Sommaire

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).
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 ayant l'extension ***.py***. 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.

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 et logiques:

=	**: affectation**	**==**	**: égalité**
+	: addition numérique ou concaténation des chaines de caractères		
-	: soustraction	< / <=	: inférieur / inférieur ou égale
*****	: multiplication	> / >=	: supérieur / supérieur ou égale
/	: division	!=	: différent
//	: division entière	and	: ET logique
%	: reste de la division	or	: OU logique
******	: puissance	not	: négtion

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'
>>> reponse=(2>8)
>>> reponse
False
>>> reponse=(2<8) or (3!=3)
>>> reponse
True
>>> r=30%8
>>> r
6
```

3. Les entrées Sorties (fonctions prédéfinies)

3.1 La fonction print :

Pour afficher une variable et/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, alphabétique ou alphanumérique) pour lire un entier ou un réel, on doit transtyper la valeur lue au clavier par la fonction input().
Remarque : pour convertir, en entier ou en réel, une variable lue avec **input ,** on utilise les fonctions **int()** ou **float().**

Exemple :

```
>>> b=input(" Entrer une valeur :")
 Entrer une valeur :21
>>> type(b)
>>> print(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
```

4. Les variables

4.1 Définition :

Une variable est une zone 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) et/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. Les mots réservés au langage Python sont :

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 variables :

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
```

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")
```

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)or (4==4)

>>> ok

True

>>> ok=not(4>5)

True

>>> b=30

>>> ok=a is b

>>> ok

False

>>> a=30

>>> ok=a is b

>>> ok

True

>>> ok=(4>5) and (4==4)

>>> ok

False

>>> ok=not(4>5)

>>> ok ok vaut True

>>> a=10

>>> b=a

>>> ok=a is b

>>> ok ok vaut True

>>> b=20

>>> ok=a is b

>>> ok ok vaut False

>>> b=10

>>> ok=a is b

>>> ok ok vaut True

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 (recherche de condition)

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 incrémente 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 après la boucle : 11

Un exemple de calcul d'intérêts composés:

```
taux = 0.03
capital = 1000.0
annee = 2016
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)
10
9
8
7
6
5
4
2
1
```

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

6. Les fonctions en python

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
>>> fonction(argument1, argument2, ...) # la fonction ne retourne rien

Exemple :

définition	appel	exécution

`>>> def mafocntion1():`
`  print(" j'utilise ma premiere fonction ")`
`  print(" et je suis tres content ")`
	`>>> mafocntion1()`	`j'utilise ma premiere fonction`
		`et je suis tres content`

`>>> def jesuis(ana,omri):`
`  print(" je m'appelle ",ana)`
`  print(" et j'ai ", omri, " ans ")`
	`>>> jesuis("ahmed",43)`	`je m'appelle ahmed`
		`et j'ai 43 ans`

`>>> def puissance(nombre, puissance) :`
`  print("Je calcule %f à la puissance %f" % (nombre, puissance))`
`  return nombre**puissance`
	`>>> p=puissance(2.5,2)`
	`>>> x=5`
	`>>> n=7`
	`>>> print(puissance(x,n))`
		`Je calcule 2.500000 à la puissance 2.000000`
		`>>> print(p)`
		`6.25`
		
		`Je calcule 5.000000 à la puissance 7.000000`
		`78125`

```
>>> def operations(nombre, facteur) :
        print("Je suis fort en math !")
        return nombre+facteur, nombre-facteur, nombre/facteur, nombre*facteur
```

```
>>> print(operations(3,12))
```

```
Je suis fort en math !
(15, -9, 0.25, 36)
```

- Les fonctions avec des arguments ayant des valeurs par défaut

Reprenons la fonction puissance :

```
>>> def puissance(nombre, puissance=2):
        return nombre**puissance

>>> #
>>> print(puissance(5,3))
125
>>> print(puissance(5))
25
>>> print(puissance(5,2))
25
```

Si la valeur du 2ème paramètre est manquée la fonction prend la valeur par défaut qui est 2

- Les fonctions avec un nombre d'arguments indéfini :

Il existe des fonctions qui acceptent un nombre indéterminé d'arguments, c'est le cas de la fonction *print* par exemple. Comment cela fonctionne-t-il ? Et bien, c'est assez simple. Pour cela, il est possible de le faire de deux manières :

- *avec des arguments anonymes*
- *avec arguments associés à des clés.*

Nous allons donc voir les deux manières de créer des fonctions avec un nombre indéterminé de paramètres.

- Arguments anonymes :

La syntaxe est particulièrement simple. En fait, on va préciser une séquence (liste, tuple ou chaîne de caractères) comme argument. Le fait que cet argument doive être une séquence se précise par l'utilisation de l'opérateur '*' juste avant le nom de l'argument :

```
>>> def fonction(*arguments) :
        """Test de fonction avec un nombre indéfini d'arguments.
        arguments : Une séquence à écrire sur l'ecran."""
        for element in arguments :
                print(element)
```

1er appel

```
>>> fonction("ahmed", 20,18,"MP")
ahmed
20
18
MP
```

2ème Appel

```
>>> fonction("ahmed","Asmae", 16,18,"MP1","MP2")
ahmed
Asmae
16
18
MP1
MP2
```

3ème Appel :

```
>>> fonction(*"ahmed MP 1")
a
h
m
e
d

M
P

1
```

> ***Remarquez*** *bien la présence de l'étoile **

> *Si on oublie cette étoile la fonction considère la séquence ''ahmed MP 1 '' comme un seul paramètre.*

- Arguments avec clé :

Si on veut associer une clé à chacun des arguments qu'on donne en supplément à une fonction, il va falloir utiliser un dictionnaire (***voir paragraphe 7.5 page 20***). Et ceci, c'est l'opérateur '**' qui sera utilisé.

```
>>> def fonction(**arguments) :
        """Test de fonction avec un nombre indéfini d'arguments.
        arguments : Un dictionnaire à écrire en console."""
        for cle in arguments :
                print(cle,arguments[cle])
```

1er Appel

```
>>> fonction(arg1="Ahmed",arg2=20,arg3=18,arg4="MP")
arg2 20
arg1 Ahmed
arg4 MP
arg3 18
>>>
```

2ème Appel

```
>>> fonction(arg1="Ahmed",arg2="Asmae",arg3=16,arg4=18,arg5="MP1",arg6="MP2")
arg6 MP2
arg1 Ahmed
arg3 16
arg5 MP1
arg2 Asmae
arg4 18
>>>
```

Une autre façon d'appeler la fonction : remarquez la présence du double étoiles **** et le mot **dict**.

```
>>> fonction(**dict(arg1="Ahmed",arg2="Asmae",arg3=16,arg4=18,arg5="MP1",arg6="MP2"))
arg6 MP2
arg1 Ahmed
arg3 16
arg5 MP1
arg2 Asmae
arg4 18
>>>
```

Documenter une fonction : Une dernière chose très importante est la documentation de la fonction. On va commencer à documenter notre code pour le rendre plus compréhensible et exploitable par d'autres personnes.

Pour cela, c'est très simple, il suffit de placer immédiatement sous la ligne de définition de notre fonction, une chaîne de caractères entre triple 'quote' (""" ou ''') qui sera considérée un commentaire.

```
>>> def fonction(arg1, arg2, arg3) :
        """
    Je documente ma fonction. J'explique rapidement à quoi sert cette fonction.
    arg1 : J'écris à quoi correspond cet argument.
    arg2 : J'écris à quoi correspond cet argument.
    arg3 : J'écris à quoi correspond cet argument.
    return : J'indique ce que retourne la fonction
    """
        pass
```

Appel :

```
>>> fonction(1,2,3)
>>>
```

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 commentaire

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, ses éléments sont indexés et mutables

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 =[i for i in range(7)]  # Liste des entiers de 0 à 6
>>> L1
[0, 1, 2, 3, 4, 5, 6]
>>>L2=[k for k in range(2,7)]     # Liste des entiers de 2 à 6 (6=7-1)
>>>L2
[2, 3, 4, 5, 6]
>>>L3=[k for k in 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
>>>L3[4]
    10
>>>L3[-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'
```

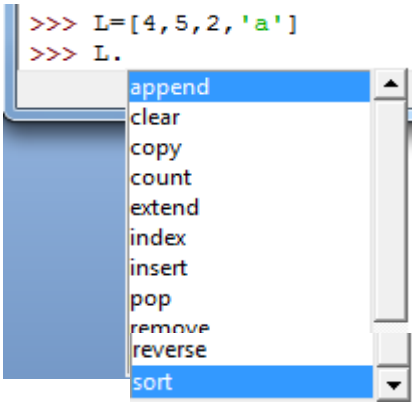
Tranche d'une liste : On peut extraire facilement des éléments d'une liste :

l'index de début (qui est inclus) étant séparé de l'index de fin (qui est exclu) par le caractère **« : »**
Si l'index de début est manquant, python considère que le début est 0, et si l'index de fin est manquant il prend la longueur de la liste.

```
  >>> L = [k for k in range(1,10)]
  >>> L
  [1, 2, 3, 4, 5, 6, 7, 8, 9]
  >>> L[3:6] # de l'indice 3 à l'indice 5 (6-1)
  [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.

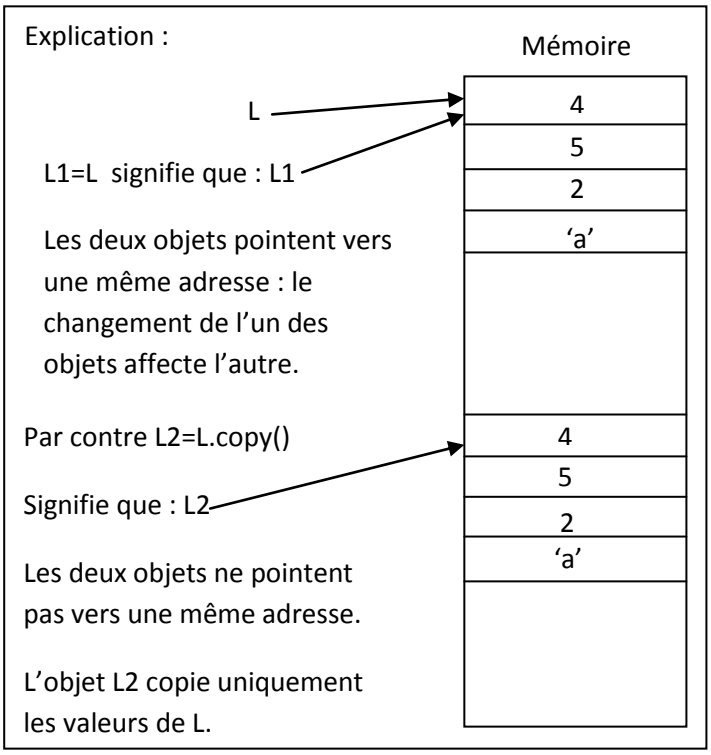
>>> L.append(12)
>>>L
[4, 5, 2, 'a', 12]

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 occurrence 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]                    >>>L[0,0]
1                                 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 listes 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]
>>>
```

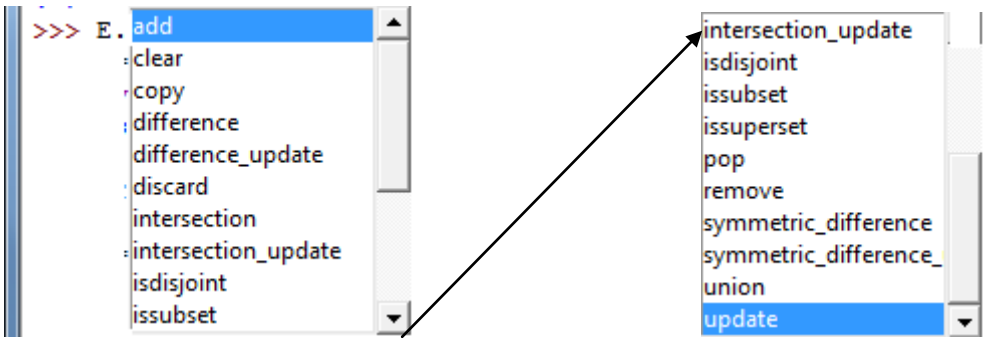
7.3. Les ensembles :

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.

```
>>> 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}
>>>
```

Déclaration :

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
```

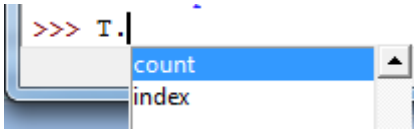
7.4. Les tuples :

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            # on ne peut pas modifier un tuple
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 :

7.5.1. Définition

Les dictionnaires sont des collections non ordonnées d'objets, c-à-d qu'il n'y a pas de notion d'ordre. On accède aux valeurs d'un dictionnaire par des clés.

Exemple :

```
>>> a = {}                  # a est un dictionnaire vide
>>> z = dict ()             # crée aussi un dictionnaire vide
>>> a['nom'] = 'girafe'     # ajouter la clé 'nom' avec la valeur 'girafe'
>>> a['taille'] = 5.0       # ajouter la clé 'taille' avec la valeur 5.0
>>> a['poids'] = 1100       # ajouter la clé 'poids' avec la valeur 1100
>>> print(a)                # afficher le dictionnaire a
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
>>> a['taille']             # afficher la valeur de la clé 'taille'
5.0
```

En premier, on définit un dictionnaire vide avec les symboles {} (tout comme on peut le faire pour les listes avec []). Ensuite, on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs (une par clé). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste). Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style dictionnaire ['cle'].

Opération	**signification**
x in d	vrai si x est une des clés de d
x not in d	réciproque de la ligne précédente
d[i]	retourne l'élément associé à la clé i
len(d)	nombre d'éléments de d
min(d)	plus petite clé
max(d)	plus grande clé
del d[i]	supprime l'élément associé à la clé i
list(d)	retourne une liste contenant toutes les clés du dictionnaire d.
dict(x)	convertit x en un dictionnaire si cela est possible

Méthode	**signification**
d.copy()	Retourne une copie de d.
d.get(k[; x])	Retourne d[k] si la clé k existe. Sinon la valeur None est retournée à moins que le paramètre optionnel x soit renseigné ; auquel cas, c'est ce paramètre qui sera retourné.
d.clear()	Supprime tous les éléments du dictionnaire.
d.update(d1)	Pour chaque clé k de d, d[k] = d1[k]
d.setdefault(k[; x])	Retourne d[k] si la clé k existe, sinon, affecte x à d[k].
d.popitem()	Retourne un élément et le supprime du dictionnaire.

7.5.2. Déclaration

Syntaxe générale :

d1={clé1 :val1, clé2 :val2, clé3 :val3, …….}

d2=dict(zip([clé1, clé2,clé3,….],[val1, val2, val3,….]))

modifier les éléments d'un dictionnaire

d1[clé4]=val4

d2[clé4]=val4

7.5.3. Exemples

```
>>> d={1:"lundi",2:"mardi",3:"mercredi",4:"jeudi",5:"vendredi",6:"samedi",7:"dimanche"}
```

ou

```
>>> d1=dict(zip([1,2,3,4,5,6,7],["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"]))
```

Ce qui donne

```
{1: 'lundi', 2: 'mardi', 3: 'mercredi', 4: 'jeudi', 5: 'vendredi', 6: 'samedi', 7: 'dimanche'}
```

Une autre façon de définir un dictionnaire :

```
>>> d2 = dict([("abcdef"[i], i) for i in range(len("abcdef"))])
>>> d2
{'a': 0, 'd': 3, 'c': 2, 'b': 1, 'e': 4, 'f': 5}
```

Modifier les éléments d'un dictionnaire :

```
>>> d[1]="monday"
>>> d
{1: 'monday', 2: 'mardi', 3: 'mercredi', 4: 'jeudi', 5: 'vendredi', 6: 'samedi', 7: 'dimanche'}
>>> d[8]="autre jour"
>>> d
{1: 'monday', 2: 'mardi', 3: 'mercredi', 4: 'jeudi', 5: 'vendredi', 6: 'samedi', 7: 'dimanche', 8: 'autre jour'}
```

7.5.4. Les méthodes keys(), values(), et items

Les méthodes keys() et values() renvoient respectivement la liste des clés et la liste des valeurs d'un dictionnaire

```
>>> d.values()
dict_values(['monday', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche', 'autre jour'])
>>> d.keys()
dict_keys([1, 2, 3, 4, 5, 6, 7, 8])
```

La méthode items() renvoie la liste des tuples (clé,valeur) d'un dictionnaire

```
In [35]:  d2.items()
Out[35]: dict_items([('a', 0), ('c', 2), ('d', 3), ('f', 5), ('b', 1), ('e', 4)])
```

7.6. Les chaines de caractères :

7.6.1. Définition

Une chaine de caractères contenant un ensemble de caractères délimités par :
Des guillemets simples : chaine1='Bonjour' #chaine sur une seule ligne
Des guillemets doubles : chaine1=''Bonjour'' #chaine sur une seule ligne
Des guillemets triples : chaine1='''''Bonjour Tout le monde '' '' '' #chaine sur plusieurs lignes

7.6.2. 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 guillemet avec un r :

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

7.6.3. 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 dans une chaine	>>> f=S.count('tout')
	>>> f donne 0 # la sous chaine n'existe pas
	>>> f=S.count('Tout') >>> print(f) donne 1

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

upper() : convertit une chaine en majuscule	>>> Sm=S.upper() >>> print(Sm) BONJOUR TOUT LE MONDE
capitalize() : convertit le 1ere lettre d'une chaine en majuscule.	>>> SM=S.capitalize() >>> print(SM) donne Bonjour tout le monde
title() : convertit tous les 1er lettres des mots d'une chaine en majuscule.	>>> t=S.title() >>> print(t) donne Bonjour Tout Le Monde
Swapcase() : intervertit les lettres majuscules et minuscules	>>> 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

7.7 Les Tableaux et les matrices :

Pour travailler avec les tableaux de données multidimensionnels on utilise le module ***numpy*** (*voir paragraphe 8.1 page 31*). C'est un module utilisé dans presque tous les projets de calcul numérique sous Python. Il fournit des structures de données performantes pour la manipulation de vecteurs et matrices.

Pour utiliser numpy il faut commencer par l'importer : **import numpy as np**

On pourrait aussi faire : **from numpy import ***

Dans la terminologie numpy, les vecteurs et les matrices sont appelés ***arrays***.

Avec la première importation, on peut utiliser les fonctions du module ***numpy*** de la manière suivante :

np .nom_fonction(…)

Avec la deuxième importation, on peut utiliser directement ces fonctions sans les précéder par "np.".

7.7 .1 Création des tableaux

Plusieurs possibilités sont offertes par le module ***numpy*** pour créer un tableau. La plus commune est l'utilisation des fonctions telles que : array, arange, ones, zeros,empty, etc.

- Au moyen de la fonction numpy.array

Créer un tableau à une dimension V contenant les élément 1,2,3,4

```
V = np.array([1,2,3,4])
```

Créer une matrice M 4×3

```
M = np.array([[16,2,3],[0,8,3],[1,72,83],[1,3,3]])
```

Remarques

- Dans ce cas, Les tableaux V et M sont tous deux du type ***ndarray*** (fourni par ***numpy***)
- On peut définir le type de manière explicite en utilisant le mot clé ***dtype*** en argument:

```
M = np.array([[1, 2], [3, 4]], dtype=complex)
```

- Autres types possibles avec dtype : int, float, complex, bool, etc.
- On peut aussi spécifier la précision en bits: int64, int16, float128, complex128.

- Au moyen de la fonction numpy.arange

On peut créer un vecteur à l'aide de la fonction ***arange,*** c'est une fonction similaire à la fonction ***range*** de Python :

np.arange(M,N,P) : tableau contenant les éléments de M à N avec pas P (M et P sont optionnels. Dans ce cas les valeurs par défaut sont M=0 et P=1)

```
a = np.arange(10)  ⇔ a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Au moyen de la fonction numpy. empty

```
np.empty(N,dtype='int') : tableau vide de N entiers

np.empty(N,dtype='float') : tableau vide de N réels
```

dtype est optionnel dans toutes les créations de tableaux et fixé par défaut à ***float***

***- Au moyen de la méthode* reshape**

C'est une méthode qui permet de redimensionner un tableau selon les paramètres communiqués à cette méthode :

Exemple : Créer une matrice

```
import numpy as np
A=np.empty(8,dtype='float')#A est à présent un vecteur de 8 réels
A=A.reshape(4,2)          #A devient maintenant une matrice 4×2 de réels
print("le vecteur A est:",A)
for i in range(4):
   for j in range(2):
      A[i][j]=i*j
P rint("A devient maintenant une matrice 4×2 de réels:")
Print(A)
```

Ce qui donne :

le vecteur A est: [0. 0. 0. 0. 0. 0. 0. 0.]
A devient maintenant une matrice 4×2 de réels:
[[0. 0.]
[0. 1.]
[0. 2.]
[0. 3.]]

- Autres possibilités

np.zeros(N) : tableau de N zéros.
np.eye(n) : tableau 2D carré de taille n x n, avec des *uns* sur la diagonale et des *zéros* partout ailleurs.
np.ones((N,M)) : tableau de dimension (N,M) de uns (matrice).
np.zeros_like(a) : tableau de zéros de même dimension et type que le tableau a.
np.ones_like(a) : tableau de uns de même dimension et type que le tableau a.
np. empty _like(a) : tableau vide de même dimension et type que le tableau a.
np.linspace(a,b,N) : tableau contenant N éléments uniformément répartis de a à b. (a et b sont compris)
b=a.copy() : copie d'un tableau a dans un tableau b

7.7.2 Dimensions d'un tableau

Soit la Matrice M définie par : `M = np.array([[1, 2], [7, 4], [0, 14]])`

Fonction	utilisation	Explication
alen(…)	>>np.alen(M) 3	Cette fonction (ce n'est pas un attribut) donne la première dimension d'un tableau (la taille pour un vecteur, le nombre de lignes pour une matrice).
size(…,0)	>>np.size(M,0) 3	np.size(a,0) donne le nombre de lignes d'une matrice.
size(…,1)	>>np.size(M,1) 2	np.size(a,1) donne le nombre de colonnes d'une matrice.

Attribut	utilisation	Explication
shape	>>M.shape (3,2)	Indique le format du tableau, sous la forme d'un tuple du nombre d'éléments dans chaque direction : ici le couple (3, 2) indique qu'il y a trois lignes et deux colonnes.
Size	>>M.size 6	donne le nombre total d'éléments.
ndim	>>M.ndim 2	renvoie le nombre d'indices nécessaires au parcours du tableau (usuellement : 1 pour un vecteur, 2 pour une matrice).

7.7.3 Opérations classiques et calcul matriciel

Les opérations classiques sur la matrices sont disponibles à l'aide de numpy : addition, multiplication par un scalaire, produit matriciel...Voici quelques exemples de ces opérations.

```
import numpy as np
A = np.random. rand ( 3 ,3 )
B=np . diag ( [ 1. , 2. , 3. ] )
v = np . array ( [ 3. , 4. , 5. ] , float )
# addition
C1 = A+B
C2 = 2.+A
# multiplication
D1 = 2*A     # coefficients de A multipliés par2
D2 = B**3   # coefficients de B à la puissance
D3 = A*B     #  multiplication terme à terme
D4 = np . dot (A,B)   # multiplication matricielle
D5 = np . dot (A, v )  # produit matrice / vecteur
D6 = np . kron (A,B) #produi t de Kronecker
# test
E1 = A<B  # renvoie une matrice de booléens effectuant le test
bo = np . array ( [ 1, 0., 0 ] , bool )
E2=B[ bo]  # extrait les éléments de B qui correspondent à la valeur vraie de bo
E3=A[A>0.5]
```

- 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

- Complexe conjugué - conj()

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

7.7.4 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 à 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 = np.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 = np.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]])
```

7.7.5 Algèbre linéaire

- Déterminant - det()

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

- Inverse - inv()

```
>>> a = np.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 = np.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.

7.7.6 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]])
```

8. Quelques commandes et fonctions des bibliothèques numpy, scipy, matplotlib et serial

8.1. La bibliothèque numpy :

numpy est une bibliothèque *numérique* apportant le support efficace de larges tableaux multidimensionnels, et de routines mathématiques de haut niveau (fonctions spéciales, algèbre linéaire, statistiques, etc.).
Les opérations classiques sur la matrices sont disponibles à l'aide de numpy : addition, multiplication par un scalaire, produit matriciel....
Bien entendu, numpy permet facilement de faire du calcul numérique matriciel : calcul du rang d'une matrice, inversion d'une matrice, résolution de systèmes linéaires.

8.2. La bibliothèque Scipy :

Scipy est une bibliothèque numérique d'algorithmes et de fonctions mathématiques, basée sur les tableaux ndarray, complétant ou améliorant (en terme de performances) les fonctionnalités de numpy.

Par exemple:

- Functions spéciales : ***scipy.special*** (fonctions de Bessel, erf, gamma, etc.)
- Intégration numérique : ***scipy.integrate*** (intégration numérique ou d'équations différentielles)
- Méthodes d'optimisation : ***scipy.optimize*** (minimisation, moindres-carrés, zéros d'une fonction, etc.)
- Interpolation : ***scipy.interpolate*** (interpolation, splines)
- Transformées de Fourier : ***scipy.fftpack***
- Traitement du signal : ***scipy.signal*** (convolution, corrélation, filtrage, ondelettes, etc.)
- Statistiques : ***scipy.stats (***fonctions et distributions statistiques)
- Traitement d'images multi-dimensionnelles : ***scipy.ndimage***
- Entrées/sorties : ***scipy.io***

- <u>Intégrer ou dériver une fonction</u> :

La bibliothèque `scipy.integrate` : `from scipy.integrate import*`

Pour intégrer une fonction, on utilise la commande `quad(fonction, borne inférieure, borne supérieure)`

Exemple1 : L'intégration de la fonction f:x→ x^3 entre 0 et 1.

```
from scipy.integrate import quad
a = 0 # borne inférieure
b = 1 # borne supérieure
def f(x): # Fonction à intégrer
      return x**3
J, err = quad(f, a, b)
print ("l'integrale vaut ", J)
print ("erreur = ", err)
```

Exemple2 : Dérivation la fonction $f:x \rightarrow x^3$, pour x=2.

```
from scipy.misc import derivative
def f(x): # Fonction à dériver
      return x**3
d1=derivative(f, 2)
print(d1)
d2=derivative(f, 2, 0.01)
print(d2)
```

- **Résolution d'une équation différentielle :**

✓ **Equation différentielle du premier ordre** : y'-ay=0

def f(t,y) :
    return -a*y

t=numpy.linspace(0,10, 200)# résolution sur l'intervalle [0 ..10]
y0= 1 # condition initiale puis **sol=odeint(f,y0,t)**

```
from scipy.integrate import odeint
#Pour résoudre l'équation
différentielle
import numpy as np
t=np.linspace(0,10,20)
print("donner la valeur de a:",)
a=input()
a=int(a)
def f(t,y):
    return a*y
#Condition initiale
y0=1# y(x0) = y0
sol=odeint(f,y0,t)
```

✓ **Equation différentielle du second ordre :** $x''+(w0/Q).x'+w0^2.x=0$.

Pour résoudre une telle équation différentielle, il suffit de réécrire l'équation différentielle du second ordre en système de deux équations différentielles du premier ordre. Par exemple, prenons l'équation de l'oscillateur harmonique amorti.

On peut réécrire cette équation en un système de deux équations différentielles d'ordre 1

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\dfrac{w_0}{Q}v - w_0^2 x \end{cases}$$

On résouds le sytème précédent avec la même procédure que celle utilisée pour un système d'équations différentielles d'ordre 1.

```
from scipy import *
from scipy.integrate import odeint # Module de résolution des équations différentielles

omega0 = 2*pi
Q = 10

def deriv(syst, t):
    x = syst[0] # Variable1 x
    v = syst[1] # Variable2 v
    dxdt = v # Equation différentielle 1
    dvdt = -omega0/Q*v-omega0**2*x # Equation différentielle 2
    return [dxdt,dvdt] # Dérivées des variables
```

.....suite page suivante

```
from scipy import *
from scipy.integrate import odeint # Module de résolution des équations différentielles

omega0 = 2*pi
Q = 10

def deriv(syst, t):
    x = syst[0] # Variable1 x
    v = syst[1] # Variable2 v
    dxdt = v # Equation différentielle 1
    dvdt = -omega0/Q*v-omega0**2*x # Equation différentielle 2
    return [dxdt,dvdt] # Dérivées des variables
```

- ✓ **Racine(s) d'une équation**

On recherche les zéros de la fonction f(x)=0

La bibliothèque **scipy.optimize :** from **scipy.optimize import ***

Sol= **bisect**(fonction,borne inférieure, borne supérieure)

Sol= **nweton** (fonction, saderivee, x_depart).

On recherche les zéros de la fonction $f: x \to x^4-5x^2+4$ qui se factorise en $(x^2-4)(x^2-1)$ dont les racines sont $-2,-1,1$ et 2.

Exemple :

```
from scipy import *
from scipy.optimize import bisect # Module de recherche des
zéros

x = linspace(-2.5,2.5,300) # Abscisses

def f(x): # Fonction à étudier
    res = x**4-5*x**2+4
    return res
zero1 = bisect(f, -2.5, -1.5) # Recherche du premier zéro
print(zero1)
zero2 = bisect(f, -1.5, -0.5) # Recherche du second zéro
print(zero2)
zero3 = bisect(f, -0.5, 1.5) # Recherche du troisième zéro
print(zero3)
zero4 = bisect(f, 1.5, 2.5) # Recherche du quatrième zéro
print(zero4)
```

- ✓ **Résoudre un système d'équations**

Pour résoudre un système d'équations, on utilise la commande fsolve(système, initialisation)

Exemple :

```
from scipy.optimize import fsolve
def syst(var): # définition du système

    x, y, z = var[0], var[1],var[2] # définition des variables
    eq1 = x +10*y-3*z-5
    eq2 = 2*x-y+2*z-2
    eq3 = -x+y+z+3
    res = [eq1, eq2, eq3]
    return res
x0, y0, z0 = 0, 0, 0 # Initialisation de la recherche des solutions
numériques
sol_ini = [x0, y0, z0]
T=fsolve(syst, sol_ini)
print(T)
```

ou bien

```
from numpy import linalg
import numpy as np
A=np.array([[1,10,-3],[2,-1,2],[-1,1,1]])
B=np.array([-5,-2,3])
sol=np.linalg.solve(A,B)
print(sol)
```

8.3. La bibliothèque matplotlib :

– La librairie **matplotlib** fournit un ensemble de fonctions permettant de représenter toutes sortes de graphiques.

Il existe deux interfaces pour deux types d'utilisation:

- **pylab**: interface procédurale, très similaire à MATLAB et généralement réservée à l'analyse interactive:

Exemple1 : Représentation de la fonction y = sin(x).

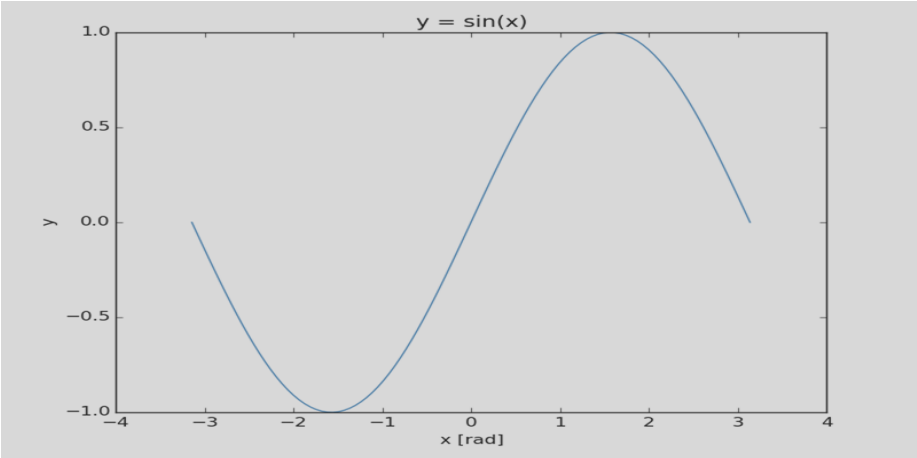
```
from pylab import * # DÉCONSEILLÉ DANS UN SCRIPT!
x = linspace(-pi, pi, 100) # pylab importe numpy dans l'espace courant
y = sin(x)
plot(x, y) # Trace la courbe y = y(x)
xlabel("x [rad]") # Ajoute le nom de l'axe des x
ylabel("y") # Ajoute le nom de l'axe des y
title("y = sin(x)") # Ajoute le titre de la figure
show() # Affichage de la figure
```

- **matplotlib.pyplot:** interface orientée objet, préférable pour les scripts

```
import numpy as N
import matplotlib.pyplot as P

x = N.linspace(-N.pi, N.pi, 100)
y = N.sin(x)
fig, ax = P.subplots() # Création de la figure et de l'axe
ax.plot(x, y) # Tracé de la courbe dans l'axe
ax.set_xlabel("x [rad]")
ax.set_ylabel("y")
ax.set_title("y = sin(x)")
P.show() # Affichage de la figure
```

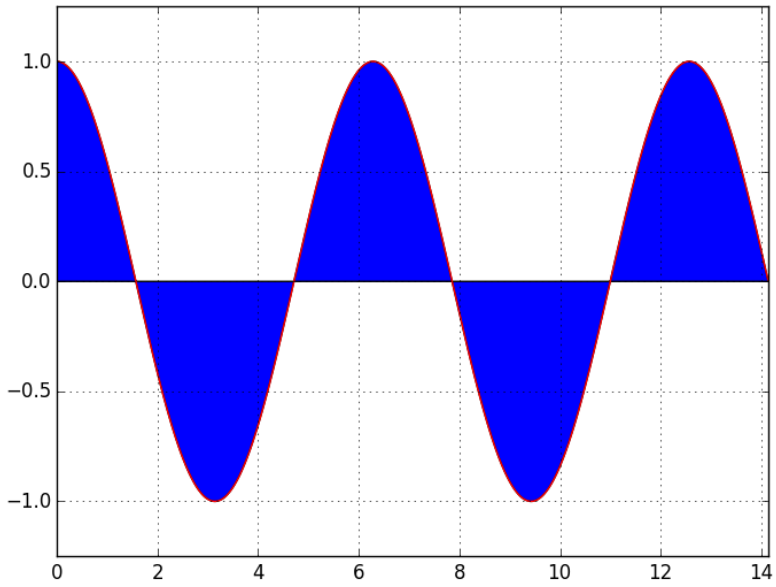
Dans les deux cas, le résultat est le même:



Exemple2 : Illustrer l'aire de l'intégration avec matplotib, en utilisant la méthode ax.fill_between(x, 0, function(x)) comme dans cet exemple: l'intégration de la fonction $f:x \rightarrow cos(x)$ entre 0 et $9\pi/2$.

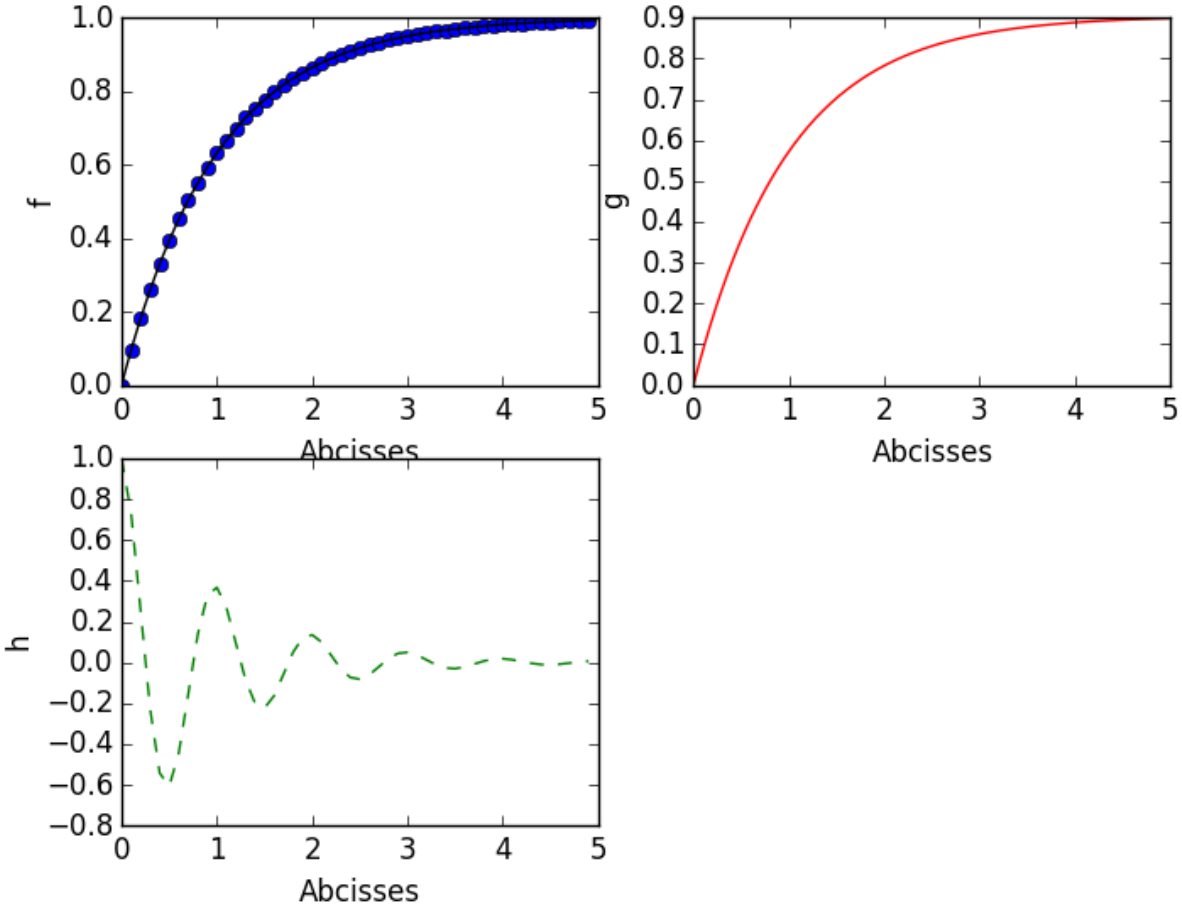
```
from scipy.integrate import quad
from pylab import *

import numpy as np
xmin = 0.0
xmax = 9.0 * ( np.pi / 2.0 )
def function(x):
    return np.cos(x)
res, err = quad(function, xmin, xmax)
print( 'norm: ', res)
t = arange(xmin, xmax, 0.01)
ax = subplot(111)
ax.plot(t, function(t),'r-')
ax.grid(True)
ax.set_xlim(xmin,xmax)
ax.set_ylim(-1.25,1.25)
ax.fill_between(t, 0, function(t))
plt.savefig('IntegraleSimplePython.png')
show()
```



Exemple3 : Affichage de plusieurs courbes

```
import numpy as np
import matplotlib.pyplot as plt
def f(t):
        return np.exp(0)-np.exp(-t)
def g(t):
        return np.exp(-0.1)*np.exp(0)-np.exp(-0.1)*np.exp(-t)-np.exp(-
9000)*np.exp(0)-np.exp(-9000)*np.exp(-t)
def h(t):
        return np.exp(-t) * np.cos(2*np.pi*t)
t1 = np.arange (0.0,5.0,0.1)
t2 = np.arange (0.0,5.0,0.02)
plt.figure(1)
plt.subplot(221)
plt.xlabel("Abcisses")
plt.plot(t1,f(t1),"bo",t2,f(t2),"k")
plt.xlabel("Abcisses")
plt.ylabel("f")
plt.subplot(222)
plt.plot(t2,g(t2),"r")
plt.xlabel("Abcisses")
plt.ylabel("g")
plt.subplot(223)
plt.plot(t1,h(t1),"g--")
plt.xlabel("Abcisses")
plt.ylabel("h")
plt.show()
```



8.4. La bibliothèque serial (La carte Arduino) :

Les cartes Arduino possèdent un microcontrôleur facilement programmable ainsi que de nombreuses entrées-sorties. Plusieurs cartes Arduino existent et qui se différencient par la puissance du microcontrôleur ou par la taille et la consommation de la carte.L'ensemble des cartes Arduino se programment en C++ à l'aide d'un logiciel de programmation gratuit et open-source fourni par Arduino.

L'offre en cartes ARDUINO et cartes compatibles est pléthorique et il est bien souvent difficile à un débutant de s'y retrouver et de déterminer quelle carte conviendra à son projet. Si on se limite aux cartes ARDUINO officielles dans leurs versions de base, cela devient plus simple.
Voici quelques cartes arduino officielles très utilisées :

- Arduino Uno Rev 3 (ou.. la Super Star !) – DEBUTANT

Elle conviendra mieux à des petits projets, caractérisée par son faible nombre de ports (6 analogiques et 14 numériques 1/0 dont 6 pwm).

- Arduino Leonardo (vers des projets plus complexes…) – DEBUTANT/INTERMEDIAIRE

Elle a le même nombre de ports que la carte Arduino Uno (12 analogiques et 20 numériques dont 7 pwm). Son gros plus : un port USB, qui permettra d'émuler un clavier ou une souris

- Arduino Mega (la Challenger !) – INTERMEDIAIRE/EXPERT

La carte Arduino Mega est **la plus vendue après l'Arduino Uno**. Dotée d'un fonctionnement identique, la **seule différence** sera **le nombre de ports disponibles** (16 analogiques et 54 numériques dont 14 pwm, contre 6 analogiques et 14 numériques 1/0 dont 6 pwm).

- Arduino Due (Intelligence Artificielle et algorithmes..) – EXPERT

Possédant le même nombre de ports que l'Arduino Mega, mais ayant une **puissance largement supérieure**, cette carte sera utilisée dans des projets de **création d'intelligence artificielle** pour des robots mobiles.

- Arduino nano (projets miniatures) – INTERMEDIAIRE/EXPERT

Une version **minimaliste et augmentée** de l'Arduino Uno, elle possède plus de ports que cette dernière (8 analogiques et 14 numériques dont 6 pwm, contre 6 analogiques et 14 numériques 1/0 dont 6 pwm) malgré sa taille réduite.

- Arduino Yun – INTERMEDIAIRE

Cette carte Arduino possède le **même processeur que l'Arduino Leonardo** (l'Atmel ATmega32U4) mais possède une différence de taille : un **module générant un réseau wifi** !
Connection possible avec un ordinateur ou un smartphone pour récupérer les infos de la carte ou bien la contrôler via un navigateur web.

- ***Communication Arduino-Python via port SERIE***

- Python permet de réaliser facilement une liaison via USB entre un PC et une carte arduino. Nous allons l'illustrer par un exemple qui lit ce qu'une Arduino Uno envoie et un autre qui envoie un nombre à la Arduino. La diode de celle-ci doit alors clignoter du nombre envoyé.

Pour la lecture de ce qu'envoie la Arduino :

```
# Module de lecture/écriture du port série
from serial import *
# Port série COM4
# Vitesse de baud : 9600
# Timeout en lecture : 1 sec
# Timeout en écriture : 1 sec
with Serial(port="COM4", baudrate=9600, timeout=1, writeTimeout=1) as port_serie:
    if port_serie.isOpen():
        while 1:
            ligne = port_serie.readline()
            print (ligne)
```

Pour l'envoie de données vers la Arduino, *en python*:

```
from serial import *
nombre = input("Entrez un nombre : ")
port_serie.write(nombre.encode('ascii'))
```

- Notez qu'il est possible d'envoyer plusieurs nombres à la suite au port série, puisque la fonction "Serial.write()" en Python prend en argument un tableau d'entiers.

Exemple : Lecture via le port série les données en provenance de la carte Arduino.

Dans ce programme on va afficher (par la fonction plot de la bibliothèque matplotlib.pyplot) les points au fur et à mesure qu'ils arrivent sur le port. La carte Arduino fournit des valeurs toute les secondes que l'on veut tracer sur une courbe en fonction du temps .

```
import serial
import matplotlib.pyplot as plt
import time
mesures=[]
temps=[]
serie=serial.Serial('COM4',9600)    # ouvre une liaison serie en 9600 bps
plt.style.use('bmh')
plt.ylabel("Mesures en °C ")
plt.xlabel("temps en s")
plt.ion()                           # on entre en mode interactif
debut=time.time()                   # mesure de l'instant initial
n=0
while (n<100):
  mesure= float(serie.readline()) # lit la donnée sur la liaison série
  mesures.append(mesure)                  # ajout de mesure a la liste des mesures
  t=time.time()-start      # calcul du temps écoule depuis l'instant initial
  temps.append(t)                 # ajout de instant a la liste des temps
  print (mesure,t)  # affiche dans la console les coordonnées du point
  plt.plot(temps, mesures, marker='*')  # trace la courbe
  plt.draw()              # affiche la courbe en mode interactif
  n=n+1
plt.ioff()                  # on quitte le mode interactif pour rendre
plt.show()                  # afficher la courbe
```