Le langage Python

1. Introduction

Python est un langage de programmation objet interprété. Son origine est le langage de script du système d'exploitation Amoeba (1990).

- ***Pour*** résumer Python en quatre points forts.
  - ***Qualité*** : L'utilisation de Python permet de produire facilement du code évolutif et maintenable et offre les avantages de la programmation orientée-objet.
  - ***Productivité*** : Python permet de produire rapidement du code compréhensible en reléguant nombre de détails au niveau de l'interpréteur.
  - ***Portabilité*** : La disponibilité de l'interpréteur sur de nombreuses plates-formes permet l'exécution du même code sur un PDA ou un gros système
  - ***Intégration :*** L'utilisation de Python est parfaitement adaptée l'intégration de composants écrit dans un autre langage de programmation (C, C++, Java avec Jython). Embarquer un interpréteur dans une application permet l'intégration de scripts Python au sein de programmes.
- Quelques caractéristiques intéressantes :
  - langage interprété (pas de phase de compilation explicite)
  - pas de déclarations de types (déclaration à l'affectation)
  - gestion automatique de la mémoire (comptage de références)
  - programmation orienté objet, procédural et fonctionnel
  - par nature dynamique et interactif
  - possibilité de générer du byte-code (améliore les performances par rapport à une interprétation perpétuelle)
  - interactions standards (appels systèmes, protocoles, etc.)
  - intégrations avec les langages C et C++
- Python, comme la majorité des langages dit de script, peut être utilisé aussi bien en mode interactif qu'en mode script / programme. Dans le premier cas, il y a un dialogue entre l'utilisateur et l'interprète : les commandes entrées par l'utilisateur sont évaluées au fur et à mesure. Pour une utilisation en mode script les instructions à évaluer par l'interprète sont sauvegardées, comme n'importe quel programme informatique, dans un fichier. Dans ce second cas, l'utilisateur doit saisir l'intégralité des instructions qu'il souhaite voir évaluer à l'aide de son éditeur de texte favori, puis demander leur exécution à l'interprète. Les fichiers Python sont identifiés par l'extension ***.py***.

2. Calcul avec python :

```
>>>a=10  # affectation de la valeur 10 à la variable a
>>> a, b, c,d=2 , 4, 12, 'abc'   # affectation au même temps les valeurs 2, 4, 12  et 'abc' aux variables a, b, c et d
>>> print('a= ',a," b= ",b," c= ",c, ," d= ",d)
        ('a= ', 2, ' b= ', 4, ' c= ', 12, d='abc'
>>> e=2*a+b
>>> e
8
```

Les opérateurs mathématiques:

+ : addition numérique ou concaténation des chaines de caractères

- : soustraction

***** : multiplication

/ : division

// : division entière

% : reste de la division

****** : puissance

Exemple :

```
>>> a=22      # a entier
>>> b=10      # b entier
>>> a+b
32
>>> a-b
12
>>> a*b
220
>>> a/b
2
>>> a//b
2
>>> a%b
2
>>>a**2
484
```

```
>>> a=22.0   # a réel
>>> b=10.0   # b réel
>>> a+b
32.0
>>> a-b
12.0
>>> a*b
220.0
>>> a/b
2.2
>>> a//b
2.0
>>> a%b
2.0
>>>a**2
484.0
```

```
>>> s1='abc'
>>> s2='2am'
>>> s=s1+s2
>>> s
'abc2am'
>>> s3=2*s1+3*s2
>>> s3
'abcabc2am2am2am'
>>> s3=s1*2+s2*4
>>> s3
'abcabc2am2am2am2am'
```

3. Les entrées Sorties (Lecture /Ecriture) :

3.1 La fonction print :

Pour afficher une variable ou un message on utilise la fonction print :

```
>>>a=10
>>>print(a)
10
>>>print(‘’ a = ‘’,a)
a =10
>>>b=2*a
>>>print(" a vaut ",a, " et son double b vaut ",b)
a  vaut  10 et son double vaut 20
>>>print("c\’est un message")
C’est un message
Pour afficher plusieurs messages :
>>>print("Bonjour","à","tous ")
Bonjouràtous
On peut remplacer  le séparateur des chaines par un espace ou
autre caractère :
>>> print("Bonjour", "à", "tous", sep ="*")
Bonjour*à*tous
>>> print("Bonjour", "à", "tous", sep ="")
Bonjouràtous
```

3.2 La fonction input :

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc.). Dans un script en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction intégrée **input()**. Cette fonction provoque une interruption dans le

programme courant. L'utilisateur est invité à entrer des caractères ou des valeurs numériques au clavier et à terminer avec <*Enter*>.

On peut invoquer la fonction **input()** en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
print("Veuillez entrer un nombre positif quelconque : ", end=" ")
ch = input()
nn = int(ch)              # conversion de la chaîne en un nombre entier
print("Le carré de", nn, "vaut", nn**2)
```

Remarque : La fonction **input()** renvoie toujours, à partir de la version 3 du python, une chaine de caractères (chaine numérique ou alphabétique ou alphanumérique) pour lire un entier ou un réel, on doit trantyper la valeur lue au clavier par la fonction input().

Remarque : dans les versions les plus anciennes la fonction input() traite la valeur saisie par l'utilisateur (entier : int ou réel : float) et n'accepte pas de valeurs caractères ou chaine de caractères, alors la fonction raw_input() retourne toujours une chaine de caractères.

Exemple pour la version 2.7 du python:

```
>>> b=input(" Entrer une valeur : ")
 Entrer une valeur : 14
>>> type(b)
<class 'int'>
>>> b=input(" Entrer une valeur : ")
 Entrer une valeur : 12.5
>>> type(b)
<class 'float'>
>>> b=input(" Entrer une valeur : ")
 Entrer une valeur : bonjour
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    b=input(" Entrer une valeur ")
  File "<string>", line 1, in <module>
NameError: name 'bonjour' is not defined
>>> b=raw_input(" Entrer une valeur : ")
 Entrer une valeur : bonjour
>>> type(b)
<class 'str'>
>>> b=raw_input(" Entrer une valeur :")
 Entrer une valeur : 21
>>> type(b)
<class 'str'>
```

Remarque : pour convertir, en entier ou en réel, une variable lue avec **raw_input ,** on utilise les fonctions **int()** ou **float()**

Exemple :

```
>>> b=raw_input(" Entrer une valeur :")
 Entrer une valeur :21
```

```
>>> type(b)
<class 'str'>
>>> c=b+b
>>> print(c)
2121
>>> b1=int(b)
>>> c=b1+b1
>>> print(c)
42
>>> b2=float(b)
>>> c=b2+b2
>>> print(c)
42.0
```

Dans les versions plus récentes du langage python (à partir de python 3.1), la fonction raw_input() n'existe plus, et la fonction input() la remplace et renvoi systématiquement une chaine de caractère. Or pour lire un entier ou un réel : il faut transtyper la valeur lue par la focntion :

```
>>> x=raw_input(' entrer une chaine')
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    x=raw_input(' entrer une chaine')
NameError: name 'raw_input' is not defined
>>> x=input(" entrer un entier : ")
 entrer un entier : 45
>>> x
'45'
>>> print(type(x))
<class 'str'>
>>> x=int(input(" entrer un entier :"))
 entrer un entier :45
>>> print(type(x))
<class 'int'>
>>> x=float(input(" entrer un entier :"))
 entrer un entier :45
>>> x
45.0
>>> print(type(x))
<class 'float'>
```

4. Variables

4.1 Définition :

Une variable est une zone de la mémoire dans laquelle on stocke une valeur; cette variable est définie par un nom. (pour l'ordinateur, il s'agit en fait d'une adresse :une zone de la mémoire).

Les noms de variables sont des noms que vous choisissez. Ce sont des suites de lettres (non accentuées) ou de chiffres. Le premier caractère est obligatoirement une lettre. (Le caractère _ est considéré comme une lettre). Python distingue les minuscules des majuscules.

4.2 Noms de variables et mots réservés :

Un nom de variable ne peut pas être un mot réservé du langage :

and	Assert	break	class	continue	def	del	elif	else	except
exec	Finally	for	from	global	if	import	in	is	lambda
not	Or	pass	print	raise	return	try	while	yield	

4.3 Type de variable

Le type d'une variable correspond à la nature de celle-ci.

Les **3** types principaux dont nous aurons besoin sont : les **entiers**, les **flottants** et les **chaines de caractères**.

Il existe de nombreux autres types (par exemple **complex** pour les nombres complexes), c'est d'ailleurs un des gros avantages de Python, ainsi que les nombres **booléens (bool)**.

4.4 Déclaration et assignation

En python, la déclaration d'une variable et son assignation (c.à.d. la première valeur que l'on va stocker dedans) se fait en même temps.

```
>>> a=10   # 10 est une valeur entière
>>> a
10
>>> print(a)
10
```

Dans cet exemple, nous avons stocké un entier dans la variable a, mais il est tout a fait possible de stocker des réels , des chaines de caractères, des complexes ou même de booléens :

```
>>> a=3.28   # 3.28 est une valeur réelle
>>> a
3.28
>>> a="bonjour"       # 'bonjour' est une chaine de caractères
>>> a
'bonjour'
>>>a=2+3j    #j est l'opérateur du complexe
>>>a
(2+3j)
>>> a=(2>3)
>>> a
False
```

4.5 La fonction type

La fonction type permet de retourner le type d'une variable

Syntaxe : type(nom_de_lavariable)

Exemples :

```
>>> a=10
>>> type(a)
<class  'int'>
>>> a=10.0
>>> type(a)
<class  'float'>
>>> a="bonjour"
>>> type(a)
<class  'str'>
>>>a=(5==5)
type(a)
<class  'bool'>
>>> a=10-12j
>>> type(a)
<class 'complex'>
```

```
>>> a=[]
>>> type(a)
<class 'list'>
>>> a={}
>>> type(a)
<class  'dict'>
>>> a=()
>>> type(a)
<class  'tuple'>
>>> a={1,2,3}
>>> type(a)
<class  'set'>
```

Remarque : pour supprimer une variable de la mémoire on utilise la fonction del.

>>> del(a)

>>>print(a)

Traceback (most recent call last):

File "<pyshell#113>", line 1, in <module>

print(a)

NameError: name 'a' is not defined

5. Les tests et les boucles :

5.1 Les tests : l'instruction if … else ….. ou if ….. elif…..else

Syntaxe1:

```
>>>if condition1:
        Action1 si condition vraie
        Action2 si condition vraie
        ……..
    Else:
        Action1 si condition fausse
        Action1 si condition fausse
        ……..
```

syntaxe2 :

```
>>>if condition1:
        Action1 si condition1 vraie
        Action2 si condition1 vraie
        ……..
    Elif condition2:  # si condition1 fausse
        Action1 si condition2 vraie
        Action1 si condition2 vraie
        ……..
    Else :
        Action1 si condition2 fausse
        Action1 si condition2 fausse
        ……..
```

Exemple :

```
>>> a = 0
>>> if a > 0 :
...     print("a est positif")
... elif a < 0 :
...     print("a est négatif")
... else:
...     print("a est nul")
```

```
>>> a = 0
>>> if a > 0 :
...     print("a est positif")
... elif a < 0 :
...     print("a est négatif")
... else:
...     print("a est nul")
...     print(" oui a est nul")
```

Les opérateurs de comparaison (opérateurs logiques) :

```
x == y          # x est égal à y
x != y          # x est différent de y
x > y           # x est plus grand que y
x < y           # x est plus petit que y
x >= y          # x est plus grand que, ou égal à y
x <= y          # x est plus petit que, ou égal à y
x is y           # x et y représentent le même objet
x is not y          # x et y ne représentent pas le même objet
```

Autres opérateurs logiques :

Condition1 **and** condition2

Condition1 **or** condition2

not condition2

Exemple :

```
>>> ok=(4>5) and (4==4)
>>> ok
False
>>> ok=(4>5)or (4==4)
>>> ok
True
```

```
>>> ok=not(4>5)
>>> ok
True
>>> a=10
>>> b=a
>>> ok=a is b
>>> ok
True
>>> b=20
>>> ok=a is b
>>> ok
False
>>> b=10
>>> ok=a is b
>>> ok
True
>>> b=30
>>> ok=a is b
>>> ok
False
>>> a=30
>>> ok=a is b
>>> ok
True
```

Exercices :

Exercice1 : Résoudre l'équation $ax^2+bx+c=0$

Exercice2 : Ecrire un programme qui lit trois variables et retourne leur max.

5.2 Les boucles :

La boucle while :

L'instruction while ("tant que", en français) permet d'exécuter une boucle tant qu'une condition est vraie.

Syntaxe : variable=VI # la variable prend une valeur initiale (VI)

```
While  Variable différente d'une valeur finale (VF) :   # tant que la condition est vraie
        Instruction1    # exécution de l'instruction 1
        Iinstruction2   # exécution de l'instruction 2
        ……..    #exécution de l'instruction n
        Incrémentation/décrémentation  # variable= ± pas
```

Rque : si VI>VF on décrémente sinon on incrémente

Exemple :

```
i = 1  # i vaut 1  la valeur initiale(VI)=1
print("valeur de i avant la boucle" , i)
while i <= 10:   #ici la valeur finale (VF)=10
```

```
    print("valeur de i dans la boucle" , i)
    i = i + 1 # on décrémente i d'un pas de 1
print("valeur de i après la boucle" , i)
```

Résultat :

('valeur de i avant la boucle', 1)

('valeur de i dans la boucle', 1)

('valeur de i dans la boucle', 2)

('valeur de i dans la boucle', 3)

('valeur de i dans la boucle', 4)

('valeur de i dans la boucle', 5)

('valeur de i dans la boucle', 6)

('valeur de i dans la boucle', 7)

('valeur de i dans la boucle', 8)

('valeur de i dans la boucle', 9)

('valeur de i dans la boucle', 10)

('valeur de i apr\xe8s la boucle', 11)

Un exemple de calcul d'intérêts composés:

```
taux = 0.03
capital = 1000.0
annee = 2014
```

```
while annee < 2020:
    annee = annee + 1
    capital = capital * (1 + taux)
    print(annee, capital)
```

La boucle For : On est souvent amené à faire des boucles pour énumérer les éléments d'une liste ou d'une séquence :

Syntaxes :

For Variable in range(VI,VF, pas) :

Instruction1 # exécution de l'instruction 1

Iinstruction2 # exécution de l'instruction 2

…… # exécution de l'instruction n

Si VI<VF : le pas est positif sinon le pas est négatif

For Variable in sequence :

Instruction1 # exécution de l'instruction 1

Iinstruction2 # exécution de l'instruction 2

…… # exécution de l'instruction n

Exemples

```
>>> for i in range(1,10):
        print(i)
1
2
3
4
5
6
7
8
9
```

```
>>> for i in range(1,10):
        print(i,end=" ")
1 2 3 4 5 6 7 8 9
```

```
>>> for i in range(10,0,-1):
        print(i,end=" ")
10 9 8 7 6 5 4 3 2 1
```

```
>>> for i in range(10,0,-1):
        print(i)
10
9
8
7
6
5
4
2
1
```

```
>>>S=[5,2,4,7,8,12]
>>> for i in S:
        print(i)
5
2
4
7
8
12
```

```
>>>S=”Bonjour”
>>> for i in S:
        print(i)
B
o
n
j
o
u
r
```

```
>>> for i in (4,5,6,7,12,20):
        print(i)
4
5
6
7
12
20
```

5.3 Les instructions break et continue

Il est possible de sortir d'une boucle avec l'instruction break. Cette instruction est très pratique pour tester une condition d'arrêt qui dépend d'une valeur entrée. Par exemple:

```
somme = 0
while True:
    n = int(input("Entrez un nombre (0 pour arrêter): "))
    if n == 0:
        break
    somme = somme + n
print("La somme des nombres est", somme)
```

Ce qui donne à l'écran :

```
Entrez un nombre (0 pour arrêter): 1
Entrez un nombre (0 pour arrêter): 7
Entrez un nombre (0 pour arrêter): 12
Entrez un nombre (0 pour arrêter): 0
La somme des nombres est 20
```

La fonction ***continue*** : retourne directement au début de la boucle, ce qui permet d'éviter des tests de condition inutiles

Exemple :

```
nb=9
while nb!=0:
    nb=input("Entrer 0 , 1, 2 ou 3? : ")
    if nb==1:
        print ("Choix un");
        continue
    if nb==2:
        print ("Choix deux");
        continue
    if nb==3:
        print ("Choix trois")
```

Exécution

```
>>>
Entrer  0, 1, 2 ou 3 ? :  1
Choix un
Entrer  0, 1, 2 ou 3? :  2
Choix deux
Entrer 0, 1, 2 ou 3? :  3
Choix tois
Entrer 0, 1, 2 ou 3? :  1
Choix  un
Entrer  0, 1, 2 ou 3? :  0
```

Exercices :

Exercice1 : Ecrire un programme en python qui lit deux variables a et b puis échange leurs valeurs.

Exercice2 : Ecrire un programme en python qui affiche les termes de la suite définie par :

U0=1, u1=1 ; $u_n=2* u_{n-1}+3*u_{n-2}$;

1. En utilisant la boucle while
2. En utilisant la boucle for
3. En utilisant la boucle do …..while

Exercice3: Ecrire un programme en python qui affiche les 20 premiers multiples de 7 :

1. En utilisant la boucle while
2. En utilisant la boucle for
3. En utilisant la boucle do …..while

6. Les fonctions en pythons

Une fonction est un sous programme qui réalise une certaine tâche.

Une fonction est composée de trois grandes parties :

- **Son nom** qui permet d'y faire appel et l'identifier des autres fonctions.
- **Ses arguments** qui permettent de spécifier des données à lui transmettre.
- **Sa sortie**, c'est-à-dire ce qu'elle retourne comme résultat.

son nom et ses arguments forment ce que l'on appelle la signature d'une fonction. Deux fonctions seront différenciées par l'interpréteur à partir du moment où elles ne possèdent pas la même signature.

- **Les fonctions** qui effectuent des instructions et retournent un résultat.
- **Les procédures** qui effectuent des instructions mais ne retournent rien du tout.

Dans la pratique, les fonctions et les procédures s'appellent exactement de la même manière ! La seule différence réside effectivement dans le fait que les procédures ne renvoient pas de résultats et ne servent donc qu'à effectuer une suite d'instructions. Ensuite, nous avons vu que pour faire appel à une fonction, il faut écrire quelque chose du genre :

```
variable = fonction(argument1, argument2, ...) # la fonction doit retourner un résultat
variable = fonction(argument1, argument2, ...) # la fonction doit retourner un résultat
```

7. Structures de base

7.1. Commentaires

Comme dans la majorité des langages de script, les commentaires Python sont définis à l'aide du caractère #. Qu'il soit utilisé comme premier caractère ou non, le # introduit un commentaire jusqu'à la fin de la ligne. Comme toujours, les commentaires sont à utiliser abondamment avec parcimonie. Il ne faut pas hésiter à commenter le code, sans pour autant mettre des commentaires qui n'apportent rien. Le listing suivant présente une ligne de commentaire en Python.

```
>>> # ceci est un commentaire
```

```
>>> print ('il s\'agit d\'un commentaire en python') #ceci est un commenataire

    il s'agit d'un commentaire en python
```

Les commentaires introduits par # devraient être réservés au remarques sur le code en sa mise en œuvre.

7.2. Les listes

7.2.1. Définition

Une liste est une structure de données de types différents

7.2.2. Création

Pour créer une liste, on utilise des crochets :

```
>>> L=[] # première façon de créer une liste vide
>>>L=list()  # une autre façon de créer une liste vide
>>>L =[1,2,5,3,2,1,5,2]   #création d'une liste des entiers
>>>L =list((1,2,5,3,2,1,5,2)) #une autre façon de créer d'une liste d'entiers
>>> L
  [1, 2, 5, 3, 2, 1, 5, 2]
>>>L=[1,5,'a','m','abc',12.25] # liste des valeurs de types différents
```

(ici : entier, caractère, chaine de caractères et réels).

Avec range : Pour créer des listes d'entiers en progression arithmétique, on peut utiliser la méthode **range** :

```
>>> L1 = range(7)  # Liste des entiers de 0 à 7
>>> L1
[0, 1, 2, 3, 4, 5, 6]
>>>L2= range(2,7)      # Liste des entiers de 2 à 6 (6=7-1)
>>>L2
[2, 3, 4, 5, 6]
>>>L3= range(2,17,2)   # Liste des entiers de 2 à 16 avec un pas de 2
>>>L3
[2, 4, 6, 8, 10, 12, 14, 16]
```

7.2.3. La taille d'une liste : la fonction len()

```
>>> len(L3)
   8
```

7.2.4. Eléments et indice :

NB : On se souviendra que le premier élément d'une liste est l'élément d'indice **0**.

Or en python le premier élément d'une liste a aussi un indice négatif qui = -len(liste)

L3	**élément**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
	indice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
		-8	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**

7.2.5. Accès aux éléments d'une liste :

```
>>>L3[0]
   2
>>>L[4]
   10
>>>L[-5]
   8
```

```
>>> L3[-len(L3)]
   2
```

On peut accéder à un élément avec sa position, et le modifier :

```
>>>L=[4,5,12.0,'a','abc']
>>>L
   [4,5,12.0,'a','abc']
>>>L[0]
   4
>>> L[0] = 7
>>> L
   [7,5,12.0,'a','abc']
```

Par contre si la liste est vide on ne peut pas lui affecter une valeur dans une position quelconque :
Exemple :
L1=[]
L1[0)=12 ; # refusé par l'interpréteur
On écrit : L1.append(12)

7.2.6. Accès au dernier élément de la liste :

On peut atteindre le dernier élément de **L**, on peut procéder ainsi :

```
>>> n = len(L)
>>> print L[n-1]
'abc'
```

En se rappelant que, puisque l'on commence à 0, le dernier élément est *n-1*.

Une autre méthode consiste à utiliser une particularité de la syntaxe python : **L[-1]** représente le dernier élément.

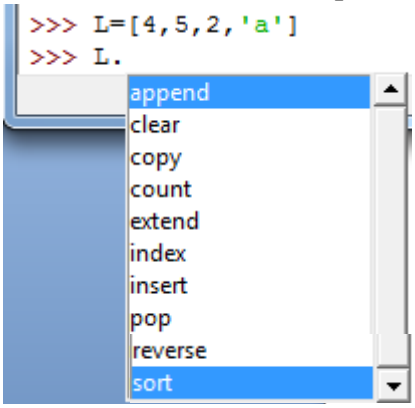
```
>>> print L[-1]
'abc'
```

Tranche d'une liste : On peut extraire facilement des éléments d'une liste :

```
>>> L = range(1,10)
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[3:6]
[4, 5, 6]
>>> L[3:]
[4, 5, 6, 7, 8, 9] # de l'indice 3 à la fin de la liste
>>> L[:5]
[1, 2, 3, 4, 5] # de l'indice 0 à l'indice 4 (5-1)
>>> L[3:6 :3]
[4] # de l'indice 3 à l'indice 5 (6-1) avec un pas de 3
```

7.2.7. Les méthodes d'une liste

Une fois définie, une liste possède un ensemble de méthodes :



append : Pour ajouter un élément à la fin d' une liste.

```
>>> L.append(12)
>>>L
```

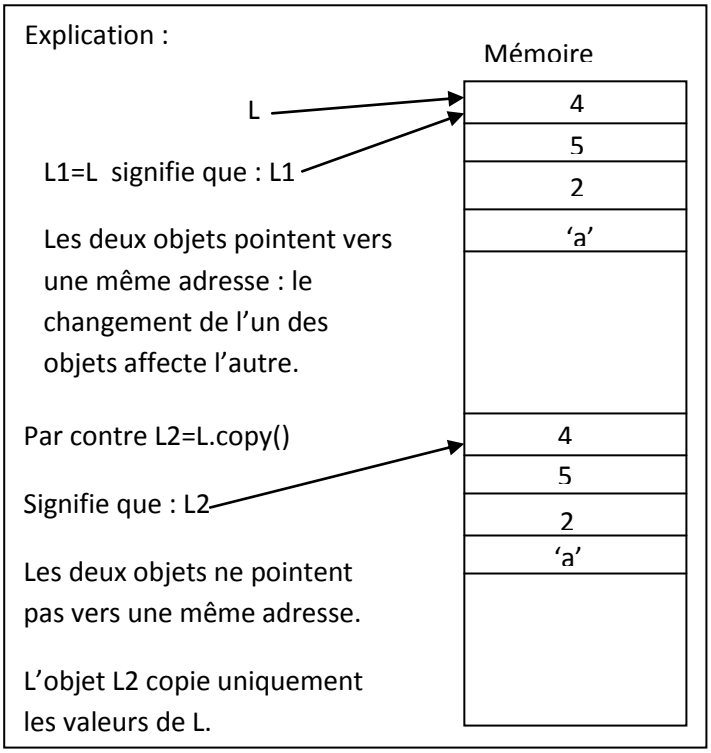
[4, 5, 2, 'a', 12]

clear : Pour supprimer tous les éléments de la liste = vider la liste.

```
>>> L.clear()
>>>L
[ ]
```

Copy : pour copier une liste dans une autre

```
>>> L=[4,5,2,'a']
>>> L1=L
>>> L1
[4, 5, 2, 'a']
>>> L
[4, 5, 2, 'a']
>>> L1[0]=33
>>> L1
[33, 5, 2, 'a']
>>> L
[33, 5, 2, 'a']
>>> L[2]=77
>>> L
[33, 5, 77, 'a']
>>> L1
[33, 5, 77, 'a']
>>>
>>> L2=L.copy()
>>> L
[33, 5, 77, 'a']
>>> L2
[33, 5, 77, 'a']
>>> L2[0]=22
>>> L
[33, 5, 77, 'a']
>>> L2
[22, 5, 77, 'a']
>>>
```



count : Pour compter le nombre d'occurrences d'un élément dans une liste.

```
>>> M = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> M.count(1)
  3
```

Extend : Soient **M** et **L** deux listes. **M.extend(L)** est équivalent à **M = M + L** (concaténation de deux listes).

```
>>> M = [1,2,3]
>>> L = [4, 5, 6]
>>> M.extend(L)
>>>   M
[1, 2, 3, 4, 5, 6]
Remarque: on peut écrire: M=M+L
```

index

L.index(x) retourne l'indice de la première occurrence de l'élément **x** dans la liste **L** :

```
>>> L = [13, 2, 3, 1, 2, 3, 1, 2, 3, 'a']
>>> L.index(13)
  0
  >>> L.index(3)
  2  # retourne l'indice de la première valeur rencontrée
```

insert

L.insert(n,x) insère l'élément **x** dans la liste **L**, en position **n** :

```
>>> L = [13, 2, 3, 18, 2, 3, 1, 2, 3, 'a']
>>> L.index(18)
  3
```

```
>>> L.insert(3,1234)
>>> L
  [13, 2, 3, 1234, 18, 2, 3, 1, 2, 3, 'a']
>>> L.index(18)
  4
```

pop

L.pop() retourne le dernier élément de la liste **L**, et le supprime de **L** :

```
>>> L
    [13, 2, 3, 1234, 18, 2, 3, 1, 2, 3, 'a']
>>> L.pop()
    'a'
>>> L
   [13, 2, 3, 1234, 18, 2, 3, 1, 2, 3]
```

Cela peut aussi s'appliquer à un autre élément, dont l'indice est passé en argument :

```
>>> L
[13, 2, 3, 1234, 18, 2, 3, 1, 2, 3]
>>> L.pop(3)# supprime l'element d'indice 3
1234
>>> L
[13, 2, 3, 18, 2, 3, 1, 2, 3]
```

Remove : L.remove(x) supprime la première occurence de l'élément **x** dans la liste **L** :

```
>>> L
   [13, 2, 3, 18, 2, 3, 1, 2, 3]
>>> L.remove(3) # supprimer la valeur 3 de la liste
>>> L
   [13, 2, 18, 2, 3, 1, 2, 3]
```

Reverse Renverse la liste.

```
>>> L
   [13, 2, 18, 2, 3, 1, 2, 3]
>>> L.reverse()
>>> L
    [3, 2, 1, 3, 2, 18, 2, 13]
```

Sort : Trie la liste.

```
>>> L
  [3, 2, 1, 3, 2, 18, 2, 13]
>>> L.sort()
>>> L
[1, 2, 2, 2, 3, 3, 13, 18]
```

On peut trier par ordre décroissant :

```
>>> L.sort(reverse = True)
>>> L
[18,13, 3, 3, 2, 2, 2, 1]
```

Autres méthodes de création des listes : Les listes de compréhension :

Exemple :

```
>>> L=[ I for i in range(10) ]
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L1=[ 2*x**2-3*x+2 for x in range(10)]
```

```
>>> L1
[2, 1, 4, 11, 22, 37, 56, 79, 106, 137]
>>> import numpy as np
>>> L1=[ 2*x**2-3*x+2 for    x   in   np.arange( 0,  2,  0.1, float)]
>>> L1
[2.0, 1.72, 1.48, 1.2799999999999998, 1.1199999999999999, 1.0, 0.91999999999999993,
0.88000000000000012, 0.87999999999999989, 0.91999999999999993, 1.0, 1.1200000000000001,
1.2800000000000002, 1.48, 1.7200000000000006, 2.0, 2.3200000000000003, 2.6800000000000006,
3.0800000000000001, 3.5200000000000005]
```

Les listes à plusieurs dimensions :

```
>>> L=[[1,2,3,4,5],[11,22,33,44,55],[10,20,30,40,50]]
>>> L
[[1, 2, 3, 4, 5], [11, 22, 33, 44, 55], [10, 20, 30, 40, 50]]
>>> L[0][0]
1
>>> L[0][3]
4
>>> L[2][3]
40
>>> L[2][:]
[10, 20, 30, 40, 50]
>>> L[:][1]
[11, 22, 33, 44, 55]
>>> L[:][0]
[1, 2, 3, 4, 5]
>>> L[0][:]
[1, 2, 3, 4, 5]
>>> L[:][2]
[10, 20, 30, 40, 50]
>>> L[1][:]
[11, 22, 33, 44, 55]
```

Les liste de compréhension à plusieurs dimensions :

```
>>> L1=[ [i+j for i in range(10)]  for j in range(10)]
>>> L1
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [2, 3, 4, 5, 6, 7, 8, 9, 10
, 11], [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], [4, 5, 6, 7, 8, 9, 10, 11, 12, 13], [5, 6, 7, 8, 9,
10, 11, 12, 13, 14], [6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 8, 9, 10, 11, 12, 13, 14, 15,
16], [8, 9, 10, 11, 12, 13, 14, 15, 16, 17], [9, 10, 11, 12, 13, 14, 15, 16, 17, 18]]
>>> L1[:][2]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> L1[2][:]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>>
```

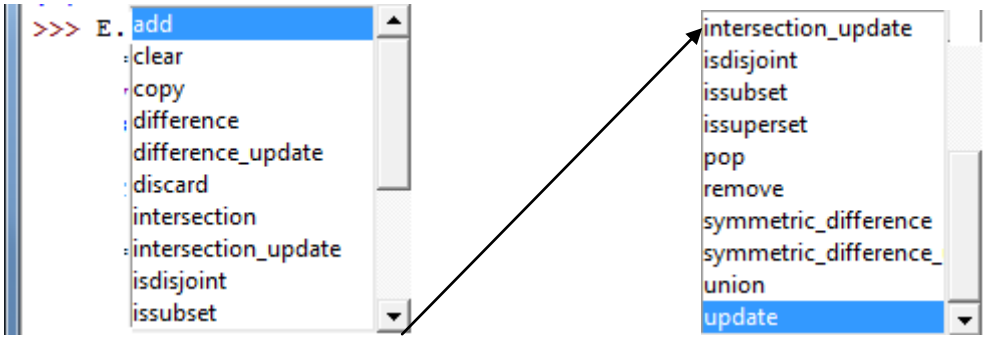
7.3. Les ensembles ou set

Un ensemble en python est semblable à une liste mais ne peut pas contenir des doublons c.à.d ne contient que des éléments distincts.

Déclaration :

```
>>> E={4,5,4,7,8,12}
>>> type(E)
<class 'set'>
>>> E
{8, 12, 4, 5, 7}
>>> # de meme
>>> E=set((4,5,4,7,8,12))
>>> E
{8, 4, 5, 12, 7}
>>>
```

Les méthodes d'un ensemble :



Add : Ajouter un élément à l'ensemble :

```
>>> E.add(25)
>>> E
 {8,  4,  5,  7,  12,  25}
```

Copy : Copier un ensemble.

```
>>> E1=E.copy()
>>>  E
{8,  4,  5,  7,  12,  25}
>>>E1
  {8,  4,  5,  7,  12,  25}
```

Clear : efface le contenu d'une liste

```
>>> E.clear()
>>> E
  set()
```

Opérations ensemblistes

Les diverses opérations ensemblistes, définies dans le cours de mathématiques, sont réalisables avec des **set**.

Supposons, par exemple, que l'on ait défini les deux ensembles suivants :

```
>>> E1 = set((1, 2, 3))
>>> E2 = set((0, 1))
```

LA RÉUNION

Pour réaliser l'union ***E1 ∪ E2*** de deux ensembles, on peut utiliser au choix la méthode **union**, ou l'opérateur | :

```
>>> E1.union(E2)
set([0, 1, 2, 3])
>>> E1 | E2
set([0, 1, 2, 3])
```

L'INTERSECTION

Pour réaliser l'intersection **E1 ∩ E2** de deux ensembles, on peut utiliser au choix la méthode **intersection**, ou l'opérateur **&** :

```
>>> E1.intersection(E2)
set([1])
>>> E1 & E2
set([1])
```

LA DIFFÉRENCE

Pour réaliser la différence S1\S2 de deux ensembles, on peut utiliser au choix la méthode**difference**, ou l'opérateur **-** :

```
>>> E1.difference(E2)
set([2, 3])
>>> E1-E2
set([2, 3])
```

LA DIFFÉRENCE SYMÉTRIQUE

Pour réaliser la différence symétrique E1 Δ E2 de deux ensembles, on peut utiliser au choix la méthode **symetric_difference**, ou l'opérateur ^ :

```
>>> E1.symmetric_difference(E2)
set([0, 2, 3])
>>> E1 ^ E2
set([0, 2, 3])
```

Inclusion

Deux méthodes sont fournies pour tester si A est inclus dans B, ou si A est un sur-ensemble de B :

```
>>> A = set([1, 2])
>>> B = set([0, 1, 2, 3])
>>> A.issubset(B)
True
>>> B.issuperset(A)
True
```

7.4. Les tuples

Python propose un type de données appelé *tuple*, qui est assez semblable à une liste mais qui n'est pas modifiable. Du point de vue de la syntaxe, un tuple est une collection d'éléments séparés par des virgules : **>>> T=(4,12,8,25)**

>>> T

(4, 12, 8, 25)

>>> type(T)

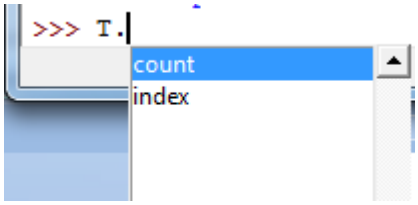
<class 'tuple'>

>>> T=tuple((4,12,8,25))

>>> type(T)

<class 'tuple'>

Les méthodes d'un tuple :



Le tuple ne contient que deux méthodes count et index.

Accès aux éléments d'une liste

```
>>> T[0]
4
>>> T[:4]
(4, 12, 8, 25)
>>> T[1:3]
(12, 8)
>>> T[2:]
(8, 25)
>>> T[2:4:3]
(8,)
```

Modifier les éléments d'un tuple :

```
>>> T[0]=20
Traceback (most recent call last):
  File "<pyshell#178>", line 1, in <module>
    T[0]=20
TypeError: 'tuple' object does not support item assignment
```

7.5. Les dictionnaires

7.6. Les chaines de caractères :

6.3.1 Définition

Une chaine de caractères contenant un ensemble de caractères délimités par :

Des guillemets simples : chaine1='Bonjour' #chaine sur une seule ligne

Des guillemets simples : chaine1=''Bonjour'' #chaine sur une seule ligne

Des guillemets simples : chaine1='''''Bonjour

Tout le monde '' '' '' #chaine sur plusieurs lignes

6.3.2 Accès aux caractères d'une chaine :

```
>>> S1='Bonjour'
>>> S1[0]
'B'
>>> S1[2]
'n'
>>> S1[1:]
'onjour'
>>> S1[:3]
'Bon'
>>> S1[2:5]
'njo'
>>> S1[::2]
'Bnor'
```

```
>>> S2="Bonjour"
>>> S2[0]
'B'
>>> S2[2]
'n'
>>> S2[3:5]
'jo'
>>> S2[::2]
'Bnor'
```

```
>>> S3="""Bonjour
tout le monde"""
>>> S3[0]
'B'
>>> S3[4]
'o'
>>> S3[2:]
'njour\ntout le monde'
>>> S3[:7]
'Bonjour'
>>> S3[3:9:2]
'ju\n'
>>> S3[2:11:3]
'nut'
```

Caractère d'échappement

Le symbole \ est spécial : il permet de transformer le caractère suivant :

- \n est un saut de ligne
- \t est une tabulation
- \b est un « backspace »
- \a est un « bip »
- \' est un « ' », mais il ne ferme pas la chaine de caractères
- \" est un « " », mais il ne ferme pas la chaine de caractères
- \\ est un « \ »

Si on veut que le symbole \ reste simplement un \ dans une chaîne, on peut utiliser une chaîne « brute » ("raw string") en préfixant le premier guillemets avec un r :

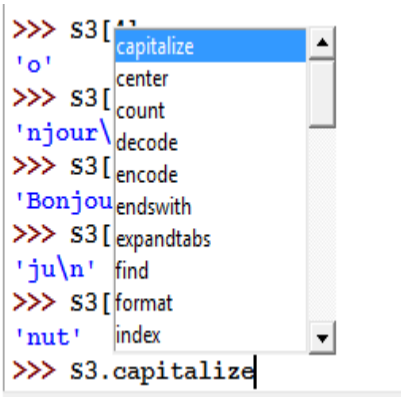
```
s = r"Ceci est une chaîne de caractères\nsur une seule ligne"
```

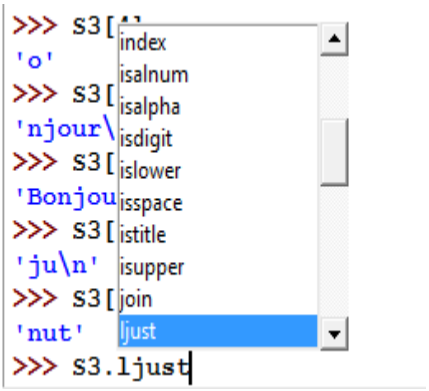
6.3.3 les méthodes d'une chaine de caractères

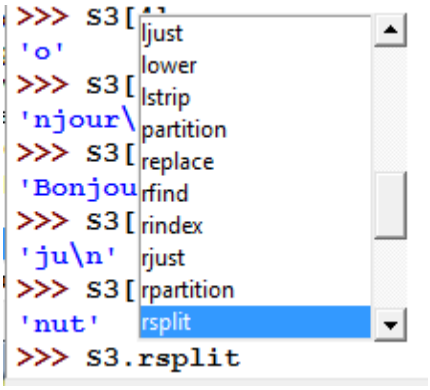
Une méthode est une fonction qu'on peut appliquer sur chaine :

Pour afficher les méthodes d'une chaine, il suffit d'écrire la chaine suivie d'un point.

L'exemple suivant montre les méthodes de la chaine S3 définie ci haut.







Méthode	
Split() : découpe une chaine en une liste de mots	S= ''Bonjour tout le monde '' LS=S.split() donne LS=['Bonjour','tout','l' , 'monde']

Join(liste de chaines) : concatène une liste de chaine en chaine unique	>>> S1="-"
	 >>> S="Bonjour Tout Le Monde"
	 >>> S3=S1.join(S)
	 >>> print(S3) B-o-n-j-o-u-r- -T-o-u-t- -L-e- -M-o-n-d-e
	 >>> S4=S1.join(['Bonjour','tout','le','monde'])
	 >>> print(S4) Bonjour-tout-le-monde
	 >>>S5= «'' ''.join(['Bonjour','tout','le','monde'])
	 >>>print(S5) Bonjour tout le monde

Find(sous chaine) : donne la position d'une sous chaine dans une chaine	>>> f=S.find('tout')
	 >>> f donne -1 # la sous chaine n'existe pas
	 >>> f=S.find('Tout') >>> print(f) donne 8

Count(sous chaine) : donne le nombre le nombre d'apparition d'une sous chaine	>>> f=S.count('tout') >>> f donne 0 # la sous chaine n'existe pas

dans une chaine	>>> f=S.count('Tout') >>> print(f) donne 1

Lower() : convertit une chaine en miniscule	>>> print(S) donne Bonjour Tout Le Monde
	>>> Sm=S.lower()
	>>> print(Sm) bonjour tout le monde

upper() : convertit une chaine en mijuscule	>>> Sm=S.upper() >>> print(Sm) bonjour tout le monde
capitalize() : convertit le 1ere lettre d'une chaine en mijuscule.	>>> SM=S.upper() >>> print(SM) donne BONJOUR TOUT LE MONDE
title() : convertit tous les 1er lettres des mots d'une chaine en mijuscule.	>>> t=S.title() >>> print(t) donne Bonjour Tout Le Monde
Swapcase() : intervertit les lettres majuscules et miniscules	>>> tt=t.swapcase() >>> print(tt) donne bONJOUR tOUT lE mONDE
Strip() : supprime les espaces blancs en début en en fin de la chaine	>>>ss='' bonjour tout le monde'' >>>sss=ss.strip() donne ''bonjour tout le monde''
Replace() : remplace une sous chaine par une autre	>>> s1=S.replace('Tout Le Monde','la compagnie') >>> print(s1) donne Bonjour la compagnie

Index(sous chaine) : retourne la position de la sous chaine dans une chaine	>>> S="Bonjour"
	>>> S.index('j')
	3
	>>> S.index('njour')
	2

Center(nombre) : centrer la chaine sur nombre caractères	>>> print(S.center(40)) Bonjour 40 caractères

Format() : remplace un format dans une chaine	>>> s1="Voici {0}chaîne à {1} trous."
	>>>print(s1)
	Voici {0}chaîne à {1} trous.
	>>> print(s1.format("une " , 2))
	Voici une chaîne à 2 trous.
	>>> print("a{0}cada{0}".format("bra"))
	abracadabra

Les fonctions **isupper()** et **islower()** : Testent si une sous chaine est majuscule ou minuscule	>>> print(s1.isupper())
	False
	>>> print(s1.islower())
	False
	>>> print("BONJOUR".isupper())
	True

8. Les Tableaux et les matrices

Nous allons voir comment créer des tableaux avec la fonction **array()** de NumPy. Ces tableaux pourront être utilisés comme des vecteurs ou des matrices grâce à des fonctions de NumPy. Les fonctions (**dot()**, **det()**,**inv()**, **eig()**, ….etc.) permettent de réaliser des calculs matriciels utilisés en algèbre linéaire.

```
>>> import numpy as np
```

7.1 Les Tableaux à une ou plusieurs dimension : np.array()

Pour créer des tableaux, nous allons utiliser la fonction array() de la bibliothèque **numpy**.

7.1.1 Tableaux monodimensionnels (1D)

Pour créer un tableau 1D, il suffit de passer une liste de nombres en argument de **array()**. Un liste est constituée de nombres séparés par des virgules et entourés de crochets ([ ]).

```
>>> T = array([4,7,9])
>>> T
array([4, 7, 9])
>>> print(T)
[4 7 9]
```

Pour connaître le type du résultat de **array()**, on peut utiliser la fonction **type()**.

```
>>> type(a)
<type 'numpy.ndarray'>
```

On constate que ce type est issu du package numpy. Ce type est différent de celui d'une liste.

```
>>> type([4, 7, 9])
<type 'list'>
```

Tableaux bidimensionnels (2D) :

Pour créer un tableau 2D, il faut transmettre à array() une liste de listes grâce à des crochets imbriqués.

```
>>> T = array([[1, 2, 3], [4, 5, 6]])
>>> T
array([[1, 2, 3],
       [4, 5, 6]])
```

La fonction size() : La fonction size() renvoie le nombre d'éléments du tableau.

```
>>> T1 = array([2,5,6,8])
>>> size(T1)
4
>>> T2 = array([[1, 2, 3],
                [4, 5, 6]])
>>> size(T2)
6
```

La fonction shape()

La fonction **shape()** (*forme*, en anglais) renvoie la taille du tableau.

```
>>> T1 = array([2,5,6,8])
>>> shape(T1)
(4,)
>>> T2 = array([[1, 2, 3],
                [4, 5, 6]])
>>> shape(T2)
(2, 3)
```

On distingue bien ici que T1 et T2 correspondent à des tableaux 1D et 2D, respectivement.

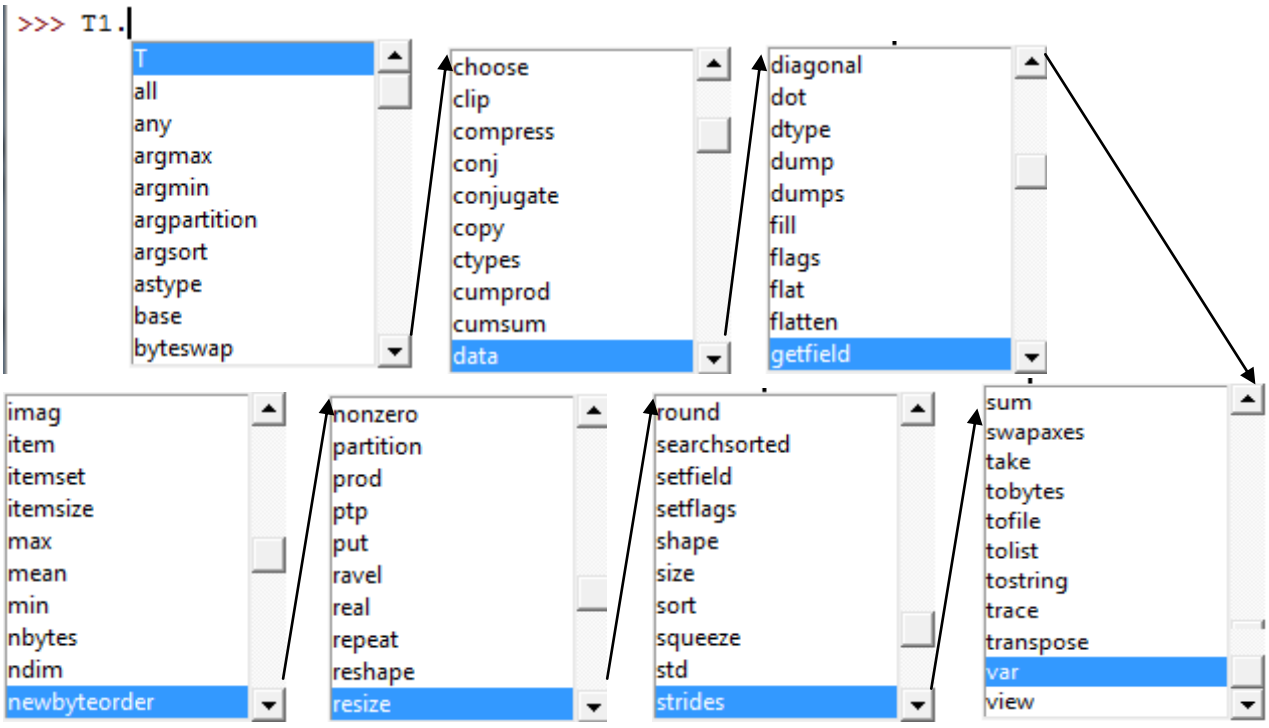
Produit terme à terme

Il est possible de réaliser un produit terme à terme grâce à l'opérateur *. Il faut dans ce cas que les deux tableaux aient la même taille.

```
>>> T1 = array([[1, 2, 3],
                [4, 5, 6]])
>>> T2 = array([[2, 1, 3],
                [3, 2, 1]])
>>> T1*T2
array([[ 2,  2,  9],
       [12, 10,  6]])
```

Comme tout objet structure en python un tableau quelque soit en une seule ou plusieurs dimensions

possède un ensemble de méthodes :



Transposition d'une matrice :

La fonction (ou la méthode) `transpose`, ou plus simplement la méthode `T`, renvoient la transposée d'une matrice :

```
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
>>> a.transpose()
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

```
>>> a.T
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

Supprimer des lignes et des colonnes

delete(a,k,axis=0) suppriment la k-ième ligne de la matrice a (axis=0)

delete(a,k,**axis**=1) suppriment la k-ième colonne de a (axis=1)

```
>>> a
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

une a matrice de type 4×5

```
>>> b = np.delete(a,2,0);
>>> b
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [30, 31, 32, 33, 34]])
```

la matrice b est obtenue en supprimant la ligne d'indice 2 de a

```
>>> c = np.delete(b,3,1)
>>> c
array([[ 0,  1,  2,  4],
       [10, 11, 12, 14],
       [30, 31, 32, 34]])
```

la matrice c est obtenue en supprimant la colonne d'indice 3 de b

On peut également supprimer plusieurs colonnes (ou lignes) à la fois. Voici quelques exemples :

```
>>> np.delete(a,np.s_[0::2],1)
array([[ 1,  3],
       [11, 13],
       [21, 23],
       [31, 33]])
```

supprime les colonnes d'indice pair

```
>>> np.delete(a,np.s_[1::2],1)
array([[ 0,  2,  4],
       [10, 12, 14],
       [20, 22, 24],
       [30, 32, 34]])
```

supprime les colonnes d'indice impair

```
>>> np.delete(a,[0,2,3],1)
array([[ 1,  4],
       [11, 14],
       [21, 24],
       [31, 34]])
```

supprime les colonnes d'indice 0, 2, 3

Insérer des lignes et des colonnes :

Les expressions `insert(a,k,v,axis=0)` et `insert(a,k,v,axis=1)` permettent d'insérer la valeur v respectivement avant la k-ième ligne ou avant la k-ième colonne de a. Voici deux exemples :

```
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

une matrice a de type 3×4

```
>>> np.insert(a,2,-1,axis=1)
array([[ 0,  1, -1,  2,  3],
       [10, 11, -1, 12, 13],
       [20, 21, -1, 22, 23]])
```

insère des -1 juste avant la colonne d'indice 2

```
>>> np.insert(a,1,0,axis=0)
array([[ 0,  1,  2,  3],
       [ 0,  0,  0,  0],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

insère des 0 juste avant la ligne d'indice 1

Permutations/rotations de lignes, de colonnes :

`fliplr(m)` inverse l'ordre des colonnes de m : ici « `lr` » est mis pour « `left right` ».

Remarque : ça ne marche pas pour les vecteurs, car il faut qu'il y ait au moins deux dimensions.

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

la matrice initiale m

```
>>> np.fliplr(m)
array([[ 3,  2,  1,  0],
       [13, 12, 11, 10],
       [23, 22, 21, 20]])
```

inverse l'ordre des colonnes de m

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

le contenu initial de m est inchangé

`flipud(m)` inverse l'ordre des lignes de m : ici « `ud` » est mis pour « `up down` ».

```
>>> m
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

la matrice initiale m

```
>>> np.flipud(m)
array([[30, 31, 32, 33, 34],
       [20, 21, 22, 23, 24],
       [10, 11, 12, 13, 14],
       [ 0,  1,  2,  3,  4]])
```

inverse l'ordre des lignes de m

```
>>> m
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

le contenu initial de m est inchangé

Pour ceux que ça intéresse, **rot90(m,k=1)** renvoie une copie de la matrice m après k rotations d'angle π/2.

```
>>> m
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]]
```

```
>>> np.rot90(m)
array([[ 3, 13, 23],
       [ 2, 12, 22],
       [ 1, 11, 21],
       [ 0, 10, 20]])
```

rotation de 90°

```
>>> np.rot90(m,-1)
array([[20, 10,  0],
       [21, 11,  1],
       [22, 12,  2],
       [23, 13,  3]])
```

rotation de -90°

```
>>> np.rot90(m,2)
array([[23, 22, 21, 20],
       [13, 12, 11, 10],
       [ 3,  2,  1,  0]])
```

rotation de 180°

L'expression **swapaxes(a,axe,axe')** effectue un échange des axes sur le tableau a. Pour les vecteurs, c'est sans effet, pour les matrices c'est équivalent à la transposition. Ça ne peut donc avoir d'utilité pour les tableaux à n ≥ 3 indices.

```
>>> a = np.arange(30).reshape(2,3,5); a
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],

       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
```

deux tableaux de format 3 × 5

```
>>> np.swapaxes(a,0,1)
array([[[ 0,  1,  2,  3,  4],
        [15, 16, 17, 18, 19]],

       [[ 5,  6,  7,  8,  9],
        [20, 21, 22, 23, 24]],

       [[10, 11, 12, 13, 14],
        [25, 26, 27, 28, 29]]])
```

trois tableaux de format 2 × 5

Produit matriciel - dot()

Un tableau peut jouer le rôle d'une matrice si on lui applique une opération de calcul matriciel. Par exemple, la fonction **dot()** permet de réaliser le produit matriciel.

```
>>> T1 = array([[1, 2, 3],
                [4, 5, 6]])
>>> T2 = array([[4],
                [2],
                [1]])
>>> dot(T1,T2)
array([[11],
       [32]])
```

Le produit d'une matrice de taille n x m par une matrice m x p donne une matrice n x p.

Transposé :

```
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Complexe conjugué - conj()

```
>>> u = array([[  2j, 4+3j],
               [2+5j, 5   ],
               [   3, 6+2j]])
>>> conj(u)
array([[ 0.-2.j,  4.-3.j],
       [ 2.-5.j,  5.+0.j],
       [ 3.+0.j,  6.-2.j]])
```

Transposé complexe conjugué :

```
>>> conj(u).T
array([[ 0.-2.j,  2.-5.j,  3.+0.j],
       [ 4.-3.j,  5.+0.j,  6.-2.j]])
```

Tableaux et slicing

Lors de la manipulation des tableaux, on a souvent besoin de récupérer une partie d'un tableau. Pour cela, Python permet d'extraire des *tranches* d'un tableau grâce une technique appelée **slicing**(tranchage, en français). Elle consiste à indiquer entre crochets des indices pour définir le début et la fin de la *tranche* et à les séparer par deux-points :.

```
>>> a = array([12, 25, 34, 56, 87])
>>> a[1:3]
array([25, 34])
```

Dans la tranche [n:m], l'élément d'indice n est inclus, mais pas celui d'indice m. Un moyen pour mémoriser ce mécanisme consiste à considérer que les limites de la tranche sont définies par les numéros des positions situées entre les éléments, comme dans le schéma ci-dessous :

Il est aussi possible de ne pas mettre de début ou de fin.

```
>>> a[1:]
array([25, 34, 56, 87])
>>> a[:3]
array([12, 25, 34])
>>> a[:]
array([12, 25, 34, 56, 87])
```

Slicing des tableaux 2D

```
>>> a = array([[1, 2, 3],
               [4, 5, 6]])
>>> a[0,1]
2
>>> a[:,1:3]
array([[2, 3],
       [5, 6]])
>>> a[:,1]
array([2, 5])
>>> a[0,:]
array([1, 2, 3])
```

Warning

a[:,n] donne un tableau 1D correspondant à la colonne d'indice n de a. Si on veut obtenir un tableau 2D correspondant à la colonne d'indice n, il faut faire du slicing en utilisant a[:,n:n+1].

```
>>> a[:,1:2]
array([[2],
```

```
       [5]])
```

Tableaux de 0 - zeros()

zeros(n) renvoie un tableau 1D de n zéros.

```
>>> zeros(3)
array([ 0.,  0.,  0.])
```

zeros((m,n)) renvoie tableau 2D de taille m x n, c'est-à-dire de *shape* (m,n).

```
>>> zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Tableaux de 1 - ones()

```
>>> ones(3)
array([ 1.,  1.,  1.])
>>> ones((2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Matrice identité - eye()

eye(n) renvoie tableau 2D carré de taille n x n, avec des *uns* sur la diagonale et des *zéros* partout ailleurs.

```
>>> eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Algèbre linéaire

Déterminant - det()

```
 >>> a = array([[1, 2],
               [3, 4]])
 >>> np.linalg.det(a)
 -2.0
```

Inverse - inv()

```
>>> a = array([[1, 3, 3],
               [1, 4, 3],
               [1, 3, 4]])
>>> np.linalg.inv(a)
array([[ 7., -3., -3.],
       [-1.,  1.,  0.],
       [-1.,  0.,  1.]])
```

Valeurs propres et vecteurs propres - eig()

```
>>> A = array([[ 1, 1, -2 ], [-1, 2, 1], [0, 1, -1]])
>>> A
array([[ 1,  1, -2],
       [-1,  2,  1],
       [ 0,  1, -1]])
>>> D, V = np.linalg.eig(A)
>>> D
array([ 2.,  1., -1.])
>>> V
array([[  3.01511345e-01,  -8.01783726e-01,   7.07106781e-01],
       [  9.04534034e-01,  -5.34522484e-01,  -3.52543159e-16],
       [  3.01511345e-01,  -2.67261242e-01,   7.07106781e-01]])
```

Les colonnes de V sont les vecteurs propres de A associés aux valeurs propres qui apparaissent dans D.

Exercice : Vérifier que les colonnes de V sont bien des vecteurs propres de A

Changement de la taille d'un tableau

Il est possible de changer la taille d'un tableau en utilisant l'attribut **shape** de ce tableau.

```
>>> u = arange(1,16)
>>> u
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> shape(u)
(15,)
>>> u.shape = (3,5)
>>> u
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
>>> shape(u)
(3, 5)
```

Obtention d'un tableau 2D ligne ou colonne

```
>>> a = arange(1,6)
>>> a
array([1, 2, 3, 4, 5])
>>> a.shape = (1,size(a))
>>> a
array([[1, 2, 3, 4, 5]])
>>> a.shape = (size(a),1)
>>> a
array([[1],       [2],       [3],       [4],       [5]])
```