with plain coffee:
my_coffee = Coffee()
print("Base:", my_coffee.description(), "Cost:", my_coffee.cost())

# Add milk:
my_coffee = MilkDecorator(my_coffee)
print("After milk:", my_coffee.description(), "Cost:", my_coffee.cost())

# Add sugar:
my_coffee = SugarDecorator(my_coffee)
print("After sugar:", my_coffee.description(), "Cost:", my_coffee.cost())
```

Les décorateurs vous permettent d'étendre le comportement d'un objet sans modifier sa structure de base. Ils sont idéaux lorsque vous avez besoin d'ajouter des fonctionnalités de manière dynamique, comme l'ajout de nouvelles fonctionnalités à un module logiciel sans réécrire le code d'origine.

20.3 Les observateurs de classe

Le modèle Observer est une question de communication entre les objets. Lorsqu'un objet change, il notifie un groupe d'autres objets susceptibles d'être intéressés par cette modification. C'est comme une mise à jour de statut dans le canal de discussion de votre projet : lorsqu'un module est mis à jour, les autres le savent immédiatement.
Un sujet tient à jour une liste d'observateurs et les avertit automatiquement lorsqu'un événement se produit. De cette façon, vos composants de code restent faiblement couplés et vous pouvez ajouter ou supprimer des observateurs sans trop de problèmes. Le modèle Observer est inestimable pour les applications pilotées par les événements. Il permet aux différentes parties de votre programme de réagir aux changements sans être étroitement interconnectées. Cela facilite la gestion et l'extension de votre code, en particulier lorsque vous travaillez sur de grands projets avec plusieurs composants.

```
class Subject:
    def __init__(self):
        self._observers = []  # List to store observers

    def register(self, observer):
        self._observers.append(observer)

    def notify_all(self, message: str):
        for observer in self._observers:
            observer.notify(message)

class Observer:
    def notify(self, message: str):
        print(f"Observer received: {message}")

# Setting up observers:
```

```
subject = Subject()
observer1 = Observer()
observer2 = Observer()

subject.register(observer1)
subject.register(observer2)

subject.notify_all("Hello, Observers!")  # Everyone gets the update!


#Observer received: Hello, Observers!
#Observer received: Hello, Observers!
```

21 méthodes dunder

La maîtrise des méthodes dunder vous donne la possibilité de créer un code pythonique très flexible qui interagit de manière transparente avec les fonctionnalités de base de Python. Qu'il s'agisse de créer des conteneurs personnalisés, de surcharger des opérateurs ou de gérer des ressources, les méthodes dunder vous permettent de contrôler le comportement de vos objets dans pratiquement toutes les situations.

La méthode __str__

Elle fournit une représentation de chaîne lisible par l'homme de votre objet.

```
class Book:
    def __init__(self, title, author):
            self.title = title
            self.author = author

    def __str__(self):
            return f"'{self.title}' by {self.author}"

book = Book("1984", "George Orwell")
print(book)  # Output: '1984' by George Orwell
```

La méthode __add__

Cette méthode vous permet de définir le comportement de vos objets de classe lors de l'utilisation de l'opérateur.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)  # Output: Vector(4, 6)
```

La méthode __repr__

La méthode est cruciale pour fournir une représentation officielle et sans ambiguïté d'un objet qui peut être utile lors du débogage. Il vous permet de voir clairement les composants internes de vos objets dans la console.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

point = Point(3, 4)
print(repr(point))  # Output: Point(x=3, y=4)
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, aged {self.age}"

    def __repr__(self):
        return f"Person('{self.name}', {self.age})"


# Usage
p = Person("Alice", 30)
print(str(p))  # Output: Alice, aged 30
print(repr(p)) # Output: Person('Alice', 30)
```

__del__: Effectuer un nettoyage personnalisé lorsqu'un objet est détruit. Utile pour libérer des ressources ou gérer la fermeture d'un fichier.
__eq__, , : Remplacer les opérations de comparaison telles que l'égalité (), inférieur à () et supérieur à ().__lt____gt__==<>

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __eq__(self, other):
        return self.title == other.title and self.author == other.author


# Usage
book1 = Book("1984", "George Orwell")
book2 = Book("1984", "George Orwell")
print(book1 == book2)  # Output: True
```

__hash__: Définissez comment votre objet est haché, utile pour utiliser des objets comme clés de dictionnaire ou dans des ensembles.
__iter__: Implémentez l'itération sur un objet, transformant votre classe en itérable.

__next__ : Pour récupérer l'objet itérateur suivant dans le flux de données

```
class Counter:
   def __init__(self, low, high):
      self.current = low
      self.high = high

   def __iter__(self):
      return self

   def __next__(self):
      if self.current > self.high:
         raise StopIteration
      else:
         self.current += 1
         return self.current - 1

for num in Counter(1, 5):
   print(num)
```

__bool__: Contrôle de la valeur de vérité d'un objet, utile pour la logique booléenne personnalisée.

22 L'héritage multiple des classes en Python

Dans l'héritage multiple, Python nous permet d'hériter de plus d'une classe.

```
class A:
   def methode_A(self):
      print("Je suis la classe A")

class B:
   def methode_B(self):
      print("Je suis la classe B")



class C(A, B):
   def methode_C(self):
      print("Je suis la classe C et j'hérite des classe A et B.")

objc= C()
objc.methode_B()
```

Les classes abstraites

En Python le type Interface n'existe pas. Nous utilisons des classes abstraites

Partons de l'exemple suivant :

```
Class Employee :
        Def __init__ (self, name, salary):
                self.name = name
                self.salary = salary

        def get_info ( self ):
                return f"Employee {self.name} , salary {self.salary}"

class Manager (Employee):
        def get_info(self):
                return f"Manager{self.name}, salary{self.salary}"

class Programmer (Employee):
        def get_info ( self ):
                return f"Programmer {self.name} , salary {self.salary} "

employees = [
Manager("Vera",2000),
Programmer("Chuck",1800),
Programmer ("Dave",1900)]

for e in employees:
        print (e.get_info())
```

où l'employé est-il instancié ? La réponse est nulle part . La classe d'employée n'est utilisée que pour hériter. Cela rend obsolète une méthode de ce code.
Supprimons là .

Un autre développeur ajoute une nouvelle employée, Wendy, à la liste des employés, mais ne sait pas si Wendy est responsable ou programmeuse. Wendy est donc ajoutée en tant qu'instance d'employée.

```
employees = [
Manager("Vera",2000),
Programmer("Chuck",1800),
Programmer("Dave",1900),
~~Employee("Wendy",2100)]~~
```

Cela ne fonctionne plus. La classe employee ne doit plus être instanciée. Pour éviter qu'une classe ne soit instanciée, elle doit être rendue abstraite.

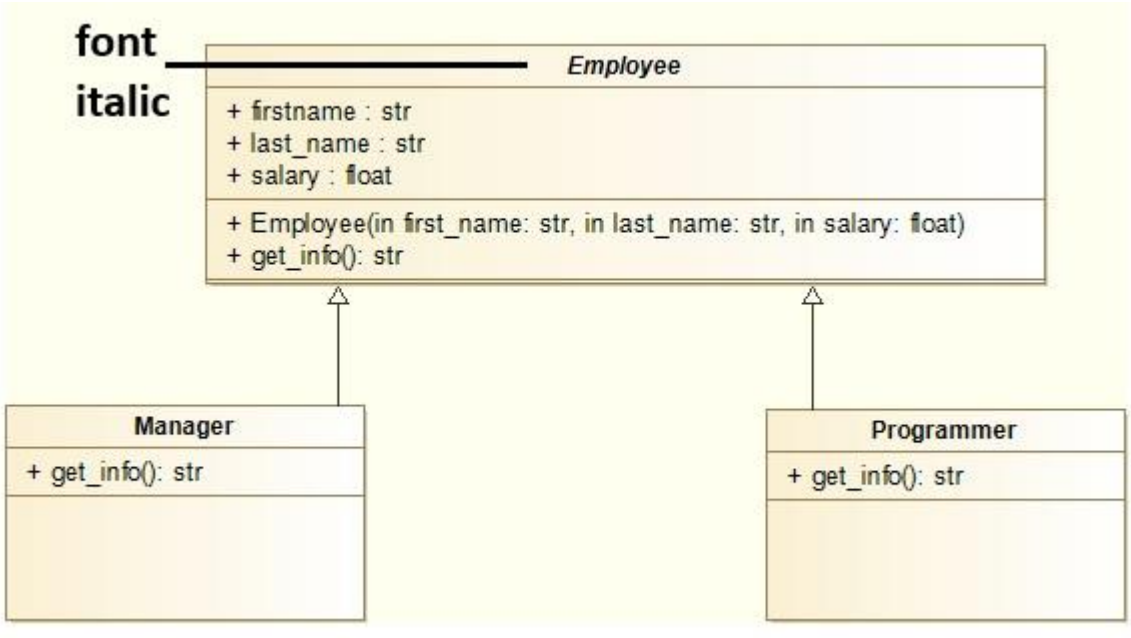
Les classes abstraites sont des classes qui ne **peuvent pas être instanciées**, elles contiennent **une ou plusieurs méthodes abstraites**. C'est un modèle pour d'autres classes qui héritent un ensemble de méthodes et de propriétés. Une méthode abstraite est une méthode déclarée, mais qui n'a pas d'implémentation (de définition). Toutefois, une classe abstraite nécessite **des sous-classes qui fournissent des implémentations pour les méthodes abstraites**.
il y a un deuxième avantage à créer une classe de base. Les sous-classes d'une classe abstraite peuvent être forcées d'implémenter des méthodes.

```
class Employee :

        def __init__ (self, name, salary):
                self.name = name
                self.salary = salary
        @abstractmethod
        def get_info ( self ):
                pass # la classe Employee devient abstraite

class Manager (Employee):
        def get_info(self):
                return f"Manager{self.name}, salary{self.salary}"

class Programmer (Employee):
        def get_info ( self ):
                return f"Programmer {self.name} , salary {self.salary} "
```



L'abstraction est très utile lors de la conception de systèmes complexes pour **limiter la répétition et assurer la cohérence**. C'est similaire à une interface dans d'autres langages.

Tout d'abord, commençons par définir trois classes Lion et Girafe :

```
class Lion(object):
   def lionsedeplace(self):
      print("lelion se déplace")

class Girafe(object):
   def girafesedeplace(self):

      print("la girafe se se déplace")
```

Tous les animaux peuvent se déplacer

```
objlion= Lion()
objlion.lionsedeplace()
objgirage= Girafe()
objgirage.girafesedeplace()
```

Mais imaginez combien de temps cela prendrait pour chaque animal d'un grand zoo, en répétant le même processus et en codant des centaines de fois. Cela rendrait également le code plus difficile à maintenir.
En fait, afin d'optimiser le processus, la structure de notre programme pourrait ressembler à ceci :

```
# mettre les animaux dans une liste

zoo=[objlion,objgirafe]

# déplacer les animaux à travers une boucle

for i in  zoo:
        #mais quelle méthode à utiliser
```

Le problème est que chaque classe a un nom de méthode différent, quand on déplace un lion, c'est la méthode lionsedeplace (), quand on déplace une girafe, c'est girafesedeplace().

Ce code est un gâchis, car les méthodes qui font la même chose doivent avoir le même nom.
Si nous pouvions seulement forcer nos classes à implémenter les mêmes noms de méthodes.

Il s'avère que la classe abstraite est ce dont nous avons besoin. Puisqu'elle force ses sous-classes à implémenter toutes ses méthodes abstraites.
En créant une classe abstraite Animal, chaque classe enfant qui hérite de la classe Animal doit implémenter des méthodes abstraites de Animal, qui dans notre cas est la méthode sedeplacer().

Nous avons importé le module intégré abc. Lors de la définition d'une classe abstraite, nous avons hérité de la classe abstraite de base – ABC. Pour définir une méthode abstraite dans la classe abstraite, nous allons utilisé un décorateur **@abstractmethod.**

Si vous héritez de la classe Animal, mais n'implémentez pas les méthodes abstraites, vous obtiendrez une erreur.
Si nous essayons d'instancier la classe, cela générera un TypeError,

```
class Animal(ABC): # hériter de ABC(Abstract base class)
   @abstractmethod  # un décorateur pour définir une méthode abstraite
   def sedeplacer(self):
      pass
```

```
class Lion(Animal):
    def sedeplacer(self):
        print("le lion se déplace")

class Girafe(object):
    def sedeplacer(self):

        print("la girafe se se déplace")

objlion= Lion()

objgirafe= Girafe()

zoo=[objlion,objgirafe]

for i in zoo:
    i.sedeplacer()
```

La fonction property

L'écriture des méthodes accesseur et successeur est un peu lourde. Python propose deux écritures (même code interne) permettant d'enfouir les accesseurs et modifieurs et de les rendre invisibles pour le développeur extérieur :

• la fonction property : cette fonction va créer des alias vers les méthodes d'accès et de modification. Cette fonction sera appelée après chaque définition de paire accesseur/modifieur en lui indiquant quelle est la méthode employée pour accéder a I 'attribut et quelle est la méthode pour modifier I 'attribut :

En passant les méthodes et get_x et set_x en visibilité privée et en créant une propriété nommée x utilisant ces méthodes, nous avons défini a un attribut x accessible pour le développeur qui ne travaille pas directement sur le code de la classe

```
class Point3d(object):
    def __get_x(self):
        return self.__x

    def __set_x(self, x):
        self.__x = x
        # print(self.__pts)

    pts = property(__get_x, __set_x)

    def get_y(self):
        return self.__y

    def set_y(self, y):
        self.__y = y
        # print(self.__pts)

    def __init__(self,x,y,z):
        self.__x=x
        self.__y=y
        self.__z = z
```

• les décorateurs property et setter : il s'agit de fonctions spéciales (leur appel commence par un 0) qui s'exécutent << pardessus >> la fonction qu'ils précèdent (ou décorent). L'utilisation des décorateurs ne modifie ici que L'écriture du code, le résultat étant le même

```
@property
   def z(self):
    return self.__z
   @z.setter
   def z(self,z):
   self.__z=z


P1=Point3d(1,2,3)
P1.__x=3       # appel méthode successeur par property
print(f" la valeur de x {P1.__x}")   # appel méthode accesseur par property
P1.__z=4       # appel méthode successeur par setter
print(f" la valeur de z {P1.__z}")   # appel méthode accesseur par property
P1.set_y(4)
print(f" la valeur de y {P1.get_y()}")
```

L'objet spécial None

Il existe un objet spécial pour exprimer l'absence d'objet

• Le mot clef None est utilisé
• Il peut être utilisé dans une égalité

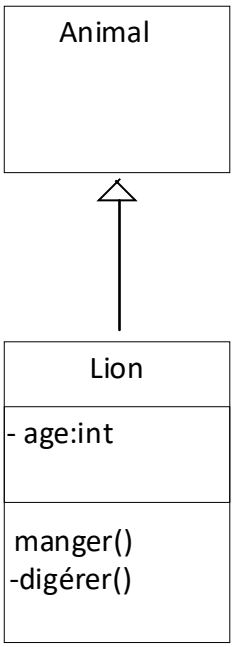
```
if my_object == None:
        print('There is no object')
```

23 Les Exceptions

Les exceptions utilise le concept d'héritage

Notion d'héritage :

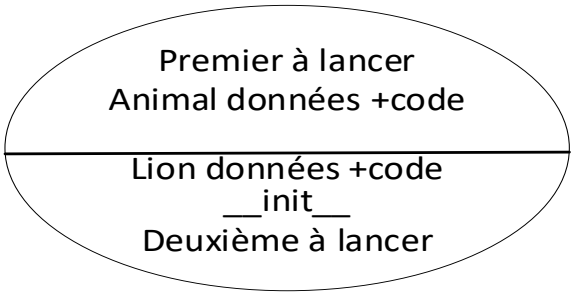
Créer un objet du type exception → lancer (gérer toutes les erreurs possibles) -→ Attraper (la capture de l'exception)



Appel constructeur classe mère

```
class Lion (Animal) :
        def __init__(self,age:int,nom:str):
                super(Animal,self).__init__()
                __age=12
                __nom="l1"

        def manger():
                self.__poids +=19
        def __digérer:
                --------
```



Déclaration et appel de l'exception:

La capture est optionnelle (try …except) en python mais fortement recommandé
Pour pouvoir récupérer une exception et la traiter, il faut que l'instruction ayant levée l'exception se trouve dans un bloc d'analyse défini par try. La partie traitement se trouve dans un bloc except et il est possible d'ajouter un bloc finaly indiquant du code a exécuter que l'exception soit levée ou non

```
try:
   #instructions susceptible de lever un instruction
except:
   #instructions à executer lors de l'exception
finally:
   # code executer exception ou pas
```

Dans le modèle précèdent, toutes les exceptions sont traitées dans le même bloc. Pour effectuer un traitement plus fin, il faut nommer les exceptions pour chaque bloc except :

```
try:
    #instructions susceptible de lever un instruction
except (IOerror,ZeroDivision):
    #instructions à executer lors de l'exception IOerror,ZeroDivision
except TypeError:
    #instructions à executer lors de l'exception TypeError
except:
    # traitement des autres erreurs
```

Ces instruction permettent de récupérer les exceptions standards.

Si vous souhaitez lever une exception (**raise**) il vous faudra créer une classe exception vous-même:

Déclaration et appel de l'exception:

Je crée une classe pour l'objet Exception

```
class MonException(Exception):
  def __init__(self, raison: str = "poids maxi dépassé"):
    super(MonException, self).__init__(raison)
```

Je lance et j'attrape

```
try:
  if p>230:
          raise MonException () # à partir de ce point return (raise est equivalent à throw)
except  MonException as e: #e objet exception intercepter
  print(e)
```

```
# -*- coding:utf-8 -*-
class MonException(Exception):
  def __init__(self,valeur1, raison: str = "poids maxi dépassé"):
    super(MonException, self).__init__(valeur1,raison)
    self.__valeur1=valeur1
  def afficheerreur(self):
    print(f"déclenchement à cause du poids {self.__valeur1}")


#Je lance et j'attrape

p=int(input("rentrez une valeur pour votre poids"))

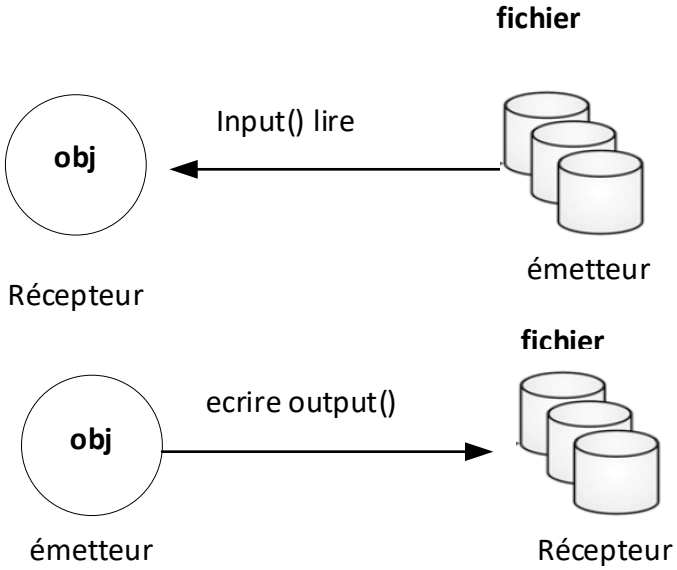
try:
  if p>230:
    raise MonException(p)
except MonException as e:
  print(e)
  e.afficheerreur()
finally:
  print("surpoids ou pas c'est pareil")
```

24 La sérialisation

2 types de flux :

- ✓ Le binaire pur
- ✓ Le bloc (ASCII UTF)

Le flux unidirectionnel



La fonction Open permet d'ouvrir et de retourner un handler sur File

fic = open("cheminetnomdufichier", mode)

mode :

"rb" lecture en binaire
"wb" écriture en binaire
"r" lecture d'un bloc
"w" écriture d'un bloc

Attention l'objet du type File (descripteur de fichier) fic se déplace

Ne pas oublier de fermer le fichier à la fin
fic.close()

L'écriture suivante permet de gérer les exceptions et la fermeture du fichier
with open(nomFichier, "r") as fic:

```
def lireTexte(nomFichier:str="exemple.txt")->list:
   with open(nomFichier,"r") as fic:# fermeture du flux et gestion des exceptions
      #print(fic.readlines())
      #list1=fic.readlines()
      return [ligne.strip("\n") for ligne in fic]
```

La méthode strip() supprime tous les caractères spécifié en paramètre d'entrée à droite et à gauche (l'espace est le caractère par défaut à supprimer si il n'y a pas de paramètre d'entrée)

25 Les fonctions génératrices et les iterables

25.1 les fonctions iter() et next()

La fonction iter() permet d'obtenir un itérateur à partir d'un objet. Concrètement, cette fonction appelle la méthode __iter__() de l'objet passé en paramètre.

La fonction next() attend un itérateur en paramètre et retourne l'élément suivant. Si l'itérateur est déjà positionné sur le dernier élément, cette fonction lève une exception de type StopIteration. Concrètement, cette fonction appelle la méthode __next__() de l'itérateur passé en paramètre.

```
it = iter(range(3))
print(next(it))
# affiche 0
print(next(it))
# affiche 1
print(next(it))
# affiche 2
print(next(it))
```

provoque une exception StopIteration

```
# list of vowels
vowels = ["a", "e", "i", "o", "u"]
# iter() with a list of vowels
vowels_iter = iter(vowels)
print(next(vowels_iter))
print(next(vowels_iter))
print(next(vowels_iter))
print(next(vowels_iter))
print(next(vowels_iter))
```

25.2 Les fonctions génératrices avec yield

Python dispose du mot-clé yield. Il permet de transformer une fonction en générateur. yield retourne l'élément suivant du générateur. Tout se passe comme si une instruction à yield suspendait l'exécution de la fonction qui se continuera au passage à l'élément suivant du générateur.

```
def ma_fonction():
   yield "un"
   yield "deux"
   yield "trois"

for x in ma_fonction():
   print(x)

# Affiche
#  un
#  deux
#  trois
```

Ainsi il est très facile d'implémenter la fonctionnalité identique à notre classe Compteur mais cette fois-ci sous la forme d'une fonction génératrice :

```
def compteur():
   cpt = 1
   while cpt <= 10:
      yield cpt
      cpt += 1
for x in compteur():
   print(x)
```

Affiche les nombres de 1 à 10

Une fonction génératrice est très souvent beaucoup plus simple à implémenter et à comprendre qu'un itérateur tout en permettant d'arriver au même résultat.

26 Les fonctions les plus usuelles en python

nom	***commentaire***	***exemple***
abs()	*Cette fonction renvoie la valeur absolue d'un nombre.*	*print(abs(-15)) # Output: 15*
all()	*Il est utilisé pour vérifier si tous les éléments d'un itérable sont .True*	print(all([True, 1, "Hello"])) # Output: True
any()	Il est utilisé pour vérifier si un élément d'un itérable est .True	print(any([False, 0, "Python"])) # Output: True
ASCII()	Il permet de renvoyer une version lisible d'un objet et d'échapper les caractères non ASCII.	print(ascii("Python 🐍")) # Output: 'Python\\U0001f40d'
bin()	Il convertit un nombre en sa représentation binaire.	print(bin(10)) # Output: '0b1010'
bool()	Il convertit une valeur en ou .TrueFalse	print(bool(0)) # Output: False print(bool("Hi")) # Output: True
bytearray()	Il renvoie une séquence mutable d'octets.	data = bytearray(5) print(data) # Output: bytearray(b'\x00\x00\x00\x00\x00')
bytes()	Il renvoie une séquence immuable d'octets.	data = bytes("Python", "utf-8") print(data) # Output: b'Python'
callable()	Il est utilisé pour vérifier si un objet est appelable (comme une fonction).	def func(): pass print(callable(func)) # Output: True
chr()	Il est utilisé pour convertir un entier en son caractère Unicode correspondant.	print(chr(66)) # Output: 'B'
compile()	Il compile le code source dans un objet de code pour l'exécution.	code = compile('print("Hello World")', 'test', 'exec') exec(code) # Output: Hello World
complex()	Il est utilisé pour créer un nombre complexe.v	print(complex(3, 5)) # Output: (3+5j)
delattr()	Il est utilisé pour supprimer un attribut d'un objet.	class MyClass: self.x = 10 obj = MyClass() delattr(obj, 'x')
dir()	Il renvoie une liste d'attributs et de méthodes d'un objet.	print(dir([]))
divmod()	Il est utilisé pour retourner le quotient et le reste d'une division.	print(divmod(10, 3)) # Output: (3, 1)
enumerate()	Il est utilisé pour ajouter un index à un itérable.	fruits = ["apple", "banana", "cherry"] for index, fruit in enumerate(fruits): print(index, fruit)
eval()	Il est utilisé pour évaluer une chaîne en tant que code Python.	code = "5 + 5" print(eval(code)) # Output: 10
filter()	Il est utilisé pour filtrer les éléments d'un itérable en fonction d'une fonction.	nums = [1, 2, 3, 4, 5] even = filter(lambda x: x % 2 == 0, nums) print(list(even)) # Output: [2, 4]
float()	Il est utilisé pour convertir un nombre ou une chaîne en flottant.	print(float("3.7")) # Output: 3.7

format()	Il est utilisé pour formater une valeur en une chaîne.	print(format(1234.5678, ".2f")) # Output: '1234.57'
getattr()	Il est utilisé pour obtenir la valeur d'un attribut d'un objet.	class MyClass: y= 10 obj = MyClass() print(getattr(obj, "y")) # Output: 10
hasattr()	Il est utilisé pour vérifier si un objet a un attribut spécifique.	class MyClass: y= 10 obj = MyClass() print(hasattr(obj, "y")) # Output: True
hash()	Il est utilisé pour retourner la valeur de hachage d'un objet (utile pour les clés de dictionnaire).	print(hash("Python")) # Output: A unique hash value
help()	Il est utilisé pour afficher la documentation d'un objet.	help(print)
hex()	Il est utilisé pour convertir un entier en une chaîne hexadécimale.	print(hex(255)) # Output: '0xff'
Id()	Il est utilisé pour renvoyer l'adresse mémoire d'un objet.	x = 5 print(id(x)) # Output: A unique memory address
isinstance()	print(isinstance(7, int)) # Output: True	print(isinstance(7, int)) # Output: True

issubclass()	Il est utilisé pour vérifier si une classe est une sous-classe d'une autre classe.	class A:
		  pass
		
		class B(A):
		  pass
		
		print(issubclass(B, A)) # Output: True

iter()	Il est utilisé pour retourner un itérateur à partir d'un itérable.	num = [1, 2, 3] itt = iter(num) print(next(itt)) # Output: 1
len()	Il est utilisé pour renvoyer la longueur d'un objet.	print(len("Pythonic")) # Output: 8
list()	Il est utilisé pour créer une liste à partir d'un itérable.	num = list(range(6)) print(num) # Output: [0, 1, 2, 3, 4, 5]

locals()	Il est utilisé pour renvoyer la table de symboles locale sous forme de dictionnaire.	def func():
		  x = 5
		  print(locals())
		
		func()
		# Output: {'x': 5}

map()	Il est utilisé pour appliquer une fonction à tous les éléments d'un itérable.	num = [1, 2, 3] squared = map(lambda x: x**2, num) print(list(squared)) # Output: [1, 4, 9]
max()	Il est utilisé pour retourner l'élément le plus volumineux dans un itérable.	print(max([4, 5, 8])) # Output: 8
min()	Il est utilisé pour retourner le plus petit élément dans un itérable.	print(min([1, 7, 9])) # Output: 1
next()	Il est utilisé pour récupérer l'élément suivant d'un itérateur.	num = iter([1, 2, 3]) print(next(num)) # Output: 1

oct()	Il est utilisé pour convertir un entier en une chaîne octale.	print(oct(8)) # Output: '0o10'
ord()	Il est utilisé pour convertir un caractère en son point de code Unicode.	print(ord('A')) # Output: 65
pow()	Il est utilisé pour renvoyer le résultat de l'élévation d'un nombre à une puissance.	print(pow(2, 3)) # Output: 8
reversed()	Il est utilisé pour retourner un itérateur inversé.	nums = [1, 2, 3] for num in reversed(nums): print(num)
round()	Il est utilisé pour arrondir un nombre à l'entier le plus proche ou aux décimales spécifiées.	print(round(3.14159, 2)) # Output: 3.14
sorted()	Il est utilisé pour créer un ensemble.	um = [8, 5, 9] print(sorted(num)) # Output: [5, 8, 9]
tried()	Il est utilisé pour renvoyer une liste triée d'itérables.	num = [8, 5, 9] print(tried(num)) # Output: [5, 8, 9]
zip()	Il est utilisé pour combiner deux ou plusieurs itérables en tuples.	names = ["Kiran", "Janvi"] ages = [25, 30] print(list(zip(names, ages))) # Output: [('Kiran', 25), ('Janvi', 30)]

27 Les principaux formats de fichiers de données et de configuration

YAML, XML, JSON, CSV, INI sont des noms de formats de données texte.
Les données :
carnet d'adresses, configuration d'un logiciel, contenu d'une base de données, etc. Bref, tout groupe d'informations que vous souhaitez pourvoir communiquer, ou relire plus tard.
Il est nécessaire ranger les données d'une certaine façons, c'est le format.

27.1 Le format XML

Dans le format XML, les données sont encadrés par des balises.

Les balises ont la forme suivante : `< nomDeBalise >`.

Les balises sont utilisées pour encadrer un contenu : il y a une balise ouvrante et une balise fermante.

27.2 Le format CSV

L'acronyme CSV signifie Coma Separated Values, littéralement valeurs séparées par des virgules.

La première ligne du fichier contient l'entête de la table, à savoir le nom de chacune des colonnes. Les lignes suivantes contiennent les données du tableau, en respectant l'ordre des colonnes. Le séparateur n'est pas forcément une virgule, on peut par exemple utiliser le point-virgule.

27.3 Le format INI

Le format INI, utilisé pour l'INItialisation, est surtout un format pour stocker des configurations de logiciel.

Il est constitué de deux types d'élément :

- Les en-têtes, ou sections, qui s'écrivent [nom_de_l_en_tete]. Ils servent à nommer et délimiter des groupes de valeurs.
- Les valeurs, qui s'écrivent nom=valeur.

27.4 Le format JSON

A la base JSON, qui signifie JavaScript Object Notation, n'était pas un format destiné à être échangé, mais seulement la représentation textuelle des objets Javascript.

Il se trouve qu'avec l'age d'or du Web, et l'utilisation massive d'AJAX, il a été utilisé pour communiquer entre le navigateur et le serveur,

Un document JSON est essentiellement un ensemble de paires constituées d'une étiquette et d'une valeur ou d'unliste de valeurs. Les paires sont placées entres accolades et séparées par des virgules. Les valeurs des listes sont placées entre crochets et séparées par des virgules.

27.5 Le format YAML

YAML, l'humoristique "Yet Another Markup Language" a des qualités similaires au JSON, mais le format est différent.

Voici le fichier INI, traduit en YAML (les espaces sont significatifs) :

Le format YAML est néanmoins plus riche que le JSON:

- Il permet d'inclure plusieurs document dans un seul fichier en les séparant par `---`
- Il contient des types avancés comme les dates.
- Il possède plusieurs manières d'écrire les textes multi-lignes

Le mutltithreading

Python inaugure un véritable multi-threading
Le verrouillage global de l'interpréteur (GIL) peut être supprimé. Dès lors, Python ne sera plus ce que l'on appelle le pseudo-multithreading.
Qu'est-ce que le GIL ? Le nom complet de GIL est Global Interpreter Lock, qui n'est pas unique à Python, mais un concept introduit lors de l'implémentation de CPython (interpréteur Python). Nous pouvons comprendre GIL comme un mutex, qui est utilisé pour protéger les objets en Python et empêcher plusieurs threads d'exécuter le bytecode Python en même temps, garantissant ainsi la sécurité des threads.
Cependant, GIL présente un inconvénient, c'est-à-dire qu'un seul thread peut s'exécuter sur un processeur à la fois, et que plusieurs threads ne peuvent pas être mappés à plusieurs processeurs, de sorte que Python ne peut pas obtenir une véritable simultanéité multithread, ce qui réduit l'efficacité de l'exécution.
Maintenant, l'équipe Python a officiellement accepté la proposition de supprimer le GIL et en a fait un mode optionnel, ce qui est bon pour les développeurs.
Cette contribution a été apportée par un ingénieur logiciel de Meta nommé Sam Gross, et il lui a fallu plus de quatre ans pour mener à bien le projet.
Après avoir appris la nouvelle, tout le monde a applaudi. Yann LeCun, l'un des trois géants du deep learning, a envoyé un message de félicitations : sans le GIL, le code Python peut désormais exécuter librement le multi-threading.