Langage Python - Techniques de base

1 Introduction

L'interpréteur L'interpréteur se reconnait facilement. C'est lui qui contient le triple chevron `>>>` qui est l'invite de Python (prompt en anglais) et qui signifie que Python attend une commande ; il peut s'agit également de `In[]:`.
L'esprit d'utilisation de l'interpréteur est un peu le même que celui d'une calculatrice.

L'éditeur. On peut y écrire des scripts, c'est-à-dire des programmes petits ou grands.

- Taper directement dans la console a deux inconvénients :
  - l'enregistrement n'est pas possible,
  - si plusieurs lignes d'instructions ont été tapées, les modifications ne sont pas aisées.
- On souhaite souvent enregistrer une telle suite particulière d'instructions pour pouvoir s'en resservir ultérieurement ou la modifier.
- Pour enchainer plusieurs instructions, l'éditeur est l'outil approprié. Cela permet de travailler avec des fichiers dans lesquels on place des suites d'instructions.
- En principe, il faut aller à la ligne après chaque instruction. Un décalage de début de ligne appelé indentation se fait automatiquement dans certains cas.
- Il est nécessaire d'enregistrer les fichiers avec l'extension .py pour Python.

Python comme calculatrice : les opérations. Les priorités opératoires sont respectées.

- L'addition : `+`
- La soustraction :`-`
- La multiplication : `*`
- La division : `\`
- L'exponentiation (puissance) : `**`

```
In[] : 2+2
Out[]: 4
In[] : 3+5*6
Out[]: 33
In[] : (50-5*6)/4
Out[]: 5.0
In[] : 5**2
Out[]: 25
```

Les nombres entiers sont de type `int`, ceux avec une partie décimale sont de type `float` (les nombres flottants).

La division `\` donne toujours un résultats de type `float`. On peut imposer un travail sur les entiers :

```
In[] : 17/3
Out[]: 5.666666666666667
In[] : 17//3 # Quotient de la division euclidienne
Out[]: 5
In[] : 17%3 # Reste de la division euclidienne
Out[]: 2
```

2 Affectations et égalité

2.1 Affectation : 1er aperçu

```
In[] : n=7
In[] : msg="Quoi de neuf ?"
In[] : pi=3.14159
```

Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :
- trois noms de variables, à savoir `n, msg` et `pi`,
- trois séquences d'octets, où sont encodés le nombre entier `7`, la chaine de caractères `Quoi de neuf ?` et le nombre réel `3,14159`.

Les trois instrutions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :
- créer et mémoriser un **nom de variable**,
- lui attribuer un **type** bien déterminé,
- créer et mémoriser une **valeur** particulière,
- établir un **lien** (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

Les trois noms de variables sont des **références**, mémorisées dans une zone particulière de la mémoire que l'on appelle espace de noms, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres.

```
In[] : n,type(n),id(n)
In[] : whos
```

Afficher la valeur d'une variable Pour afficher la valeur d'une variable à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis `<Enter>`. Python répond en affichant la valeur correspondante.

A l'intérieur d'un **programme**, vous utiliserez toujours la seconde possibilité : la fonction `print()`.

```
In[] : msg
"Quoi de neuf ?"
In[] : print(msg)
Quoi de neuf ?
In[] : print("pi=",pi,"et n=",n)
pi=3.14159 et n=7
```

2.2 Typage des variables

Sous Python, il n'est pas nécessaire d'écrire des lignes de programmes spécifiques pour définir le type de variables avant de pouvoir les utiliser.

Il vous suffit d'assigner une valeur à un nom de variable pour que celle-ci soit **automatiquement créée avec le type qui correspond au mieux à la valeur fournie**.

Ceci constitue une particularité intéressante de Python. On dira à ce sujet que **le typage des variables sous Python est un typage dynamique**, par opposition au **type statique**.

2.3 Affectation : état d'exécution

La mémoire de l'ordinateur est constitué d'une multitude de petites boites, un programme n'utilise en général que quelques unes de ces boites. Chaque boite utilisée par le programme a un nom et contient une valeur. On appelle **état de l'exécution d'un programme**, le triplet formé par le nombre de boites, le nom de chacune d'elles et la valeur qu'elle contient.

Exemples Dans l'état 1,

Etat 1

x	y
4	5

, exécuter la suite d'instructions :

```
In []: x=8
In []: y=9
In []: z=2
```

conduit aux états successfis :

Etat 2

x	y
8	5

Etat 3

x	y
8	9

Etat 4

x	y	z
8	9	2

L'affectation `x=y+3` est une instruction. Elle est composée d'une variable `x` et d'une expression `y+3`. On attribue une **valeur** à chaque expression.

Pour les expressions sans variables, comme `(2+5)*3`, dont la valeur est `21`, la valeur s'obtient simplement en effectuant les opérations présentes dans l'expression.

La valeur d'une expression qui contient des variables, comme `(2+x)*3`, se définit de la même manière, mais dépend de l'état dans lequel on calcule cette valeur.

Etat 1

x	y
3	5

Etat 2

x	y
4	5

Par exemple, la valeur de l'expression `(2+x)*3` dans l'état 1 est 15 alors que celle de cette même expression dans l'état 2 est 18.

2.4 Affectation : 2nd aperçu

Contrairement à ce que nos habitudes de calcul algébrique pourraient nous laisser penser, l'instruction `b=2*a` n'affecte pas à `b` le double de la valeur de `a` quelle que soit la valeur de `a` au long de la session Python. Au contraire, l'instruction `b=2*a` procède en deux temps :

- l'expression située à droite du signe `=` est évaluée, c'est-à-dire calculée en fonction de l'état de la mémoire à cet instant,
- ensuite, et seulement ensuite, l'interpréteur affecte au nom situé à gauche de l'instruction d'affectation (à savoir `b`) l'objet obtenu après évaluation de l'expression de droite.

L'objet `b` n'a aucune relation avec l'objet nommé `a`.

2.5 Variables

Les noms de variables sont des noms que vous choisissez librement. Efforcez-vous de bien les choisir : de préférence assez court, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir.

Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a→z, A→ Z) et de chiffres (0→ 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuéee, les cédilles, les espaces, les caractères spéciaux sont interdits, à l'exception du caractère `_` (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués).

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables certains mots réservés (ils sont utilisés par le langage lui-même :
`and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`.

Les entiers petits (de -5 à 256 le plus souvent) ont leur place mémoire fixe parce que Python sait qu'on les utilise souvent. On peut ajouter à la liste les booléens `True` et `False` et `None`.

Pour des entiers au-dela de 256 ou des flottants, on peut obtenir à chaque fois des `id` différentes.

2.6 Affectations multiples et affectations parallèles

Python permet les affectations multiples :

```
>>> x=y=42
```

Python permet aussi les affectations parallèles :

```
>>> x,y=(1,2) # ou encore
>>> x,y=1,2
```

3 Tests

3.1 L'exécution conditionnelle

Nous devons disposer d'instructions capables de **tester une certaine condition** et de modifier le comportement du programme en conséquence. La plus simple de ces instructions conditionnelles est l'instruction `if`.

L'exécution de :

```
a=150
if a>100:
    print("a dépasse la centaine")
```

affichera :

```
a dépasse la centaine
```

Alors que :

```
a=20
if a>100:
    print("a dépasse la centaine")
```

n'affichera rien.

L'expression placée après `if` est une **condition**. L'instruction `if` permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction **indentée** après le : est exécutée. Si la condition est fausse, rien ne se passe.

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

L'instruction `else` permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut augmenter le nombre de possibilités avec l'instruction `elif`.

3.2 Opérateurs de comparaison

x==y	x est égal à y
x!=y	x est différent de y
x<y	x est strictement plus petit que y
x<=y	x est inférieur ou égal à y
x>y	x est strictement plus grand que y
x>=y	x est supérieur ou égal à y

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes `==` et non d'un seul. Le signe `=` utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison.

Les opérateurs `<, >, ==, >=, <=, !=` compare les valeurs des deux objets. Les objets n'ont pas besoin d'être du même type.

Si ce sont tous les deux des nombres, ils sont convertis en un type commun.

```
In []: 2.0 == 2 # Renvoie True
```

Sinon, les opérateurs `==` et `!=` considèrent toujours des objets de différents types comme non égaux, alors que les opérateurs `<, >, >=, <=` retournent une `TypeError` lorsqu'ils comparent des objets de types différents qui n'implémentent pas ces opérateurs pour cette paire de type.

Les chaines de caractères sont comparées dans l'ordre lexicographique en utilisant les équivalents numériques des caractères

Les tuples et les listes sont aussi comparés avec l'ordre lexicographique en utilisant la comparaison élément par élément. Ainsi pour obtenir l'égalité, chaque élément doit être égal, les deux séquences doivent être du même type et avoir la même longueur.

Dans le cas de non égalité, les séquences sont comparées selon le premier élément qui diffère. Par exemple `[1,2,x] <= [1,2,y]` a la même valeur que `x <= y`.

Si l'élément correspondant n'existe pas, la séquence la plus courte est classée en premier. Par exemple, `[1,2] < [1,2,3]` est vraie.
`x < y <= z` est équivalent à `x < y` and `y <= z`, sauf que `y` n'est évalué qu'une seule fois dans le 1er cas (alors que dans les deux cas `z` n'est pas évalué si `x < y` est faux).

On peut chainer tous les opérateurs de comparaison et on peut chainer plusieurs opérateurs.

Vous pouvez donc écrire `x<y>z` même si ce n'est pas élégant.

3.3 Opérateurs sur les booléens

- La négation ou *non logique*, dont le symbole Python est `not`.
- La conjonction ou *et logique*, dont le symbole Python est `and`.
- La disjonction ou *ou logique*, dont le symbole Python est `or`.

Les opérateurs `and` et `or` sont dits *paresseux* : ils ne calculent pas ce qui est nécessaire pour évaluer une expression. Par exemple, dans une expression du type `condition1 and condition2`, si `condition1` vaut `False`, `condition2` n'est pas évaluée.

L'opérateur `not` a précédence sur `or` et `and`.
L'opérateur `and` a précédence sur `or`.

```
In []: True or False and False
Out[]: True
In []: (True or False) and False
Out[]: False
```

3.4 L'indentation

Indenter un programme, c'est-à-dire insérer des espaces blancs en début de ligne, est un moyen de visualiser le niveau d'imbrication auquel une instruction se trouve.

Dans Python, l'indentation n'est pas qu'une aide à la lecture mais change la signification des programmes.

```
if x==4:
    y=1
else:
    y=2
    z=3
print(x,y,z)
```

Si $x = 4$, l'exécution du programme renverra une erreur `z non défini`. Si $x = 1$, l'exécution du programme renverra `1 2 3`.

```
if x==4:
    y=1
else:
    y=2
z=3
print(x,y,z)
```

Si $x = 4$, l'exécution du programme renverra `4 1 3`. Si $x = 1$, l'exécution du programme renverra `1 2 3`.

- La construction avec l'instruction `if` est un exemple d'**instructions composées**. Sous Python, les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs isntruction indentées sous cette ligne d'en-tête.

- S'il y a plusieurs instructions indentées sous la ligne d'en-tête, **elles doivent l'être exactement au même niveau** (compter un décalage de 4 caractères, par exemple). Ces instructions indentées constituent un **bloc d'instructions**. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certains conditions définies dans la ligne d'en-tête.

- Il est parfaitement possible d'**imbriquer** les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes.

3.5 Règles de syntaxe

- Dans de nombreux langages de programmations, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère fin de ligne qui joue ce rôle.
- Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que `begin` et `end`). En Java, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper ni d'indentation, ni de sauts à la ligne. On conseille cependant à tous ceux qui utilisent ces langages de se servir aussi des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.
- Avec Python, vous **devez** utiliser les sauts à la ligne et l'indentation ; il n'y a pas d'autres symboles délimiteurs de blocs.

3.6 Les commentaires

- Dès que l'on écrit un programme de plus d'une dizaine de lignes, **il est indispensable d'ajouter des commentaires** dans ce programme, autrement dit des lignes écrites en langue naturelle que la machine ne cherche pas à interpréter comme des instructions et qui expliquent le rôle des différentes parties du programme.
- Ces commentaires permettent à un programmeur de comprendre un programme écrit par un autre programmeur ou par lui-même longtemps auparavant. Ces commentaires doivent donner des informations supplémentaires sur le sens du programme.
- Pour préciser qu'une ligne est un commentaire, on la fait précéder par `#` en Python. Si l'on veut écrire un commentaire sur plusieurs lignes, il faut faire précéder chacune d'entre elles par ce symbole.

4 Les boucles

Permettre à une instruction d'être exécutée plusieurs fois au cours de l'exécution d'un programme est le but d'une instruction particulière : la **boucle**.

- La boucle `while` : c'est une instruction de la forme while expression, instruction. L'instruction est appelée **corps** de cette boucle. Exécuter la boucle a pour effet d'exécuter l'instruction plusieurs fois tant que la valeur de l'expression est égale à `True`. Le nombre d'itérations n'est pas fixé avant le début d'exécution de la boucle.
- La boucle `for` : c'est une instruction de la forme `for i in range(a,b): instruction` où `i` est une variable, `range(a,b)` désigne la séquence `a,a+1,a+2,...,b-1` (on peut aussi avoir une incrémentation différente de 1).
  Exécuter la boucle `for` à pour effet d'exécuter l'instruction `b-a` fois dans des états dans lesquels la valeurs de la variable `i` est successivement `a,a+1,...,b-1`.

4.1 Boucle while

```
a=0
while a<7:
    a=a+1
    print(a)
```

1. Avec l'instruction `while`, Python commence par évaluer la validité de la condition fournie par l'évaluation de l'expression.
2. Si la condition se révèle fausse, tout le bloc qui suit est ignoré et l'exécution du programme se poursuit au bloc suivant.
3. Si la condition est vraie, Python exécute tout le bloc d'instruction constituant le corps de la boucle.
4. Lorsque les instructions ont été exécutées, il y a une une itération et le programme boucle, c'est-à-dire que l'exécution du programme reprend à la ligne contenant l'instruction `while`. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite.

Remarques :
- La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté une valeur).
- Si la condition est fausse au départ, le corps de la bouble n'est jamais exécuté.
- Si la condition reste toujours vraie, le corps de la boucle est répété indéfiniment. Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par `while`.

```
a=0
while a<12:
    a=a+1
    print(a,a**2,a**3)

a,b,c=1,1,1
while c<11:
    print(b,end=" ")
    a,b,c=b,a+b,c+1
```

4.2 Boucle for en Python

```
for n in range(1,10):
    print(n,n**2,n**3)

for n in range(1,10,3):
    print(n,n**2,n**3)
```

```
liste=['chien','chat','crocodile','éléphant']

for animal in liste:
    print('nb de lettres pour ',animal,'=',len(animal))
for index in range(len(liste)):
    print(index,liste[index])
```

L'instruction `for` permet d'écrire des boucles, dans lesquelles l'itération traite successivement tous les éléments d'une séquence donnée.

- Le nom qui suit le mot réservé `in` est celui de la séquence qu'il faut traiter.
- Le nom qui suit le mot réservé `for` est celui que vous choisissez pour la variable destinée à contenir successivement tous les éléments de la séquence.
- Cette variable est définie automatiquement (il est inutile de la définir au préalable), et son type est automatiquement adapté à celui de l'élément de la séquence qui est en cours de traitement (dans le cas d'une liste, tous les éléments ne sont pas nécessairement du même type).
- La fonction `range()` génère par défaut une séquence de nombres entiers de valeurs croissantes et différant d'une unité. Attention, avec `range(a,b)`, la séquence générée est `a,a+1,...,b-1`.

```
In []: list(range(10))
Out[]: [0,1,2,3,4,5,6,7,8,9]
In []: list(range(5,13))
Out[]: [5,6,7,8,9,10,11,12]
In []: list(range(3,16,3))
Out[]: [3,6,9,12,15]
```

La boucle for, cas particulier de la boucle while

L'instruction

```
for i in range(a,b):
    instruction
```

peut être vu comme une manière plus simple d'écrire l'instruction

```
i=a
while i<b:
    instruction
    i=i+1
```

```
for d in range(1,101):
    if 100%d==0:
        print(d)
```

```
d=1
while d<101:
    if 100% d==0:
        print(d)
    d=d+1
```

4.3 Break - Continue - Pass

Le mot-clé `break` permet d'interrompre une boucle quelle que soit la correction de la boucle. Python sort immédiatement de la boucle et exécute le code qui suit la boucle, s'il y en a.

Parfois, `break` est véritablement utile et fait gagner du temps. Mais ne l'utilisez pas à outrance, préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un `break`.

Le mot-clé `continue` permet de continue une boucle, en repartant directement à la ligne du `while` ou `for`.

Si à un endroit on a syntaxiquement besoin d'un bloc mais qu'il n'y a rien à faire, on peut utiliser l'instruction `pass`, qui justement ne fait rien.

4.4 Terminaison de boucle : Compteur

```
c,p=10,1
while c>0:
    p=p*2
    c=c-1
```

Dans le programme précédent, la variable `c` joue le rôle d'un compteur.

Mathématiquement, il est garanti que l'on sorte de la boucle car :
– la valeur de `c` est un entier strictement positif ;
– elle décroit strictement après chaque itération.
Comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations.

Il n'est pas nécessaire d'avoir une variable du type compteur de boucle. Il suffit qu'une quantité vérifie ces deux propriétés : être un entier positif tout au long de l'algorithme et décroitre strictement après chaque opération. On appelle cette quantité un **variant de boucle**.

4.5 Variant de boucle - Exemple de la division euclidienne

```
q,r=0,n
while r>=d:
    q=q+1
    r=r-d
```

L'état initial contient deux variables `n` et `d` et, à l'issue de l'exécution, on souhaite que les deux variables `q` et `r` contiennent le quotient et le reste dans la division euclidienne de `n` par `d`.

Ici, si la valeur de `d` est un entier strictement positif, la variable `r` reste positive tout au long de l'algorithme et, après chaque itération, elle diminue de `d`, donc elle décroit strictement. Ainsi ce programme se termine-t-il.

4.6 Invariant de boucle

Lorsque l'on a écrit un programme, il reste à vérifier qu'il est correct, c'est-à-dire qu'il calcule bien ce qu'on attend ; on peut tester quelques cas significatifs, mais il est beaucoup plus satisfaisant de démontrer qu'il est correct dans tous les cas. Une des manières les plus efficaces de le faire est d'établir un **invariant de boucle**, c'est-à-dire une propriété qui est vérifiée tout au long de l'exécution d'une boucle.

Cette démarche est à rapprocher du raisonnement par récurrence : en ne s'intéressant qu'aux valeurs initiales et à leur évolution au cours d'une seule itération, on peut en déduire des propriétés valides quel que soit le nombre d'itérations.

Une invariant de boucle est une propriété qui
– est vérifiée avant d'entrer dans la boucle,
– si elle est vérifiée avant une itération, elle est vérifiée après celle-ci,
– lorsqu'elle est vérifiée en sortie de boule, elle permet d'en déduire que le programme est correct.
Invariant de boucle - Exemple de la Division euclidienne.

5 Types - introdution

5.1 Les types scalaires

- le type entier (`int`) : réservé à une variable dont la valeur est un entier relatif **stocké en valeur exacte**.
- le type flottant (`float`) : un nombre à virgule flottante correspond à une variable dont la valeur est un nombre réel, **stocké en valeur approchée** sous la forme d'un triplet (s, m, e) où s désigne le signe (dans $\{-1, 1\}$), m la mantisse (aussi appelée significande) et e l'exposant. Un tel triplet représente le nombre décimal $s \cdot m \cdot b^e$ en notation scientifique où b est la base de représentation, à savoir 2 sur les ordinateur.
- le type complexe (`complex`) : correspond à une variable dont la valeur est un nombre complexe, stocké comme un couple de `float` (donc en valeur approchée). Le nombre complexe `3+4i` est noté `3+4J` (on peut remplacer `J` par `j`).
- le type booléen (`bool`).

5.2 Les types séquentiels

Il faut distinguer les types modifiables (en anglais, mutables) des types non modifiables (en anglais, immutables).

- les listes (`list`) sont des objets séquentiels modifiables.
- les chaines de caractères (`str`) sont des objets séquentiels non modifiables. Les techniques de slicing s'appliquent aux chaines de caractères.
- les objets de la classe `tuple` sont des objets séquentiels voisins des listes mais qui sont non modifiables. Pour définir un tuple, on place souvent entre parenthèses ses éléments (même si elles sont facultatives).

Autres types :

- les dictionnaires (`dict`) sont des tableaux associatifs, c'est-à-dire des collections modifiables d'ojets indexés par des objets de n'importe quel type.
- les ensembles (`set`) sont des ensembles d'éléments uniques (l'ordre des éléments n'étant pas précisé par l'utilisateur, mais choisi par l'interpréteur).

5.3 Exemples

```
In []: x=1
In []: x,type(x)
Out[]: (1, <class 'int'>)
In []: x=2.4
In []: x,type(x)
Out[]: (2.4, <class 'float'>)
In []: x=2+4J
In []: x,type(x)
Out[]: ((2+4j), <class 'complex'>)
In []: x,y=1,2
In []: eq=x==y
In []: eq,type(eq)
Out[]: (False, <class 'bool'>)
In []: liste=['chat','chien',1,4]
In []: liste,type(liste)
Out[]: (['chat','chien',1,4], <class 'list'>)
In []: msg="Bonjour"
In []: msg,type(msg)
Out[]: ('Bonjour', <class 'str'>)
In []: x=1,2,3,4,5
In []: x,type(x)
Out[]: ((1, 2, 3, 4, 5), <class 'tuple'>)
In []: tel={'jack':4098,'john':4139}
In []: tel, type(tel)
```

```
Out[]: ({'john': 4139, 'jack': 4098}, <class 'dict'>)
In []: fruits={'pomme','banane','poire','pomme'}
In []: fruits,type(fruits)
Out[]: ({'banane', 'poire', 'pomme'}, <class 'set'>)
```

5.4 Le type int et float

Pour qu'une donnée numérique soit considérée par Python comme étant du type `float`, il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10. Les données

3.14 10. .001 1e100 3.14e-10

sont automatiquement interprétées par Python comme étant du type float.

```
a,b,c=3,2,1
while c<15:
    print(c,": ",b)
    a,b,c=b,a*b,c+1

a,b,c=1.,2.,1
while c<18:
    print(c,": ",b)
    a,b,c=b,a*b,c+1
```

5.5 Le type bool

Un booléen est un type de données qui ne peut prendre que deux valeurs : vrai ou faux. En Python, les constantes littérales sont notées `True` et `False`.

- `True` vaut 1.
- `False` vaut 0.

Tous les types de variables peuvent être interprétés de manière booléenne.

Par exemple, pour les entiers (`int`), la valeur 0 correspond à faux et les autres valeurs à vrai. Il en est de même pour tous les autres types : une valeur particulière vaut `False` et le reste des valeurs `True`. Le tableau suivant présente les valeurs `False` pour les principaux type de données.

Les valeurs `False` :

bool	int	float	str	tuple	list	dict
False	0	0.	""	()	[]	{}

5.6 Le type str

- Une donnée de type `str` peut se définir en première approximation comme une suite quelconque de caractères.
- On peut délimiter une telle suite de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (doubles quotes).
- On utilisera les guillemets pour délimiter une chaine dans laquelle il y a des apostrophes et des apostrophes pour délimiter une chaine qui contient des guillemets.

```
In []: phrase1='les mathématiques.'
In []: phrase2='"Oui", répondit-il,'
In []: phrase3="j'aime bien"
In []: print(phrase2,phrase3,phrase1)
Out[]: "Oui", répondit-il, j'aime bien les mathématiques.
```

Le caractère spécial \ (antislash) permet :

- d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande),

– d'insérer un certain nombre de codes spéciaux (saut à la ligne `\n`, apostrophes `\'`, guillemets `\"`).

Pour insérer plus aisément des caractères spéciaux dans une chaine, sans utiliser l'antislash, on peut encore délimiter la chaine à l'aide de triples guillemets ou triples apostrophes.

– Les chaines de caractères constituent un cas particulier de **données composites**. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble de données plus simples : dans le cas d'une chaine de caractères, ces entités plus simples sont les caractères eux-mêmes.

– Python considère qu'une chaine de caractères est un objet de la catégorie des **séquences**, lesquelles sont des **collections ordonnées d'éléments**. Chaque caractère de la chaine peut être désigné par sa place dans la séquence, à l'aide d'un **index**.

– Attention : les données informatiques sont presque toujours numérotées **à partir de zéro** (et non à partir de un). C'est le cas pour les caractères d'une chaine.

```
In []: ch='Mathématiques'
In []: print(ch[0],ch[4],ch[9])
Out[]: M é q
```

5.7 Opérations élémentaires sur les chaines

Python intègre de nombreuses **fonctions** qui permettent d'effectuer divers traitements sur les chaines de caractères. Par exemple, on peut

– accéder individuellement à chacun des caractères d'une chaine.
– assembler plusieurs chaines. L'opérateur de concaténation est `+`.
– déterminer la longueur (le nombre de caractères) d'une chaine, en faisant appel à la fonction `len()`.
– convertir en nombre véritable une chaine de caractères qui représente un nombre, en faisant appel à la fonction `int()` ou `f float()`.

```
In []: a="J'aime "
In []: b="Python !"
In []: c=a+b
In []: print(c)
Out[]: J'aime Python !

In []: ch='2'
In []: a=3
In []: b=int(ch)
In []: print(a+b)
Out[]: 5
```

5.8 Formatage des chaines de caractères

Cette opération se révèle parfaitement utile dans tous les cas où l'on doit construire une chaine de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses.

```
In []: coul='verte'
In []: temp=17.5
In []: ch="La couleur est {} et la température vaut {} °C"
In []: print(ch.format(coul,temp))
Out[]: La couleur est verte et la température vaut 17.5 °C
```

Les balises à utiliser sont constituées d'accolades, contenant ou non des indications de formatage :

– Si les balises sont vides, la méthode `format()` devra recevoir autant d'arguments qu'il y aura de balises dans la chaines. Python appliquera alors la fonction `str()` à chacun des arguments et les insérera ensuite dans la chaine à la place des balises, dans le même ordre.

Les balises peuvent contenir des numéros d'ordre (comptés à partir de zéro) pour désigner précisément lesquels des arguments transmis à `format()` devront les remplacer. Cette technique est efficace si le même argument doit remplacer plusieurs balises. Les balises peuvent aussi contenir des indications de formatage.

```
In []: x=1037.123456789
In []: '{:g}'.format(x) # choisit le format le plus approprié
Out[]: '1037.12'
In []: '{:.3f}'.format(x) # fixe le nombre de décimales
Out[]: '1037.123'
In []: '{:.3e}'.format(x) # notation scientifique
Out[]: '1.037e+03
```

5.9 Les listes

- Les listes sont également un type de données composites. Sous Python, on peut définir une liste comme **une collection d'éléments, séparés par des virgules, l'ensemble étant enfermé dans des crochets**.
- Les élements individuels d'une liste peuvent être de types variés.
- Comme les chaines, les listes sont des séquences, c'est-à-dire des collections ordonnées d'objets. Les divers éléments d'une liste sont toujours disposés dans le même ordre et on peut donc accéder à chacun d'entre eux individuellement si l'on connait son index dans la liste. Comme pour les chaines, la numérotation de ces index commence **à partir de zéro**.
- La fonction intégrée `len()` s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste.

A la différence de ce qui se passe pour les chaines, qui constituent un type de données non-modifiables, il est possible de changer les éléments individuels d'une liste.

```
In []: jour=['lundi','mardi',10,20.5,'jeudi','vendredi']
In []: jour[2]='mercredi'
In []: print(jour)
Out[]: ['lundi','mardi','mercredi',20.5,'jeudi','vendredi']
In []: del(jour[3])
In []: print(jour)
Out[]: ['lundi','mardi','mercredi','jeudi','vendredi']
In []: jour.append('samedi')
In []: print(jour)
Out[]: ['lundi','mardi','mercredi','jeudi','vendredi','samedi']
```

A la fin de cet exemple, nous avons appliqué **la méthode** `append()` à **l'objet jour**, avec **l'argument** `'samedi'`.

6 Fonctions

- Une fonction permet d'isoler une instruction qui revient plusieurs fois dans un programme.
- Une fonction est définie par un **nom**, par ses **arguments** qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel et éventuellement une **valeur de retour** communiquée au programme par la fonction en fin d'exécution.
- Utiliser des fonctions évite les répétitions dans les programmes et rend donc ces derniers plus courts et, surtout, plus faciles à lire et à comprendre.
- **Utiliser des fonctions permet d'organiser le travail de développement** : on peut décider d'écrire le programme principal un jour et d'écrire les fonctions un autre jour, on peut modifier le corps de la fonction sans modifier le programme principal.

6.1 Argument

- On appelle **argument** d'une fonction une variable particulière, utilisée dans le corps de la fonction, et dont la valeur est donnée dans le programme principal au moment où la fonction est appelée.
- On appelle **en-tête** d'une fonction la première ligne de sa définition, qui comporte dans l'ordre :
  1. le mot-clé `def` (en Python),
  2. le nom de la fonction,
  3. la liste de ses arguments entre parenthèses.

  L'ordre des arguments n'a pas d'importance pour la définition de la fonction. En revanche, lors d'un appel à cette fonction l'ordre des arguments doit être respecté.
- Souvent la seconde ligne de la fonction est une chaine de caractère servant à décrire la fonction.

6.2 Valeur de retour

- Le passage d'arguments permet donc de communiquer des informations depuis le programme principal vers une fonction. On veut aussi souvent communiquer des informations dans l'autre sens.
- La valeur produite par une fonction à partir des ses arguments, s'il en existe une, est appelée **valeur de retour**. Une fonction peut ne pas renvoyer de valeur, par exemple si elle ne fait qu'afficher des messages à l'écran.
- Une valeur de retour est souvent obtenue par l'instruction `return`.
- Il convient de prévoir une exécution correcte de la fonction quelle que soit la valeur donnée à chacun des arguments, y compris dans des cas que l'on n'a pas forcément anticipés dans le cours normal du programme principal.

6.3 Exemple : Fonction sans paramètre

```
def table7():
    n=1
    while n<11:
        print(n*7,end' ')
        n=n+1
table7() # renvoie 7 14 21 28 35 42 49 56 63 70
```

6.4 Exemple : Fonction avec paramètre(s)

```
def table(base):
    n=1
    while n<11:
        print(n*base,end=' ')
        n=n+1
table(7) # renvoie 7 14 21 28 35 42 49 56 63 70
```

```
def tableMulti(base,debut,fin):
    print("Extrait de la table de multiplication par ",base,' : ')
    n=debut
    while n<=fin:
        print(n,'x',base,'=',n*base)
        n=n+1
tableMulti(8,5,9)
```

6.5 Variables locales

Les variables définies à l'intérieur d'une fonction ne sont pas visibles depuis l'extérieur de la fonction ; elles correspondent aux variables muettes en mathématiques. Aussitôt l'exécution de la fonction terminée, l'interpréteur efface toute trace des variables internes à la fonction. On exprime cela en disant qu'une telle variable est **locale** à la fonction.

```
def f(x):
    y=1
    return x
##
In []: f(2)
Out[]: 2
In []: y # renvoie une erreur
In []: x # renvoie une erreur
```

```
x=0
def f(y):
    x=1
    return y
##
In []: f(2)
Out[]: 2
In []: x
Out[]: 0
```

6.6 Variables globales

Les variables définies à l'extérieur d'une fonction sont des **variables globales**. Leur contenu est visible de l'intérieur d'une fonction, mais la fonction **ne peut pas le modifier**.

```
def mask():
    p=20
    print(p,q)
p,q=15,38
##
In []: mask()
20 38
In []: print(p,q)
15 38
```

Il est utile que des variables soient ainsi définies comme étant locales, c'est-à-dire en quelque sorte confinées à l'intérieur d'une fonction. Cela signifie que l'on peut toujours utiliser une infinité de fonctions sans se préoccuper des noms de variables qui y sont utilisées : ces variables ne pourront jamais interférer avec celles que l'on aura définies par ailleurs.

On peut avoir à définir une fonction qui soit capable de modifier une variable globale. Pour cela, il suffit d'utiliser l'instruction `global`. Cette instruction permet d'indiquer, à l'intérieur de la définition d'une fonction, quelles sont les variables à traiter globalement.

```
def monter():
    global a
    a=a+1
    print(a)
a=15
##
In []: monter()
16
In []: print(a)
16
In []: monter()
17
```

6.7 Fonctions et procédures

- Certaines fonctions définies jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des **procédures**.
- Une vraie fonction doit **renvoyer une valeur** lorsqu'elle se termine.
- Une vraie fonction peut s'utiliser à la droite du signe égale dans des expressions telles que `y=sin(a)`.

```
def cube(w):
    return w*w*w
b=cube(9)
In []: b
Out[]: 729
```

- Dans certains langages de programmation, les fonctions et les procédures sont définies à l'aide d'instructions différentes. Python utilise la même instruction `def` pour définir les unes et les autres.

```
def table(base):
    resultat=[]
    n=1
    while n<11:
        b=n*base
        resultat.append(b)
        n=n+1
return resultat
ta9=table(9)
In []: ta9
Out[]: [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

L'instruction `return` définit ce que doit être la valeur renvoyée par la fonction.

6.8 return

```
def f(x):
    return x**2+2*x-3
In []: f(0),f(1),f(2)
Out[]: (-1,0,1)
```

```
def g(x):
    print(x**2+2*x-3)
In []: g(0),g(1),g(2)
-1
0
1
Out[]: (None,None,None)
```

Attention à ne pas confondre l'instruction `print` et l'instruction `return`. La plupart des fonctions sont des fonctions contenant dans leur définition l'instruction `return`.

Une propriété remarquable de la fonction `return` est qu'elle interrompt systématiquement l'exécution de la fonction. Dès que l'interpréteur atteint l'instruction `return qqchose`, il renvoie l'objet `qqchose` et abondonne aussitôt l'exécution de la fonction. Inutile donc de placer des instructions après un `return`.

6.9 Retour sur l'instruction resultat.append(b)

Nous appliquons la **méthode** `append()` à **l'objet** `resultat`.

Une **méthode** n'est rien d'autre qu'une fonction mais **une fonction qui est associée à un objet**. Elle fait partie de la définition de cet objet, ou plus précisément de la **classe** particulière à laquelle cet objet appartient.

Mettre en oeuvre une méthode associée à un objet consiste en quelque sorte à *faire fonctionner* cet objet d'une manière particulière. L'instruction du type `objet.methode()` : le nom de l'objet, le nom de la méthode, réliés par un point qui joue le rôle d'**opérateur**.

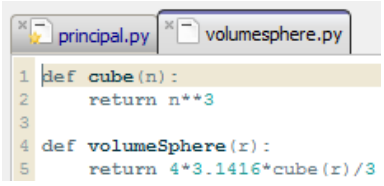
Sous Python, les **listes** constituent une **classe** particulière d'objets, auxquels on peut appliquer toute une série de **méthodes**. Ici, la méthode `append()` des objets listes sert à leur ajouter un élément à la fin ; l'élément à ajouter est transmis entre les parenthèses, comme tout argument.

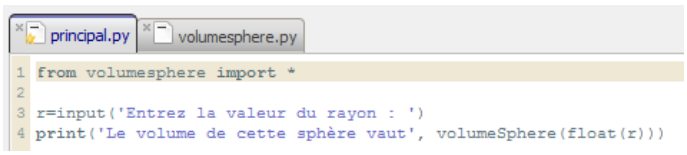
6.10 Utilisation des fonctions dans un script

```
def cube(n):
    return n**3

def volumeSphere(r):
    return 4*3.1416*cube(r)/3

r=input('Entrez la valeur du rayon : ')
print('Le volume de cette sphère vaut', volumeSphere(float(t)))
```

Ce programme comporte trois parties : les fonctions `cube()` et `volumeSphere()` et ensuite le corps principal du programme.
Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source.

Dans un script, la définition des fonctions doit précéder leur utilisation.
Fréquemment, on place les définitions de fonctions dans un **module** Python, et le programme qui les utilise dans un autre





On peut nommer les modules comme bon nous semble. Cependant, il est impossible d'utiliser un nom réservé par Python et il faut éviter de donner le nom d'un module déjà existant.

6.11 Typage des paramètres

Le **typage** des variables sous Python est un **typage dynamique**, ce qui signifie que le type d'une variable est défini au moment où on lui affecte une valeur. Ce mécanisme fonctionne aussi pour les paramètres d'une fonction. Le type d'un paramètre devient automatiquement le même que celui de l'argument qui a été transmis à la fonction.

```
def affiche3fois(arg):
    print(arg,arg,arg)
    print(type(arg))
##
In []: affiche3fois(5)
5 5 5
<class 'int'>
In []: affiche3fois('zut')
zut zut zut
<class 'str'>
```

6.12 Valeurs par défaut des paramètres

Dans la définition des fonctions, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction **qui peut être appelée avec une partie seulement des arguments attendus**.

```
def aire_rectangle(longueur,largeur=1):
    return longueur*largeur
##
In []: aire_rectangle(3)
Out[]: 3
In []: aire_rectangle(3,2)
Out[]: 6
```

On peut définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, les **paramètres sans valeur par défaut doivent précéder les autres** dans la liste.

6.13 Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis **exactement dans le même ordre** que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise une souplesse plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, on peut faire appel à la fonction en fournissant les arguments correspondants **dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants**.

```
def sommearithmetique(terme1=0,nbterme=10,raison=1):
    return terme1+(nbterme-1)*raison
##
In []: sommearithmetique()
Out[]: 9
In []: sommearithmetique(raison=2)
Out[]: 18
```

6.14 Compléments sur les arguments

Il est également possible d'utiliser des **paramètres implicites**, ce qui permet à l'utilisateur de fournir autant de paramètres qu'il souhaite. L'interpréteur place alors les valeurs des paramètres dans un dictionnaire.

```
def affiche(**arguments):
    for clé, valeur in arguments.items():
        print("arguments[{}]={}".format(clé,valeur),end=' ; ')
    print('\n arguments=',arguments)
affiche(arg1=[3,5],arg2='Hello',x=2)
```

Enfin, il possible d'utiliser des **paramètres arbitraires**. Ces paramètres sont semblables aux paramètres implicites : la différence tient au fait qu'ils ne sont pas nommés. L'interpréteur les place dans un tuple.

```
def produit(*facteurs):
    p=1
    for i in facteurs:
        p*=1
    print("Le produit des facteurs {} est {}.".format(facteurs,p))
##
In []: produit(2,3,4,5)
Le produit des facteurs (2, 3, 4, 5) est 120.
```

- Lorsqu'on combine des paramètres avec des valeurs par défaut, des paramètres implicites et arbitraires, on doit respecter l'ordre suivant :
  `def fonction(x,y,z,$\ast$args,$\ast\ast$keywords,a=1,b=2,c=3)`
- Rappelons que l'on peut munir une fonction d'une documentation appelée `docstring` : elle doit être placée à la ligne qui suit le mot-clé `def`, et être saisi entre guillements ou entre triples guillements s'il y a des retours à la ligne ou des caractères spéciaux.
- La première phrase se doit d'expliquer de manière concise à quoi sert la fonction.
- Pour accéder à la documentation de la fonction, on utilise `help`.

6.15 Paramètres modifiables

Les paramètres de types immuables et modifiables se comportent de manières différentes à l'intérieur d'une fonction. Ces paramètres sont manipulés dans le corps de la fonction, voire modifiés parfois. Selon le type du paramètre, ces modifications ont des répercussions à l'extérieur de la fonction.

Les types immuables ne peuvent être modifiés et cela reste vrai. Lorsqu'une fonction accepte un paramètre de type immuable, elle ne reçoit qu'une copie de sa valeur. Elle peut donc modifier ce paramètre sans que la variable ou la valeur utilisée lors de l'appel de la fonction n'en soit affectée. On appelle ceci un **passage de paramètre par valeur**.

A l'opposé, toute modification d'une variable d'un type modifiable à l'intérieur d'une fonction est répercutée à la variable qui a été utilisée lors de l'appel de cette fonction. On appelle ce second type de passage **un passage par adresse**.

```
def somme_n_premier_terme(n,liste):
    """calcul la somme des n premiers termes d'une liste"""
    somme = 0
    for i in liste:
        somme += i
        n -= 1 # modification de n (type immuable)
        if n <= 0: break
    liste[0] = 0 # modification de liste (type modifiable)
    return somme

l = [1,2,3,4]
nb = 3
print("avant la fonction ",nb,l) # affiche avant la fonction 3 [1, 2, 3, 4]
s = somme_n_premier_terme (nb,l)
print("après la fonction ",nb,l) # affiche après la fonction 3 [0, 2, 3, 4]
print("somme : ", s)             # affiche somme : 6
```

Dans l'exemple précédent, il faut faire distinguer le fait que la liste passée en paramètre ne soit que modifiée et non changée. L'exemple suivant inclut une fonction qui affecte une nouvelle valeur au paramètre liste sans pour autant modifier la liste envoyée en paramètre.

```
def fonction(liste):
    liste = []
##
In []: liste = [0,1,2]
In []: liste
Out[]: [0,1,2]
In []: fonction(liste)
In []: liste
Out[]: [0,1,2]
```

Il faut considérer dans ce programme que la fonction fonction reçoit un paramètre appelé liste mais utilise tout de suite cet identificateur pour l'associer à un contenu différent. L'identificateur liste est en quelque sorte passé du statut de paramètre à celui de variable locale. La fonction associe une valeur à liste - ici, une liste vide - sans toucher à la valeur que cet identificateur désignait précédemment.

Le programme qui suit est différent du précédent mais produit les mêmes effets. Ceci s'explique par le fait que le mot-clé `del` ne supprime pas le contenu d'une variable mais seulement son identificateur. Le langage Python détecte ensuite qu'un objet n'est plus désigné par aucun identificateur pour le supprimer.

```
def fonction(liste):
    del liste
##
In []: liste = [0,1,2]
In []: liste
Out[]: [0,1,2]
In []: fonction(liste)
In []: liste
Out[]: [0,1,2]
```

6.16 Fonctions locales

Il arrive que l'utilisation d'une fonction soit limitée à la définition d'une autre fonction.

```
def max3(x,y,z):
    def max2(u,v):
        if u>v:
            return u
        else:
            return v
    return max2(x,max2(y,z))
```

La fonction `max2` est une fonction locale à la fonction `max3`.

6.17 Fonctions comme valeurs de première classe (1)

En Python, une fonction est une valeur comme une autre, c'est-à-dire qu'elle peut être passée en argument, renvoyée comme résultat ou encore stockée dans une variable. On dit que les fonctions sont des **valeurs de première classe**.

```
def somme_fonction(f,n):
    s=0
    for i in range(n+1):
        s=s+f(i)
    return s

def carre(x):
    return x**x
##
somme_fonction(carre,10)
```

6.18 Fonctions anonymes

On peut même éviter de nommer la fonction `carre`, puisqu'elle est réduite à une simple expression, en utilisant une **fonction anonyme**. Une telle fonction s'écrit `lambda x: e` où `e` est une expression pouvant comporter la variable `x`. Elle désigne la fonction $x \mapsto e(x)$.

```
somme_fonction(lambda x: x*x,10)
```

6.19 Fonctions comme valeurs de première classe (2)

De même qu'on peut passer une fonction en argument, on peut renvoyer une fonction comme résultat.

```
def composee(f,g):
    def h(x):
        return f(g(x))
    return h

def composee(f,g):
    return lambda x:f(g(x))
```

6.20 Récursivité : 1er aperçu

On appelle **fonction récursive** une fonction qui comporte un appel à elle-même. Plus précisément, une fonction récursive doit respecter 3 points. Une fonction récursive :

- contient un cas de base.
- doit modifier son état pour se ramener au cas de base.
- doit s'appeler elle-même.

Une fonction mathématique définie par une relation de récurrence (et une condition initiale) peut de façon naturelle être programmée de manière récursive.

```
def puissance(x,n):
    if n==0:
        return 1
    else:
        return x*puissance(x,n-1)
##
In []: puissance(2,5)
Out[]: 32
```

6.21 Quelques primitives usuelles

On présente quelques primitives ou `built-in functions` les plus usuelles du langage. La liste complète et la description des primitives fournies par Python se trouve à la page :
http ://docs.python.org/py3k/library/functions.html

La primitive la plus utile pour connaitre les autres est la fonction `help`, qui permet d'accéder dans l'interpréteur à la documentation d'une fonction.

6.22 La fonction print()

- Elle permet d'afficher n'importe quel nombre de valeurs fournies en arguments.
- Par défaut, ces valeurs sont séparées les unes des autres par un espace, et le tout se terminera par un saut à la ligne.
- On peut remplacer le séparateur par défaut (l'espace) par un autre caractère, grâce à l'argument `sep`.
- On peut remplacer le saut à la ligne par l'argument `end`.

```
In []: print("Bonjour","à","tous",sep="***")
Bonjour***à***tous
In []: for i in range(6):
  ...:     print("hip",end="!")
  ...:
hip!hip!hip!hip!hip!hip!
```

6.23 La fonction input()

- Cette fonction provoque une interruption dans le programme courant.
- L'utilisateur est invité à entrer des caractères au clavier et à terminer avec `<Enter>`. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une **chaine de caractères** correspondant à ce que l'utilisateur a saisi. Cette chaine peut alors être assignée à une variable quelconque, convertie, etc.
- On peut invoquer la fonction `input()` en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif.
- Soulignons que la fonction `input()` renvoie toujours une chaine de caractères.
- Si l'on souhaite que l'utilisateur entre une valeur numérique, on devra donc convertir la valeur d'entrée (de type `str`) en une valeur de type numérique, par l'intermédiaire des fonctions intégrées `int()` ou `float()`.

```
In []: a=input("Entrez un nombre : ")
Entrez un nombre : 3
In []: type(a)
Out[]: <class 'str'>
In []: b=int(a)
In []: type(b)
Out[]: <class 'int'>
```

6.24 Primitives de conversion de type

Les primitives suivantes permettent de modifier le type d'un objet :
`dict, bin, int, str, bool, ord, float, complex, set, tuple, chr, list`

De plus, la primitive `isinstance` permet de tester l'appartenance d'un objet à un type.

7 Modules, fonctions prédéfinies

7.1 Importer un module de fonctions

On a déjà rencontré d'autres fonctions intégrées au langage lui-même, comme les fonctions `len()`, `id()`, `type()`, `range()`. Il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparées que l'on appelle des **modules**.

Les modules sont des fichiers qui regroupent des ensembles de fonctions. Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées, que l'on appelle des **bibliothèques**.

Il existe différentes façons importer un module

```
import module
module.fonction
```

ou

```
import module as alias
alias.fonction
```

ou encore

```
from module import fonction
fonction
```

ou enfin

```
from module import *
```

Le module `math`, par exemple, contient les définitions de nombreuses fonctions mathématiques. Pour pouvoir utiliser ces fonctions, il suffit d'incorporer la la ligne suivant au début du script :

`from math import *` ou `import math`

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant **toutes** les fonctions du module `math`, lequel contient une bibliothèque de fonctions mathématiques pré-programmées.

- `from math import *` : les nouvelles fonctions seront appelées par l'instruction `nom()`
- `import math` : les nouvelles fonctions seront appelées par l'instruction `math.nom()`

Après importation d'un module, on peut accéder à l'aide :

```
import math
help(math)
help(math.hypot)
```

7.2 Module math

Le module `math` permet d'importer les fonctions et constantes mathématiques usuelles.

Commandes Python :

- `pi`, `e`
- `exp(x)`, `log(x)`, `log(x,a)`, `pow(x,y)`
- `floor(x)`, `abs(x)`
- `factorial(n)`
- `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, ...
- `sinh(x)`, `cosh(x)`, `tanh(x)`, `asinh(x)`, ...

On dispose de deux syntaxes :

```
from math import *
cos(pi/4.0)
```

ou

```
import math
math.cos(math.pi/4.0)
```

7.3 Le module random

Le module `random` propose diverses fonctions permettant de générer des nombres (pseudo-)aléatoires qui suivent différentes distributions mathématiques.

`random.randrange(p,n,h)`	choisit un entier aléatoirement dans `range(p,n,h)`
`random.randint(a,b)`	choisit un entier aléatoirement dans $[\![a, b]\!]$
`random.choice(seq)`	choisit un entier aléatoirement dans `seq`
`random.random()`	renvoie un décimal aléatoire dans $[0, 1[$
`random.uniform(a,b)`	choisit un entier aléatoirement dans $[a, b]$

Exemple - Monte-Carlo :

Les méthodes de simulation de Monte-Carlo permettent de calculer une valeur numérique en utilisant des procédés aléatoires. Elles sont particulièrement utilisées pour calculer des intégrales en dimensions plus grandes que 1.

Par exemple, le calcul approché d'une intégrale d'une fonction à 1 variable par cette méthode repose sur l'approximation :

$$\int_a^b f(t)\,dt \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i)$$

où $i \mapsto x_i$ désigne une variable aléatoire suivant la loi uniforme sur l'intervalle $[a, b]$.

Ecrire une fonction renvoyant une approximation du nombre $\pi = 4\displaystyle\int_0^1 \sqrt{1-t^2}\,dt$ en utilisant la méthode de Monte-Carlo.

```
import random, math

def montecarlo(f, a, b, n):
    somme = 0
    for i in range(n):
        x = random.uniform(a, b)
        somme += f(x)
    return somme * (b-a) / n

def f(x):
    return math.sqrt(1 - x**2)

print('{0:>10s} | {1:^12s} | {2:^14s} | {3:^15s} |'.format('n',
        'approximation', 'erreur absolue', 'erreur relative'))

for i in range(0, 7, 2):
    n = 10**i
    approx = 4*montecarlo(f, 0, 1, n)
    erreur = math.pi - approx
    print('{0:10d} | {1: 12.10f} | {2: 14.10f} | {3: 15.10f} |'
            .format(n, approx, erreur, abs(erreur / math.pi)))
```

7.4 Le module turtle

`reset()`	efface tout
`goto(x,y)`	aller au point de coordonnées (x,y)
`forward(d)`, `backward(d)`	avancer, reculer d'une distance d
`up()`, `down()`	relever, abaisser le crayon
`left(alpha)`, `right(alpha)`	tourner à gauche, à droite de alpha (en degrés)
`color(couleur)`, `width(l)`	détermine la couleur, l'épaisseur du tracé
`fill(l)`	remplir un contour fermé

Exemple :

```
from turtle import *
a=0
while a<12:
    a=a+1
    forward(150)
    left(150)
```

7.5 Le module time

Du module `time`, nous utiliserons souvent la fonction `clock` qui renvoie un durée de temps en secondes et qui permet par soustraction de mesurer des durées d'exécution.

7.6 Le module sys

Le module `sys` contient des fonctions et des variables spécifiques au système, ou plus exactement à l'interpréteur lui-même.

Ce module est particulièrement intéressant pour récupérer les arguments passés à un script Python lorsque celui-ci est appelé en ligne de commande.

Par exemple `sys.path` est la liste des répertoires où l'interpréteur va chercher les différents modules lors de leur importation. On peut ainsi, si l'on importe un module personnel qui n'est pas dans le répertoire où se trouve le fichier à exécuter, modifier la variable `sys.path` :

```
import sys
sys.path.append('$/$home$/$user$/$programmation$/$python$/$mesmodules$/$')
```

7.7 Méthodes

Nous avons déjà employé couramment le mot **objet** sans pour autant le définir précisément. Cependant, nous avons dit qu'un objet possède toujours une valeur, un identifiant et un type. Certains objets Python possèdent en outre un certain nombre de fonctions qui ne s'appliquent qu'à un type donné d'objets : on parle alors de **méthode** associée à un objet.

Par exemple, on tapant `help(list)` ou `dir(list)`, on obtient les méthodes appliquables à l'objet liste.

La syntaxe est de la forme `objet.methode(arguments)`.

7.8 Méthodes associées aux listes

La méthode	Son effet
`list.append(x)`	ajoute l'élément x en fin de liste
`list.extend(L)`	ajoute en fin de liste les éléments de L
`list.insert(i,x)`	insére un élément x en position i
`list.remove(x)`	supprime la première occurence de x
`list.pop([i])`	supprime l'élément d'indice i et le renvoie
`list.index(x)`	renvoie l'indice de la première occurence de x
`list.count(x)`	renvoie le nombre d'occurences de x
`list.sort()`	modifie la liste en la triant
`list.reverse()`	modifie la liste en inversant l'ordre des éléments

7.9 Remarques sur les méthodes

En fait, **les méthodes sont des fonctions**, on pourrait donc les appeler comme des fonctions, par exemple `append(list,`x')' (mais cela ne marche pas !).

Mais, même si les méthodes sont des fonctions comme les autres, leur contexte est différent. Si vous tapez : `append` tout court, Python ne le reconnaîtra pas, le nom est inconnu ! En fait, `append` est défini dans le dictionnaire attaché à son sujet (et aux listes en général !). Même les fonctions dites standard sont souvent des méthodes en déguisement.

C'est l'argument (typiquement : gauche) d'un opérateur qui détermine quelle procédure appeler. Les méthodes sont là, afin de permettre à Python de prendre des décisions sélectives, et faire rapidement ce qu'il faut selon le type (on dit aussi classe) de l'argument.

Par exemple, pour calculer la longueur d'une chaîne ou d'une liste, vous appliquez la fonction `len`, mais c'est la méthode `__len__` spécifique qui se déclenche.

8 Les listes

```
In []: nombres=[5,38,10,25]
In []: nombres
Out[]: [5, 38, 10, 25]
In []: nombres[2]
Out[]: 10
In []: nombres[1:3]
Out[]: [38, 10]
In []: nombres[2:3]
Out[]: [10]
In []: nombres[2:]
Out[]: [10, 25]
In []: nombres[:2]
Out[]: [5, 38]
In []: nombres[-1]
Out[]: 25
In []: nombres[-2]
Out[]: 10
```

8.1 Les listes sont des objets modifiables

```
In []: nombres = [5,38,10,25]
In []: nombres[0]=17
In []: nombres
Out[]: [17, 38, 10, 25]
In []: nombres.sort()
In []: nombres
Out[]: [10, 17, 25, 38]
In []: nombres.append(12)
In []: nombres
Out[]: [10, 17, 25, 38, 12]
In []: sorted(nombres)
Out[]: [10, 12, 17, 25, 38]
In []: nombres
Out[]: [10, 17, 25, 38, 12]
In []: nombres.reverse()
In []: nombres
Out[]: [12, 38, 25, 17, 10]
In []: nombres.index(17)
Out[]: 3
In []: nombres.remove(38)
In []: nombres
Out[]: [12, 25, 17, 10]
In []: del nombres[2]
In []: nombres
Out[]: [12, 25, 10]
```

`del` travaille avec un index ou une tranche d'index, tandis que `remove()` recherche une valeur (si plusieurs éléments de la liste possèdent la même valeur, seul le premier est effacé).

8.2 Modification d'une liste

```
In []: mots=['jambon','fromage','confiture','chocolat']
In []: mots[2:2]=['miel']
In []: mots
Out[]: ['jambon','fromage','miel','confiture','chocolat']
In []: mots[5:5]=['saucisson','ketchup']
In []: mots
Out[]: ['jambon','fromage','miel','confiture','chocolat','saucisson','ketchup']
In []: mots[2:5]=[]
In []: mots
Out[]: ['jambon','fromage','saucisson','ketchup']
In []: mots[1:3]=['salade']
In []: mots
Out[]: ['jambon','salade','ketchup']
In []: mots[1:]
In []:
```

- Si on utilise l'opérateur `[]` à la gauche du signe égale pour effectuer une insertion ou une suppression d'élément(s), il faut obligatoirement y indiquer une tranche dans la liste cible (et non un élément isolé).
- L'élément fourni à la droite du signe égale doit lui-même être une liste.

8.3 La fonction range()

On peut créer très facilement une séquences de nombres à l'aide de la fonction intégrée `range()`. Elle renvoie une séquence d'entiers que l'on peut utiliser directement, ou convertir en une liste avec la fonction `list()`.

La fonction `range()` génère une séquence de nombres entiers.

```
In []: list(range(10))
Out[]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In []: list(range(5,13))
Out[]: [5, 6, 7, 8, 9, 10, 11, 12]
In []: list(range(3,16,3))
Out[]: [3, 6, 9, 12, 15]
In []: list(range(10,-10,-3))
Out[]: [10, 7, 4, 1, -2, -5, -8]
```

8.4 Parcours d'une liste

```
In []: phrase=['La','raison','du','plus','fort','est','toujours','la','meilleure']
In []: for mot in phrase:
  ...:     print(mot,end=' ')
  ...:
La raison du plus fort est toujours la meilleure

In []: for n in range(10,18,3):
  ...:     print(n,n**2,n**3)
  ...:
10 100 1000
13 169 2197
16 256 4096
```

Le type de la variable utilisée avec l'instruction `for` est redéfini continuellement au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de `for` sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture.

8.5 Liste en compréhension

Pour créer des listes, Python fournit une facilité syntaxique, à savoir **les listes définies par compréhension**. Elles permettent de générer des listes d'une manière très concise, sans avoir à utiliser de boucles.

La syntaxe pour définir une liste par compréhension est proche de celle utilisée en mathématiques pour définir un ensemble par compréhension.

$$\{f(x)\,|\,x \in E\} \rightarrow [f(x) \text{ for } x \text{ in liste}]$$

```
In []: liste=[2,4,6,8,10]
In []: [3*x for x in liste]
Out[]: [6, 12, 18, 24, 30]
In []: [3*x for x in liste if x>5]
Out[]: [18, 24, 30]
```

Les listes peuvent être formées en compréhension par `[ expr for indice in iterable]`, ou `[ expr for indice in` La liste est ici formée des valeus de `expr` quand `indice` parcourt `iterable` (et où `condition` est facultative).

Un itérable est tout objet qui peut être parcouru (les séquences, les ensembles, les dictionnaires, ...)

L'expression suivante forme par exemple la liste des x^2 où $1 \leq x < 100$, en se limitant à $x \equiv 3$ modulo 10.

```
[x*x for x in range(1,100) if x% 10==3]
```

En fait, les listes peuvent être construites en compréhension d'une façon plus générale encore. La syntaxe est alors

```
[ expression for indice1 in iterable1 if condition1 for indice2 in iterable2 if condition2 ... ]
```

Chacune des conditions est facultative (et la condition i s'applique à l'indice i). Dans cette syntaxe, *expression* est évaluée en fonction des valeurs des n-uplets (`indice1`, `indice2`, ...) successifs (dans l'ordre lexicographique) et le résultat est la liste de ces évaluations de `expression`.

Il faut bien comprendre la chronologie : pour chacune des valeurs possibles de `indice1`, on fait varier `indice2` (attention `iterable2` et `condition2` peuvent dépendre de `indice1`), puis (à `indice1` et `indice2` fixés) on fait varier `indice3`, etc.

Ainsi la boucle sur `indice1` contient la boucle sur `indice2`.

```
In []: [100*i+j for i in range(1,5) for j in range(1,4)]
Out[]: [101, 102, 103, 201, 202, 203, 301, 302, 303, 401, 402, 403]
In []: [100*i+j for i  in range(1,5) for j in range(1,i+1)]
Out[]: [101, 201, 202, 301, 302, 303, 401, 402, 403, 404]
In []: [100*i+j for j in range(1,i+1) for i in range(1,5)]
NameError: name 'i' is not defined
```

L'utilisation de listes en compréhension permet des constructions élégantes. L'imbrication des définitions en compréhension peut cependant conduire à des formulations assez délicates à relire.

8.6 Notion d'itérateur

En fait, la fonction `range()` est un itérateur. Les itérateurs sont des objets qui permettent de parcourir une séquence sans avoir à stocker la séquence complète en mémoire. Le principe est équivalent à un curseur de données placé sur la première donnée et qui découvre les éléments au fur et à mesure de l'avancée dans la séquence.

La fonction `enumerate` crée à partir d'une séquence un itérateur qui renvoie à chaque étape le tuple formé de l'élément de la séquence précédé de son indice de position.

```
In []: fable=['Maitre','Corbeau','sur','un','arbre','perché']
In []: for i,x in enumerate(fable):
  ...:     print(i,x)
  ...:
0 Maitre
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

La fonction `reversed` crée un itérateur inversé.

8.7 Parcours d'une liste : la fonction zip()

Lorsqu'on souhaire écrire une boucle for en parcourant les couples formés par les éléments de deux (ou plusieurs) séquences, on utilise la fonction `zip`.

```
In []: list(zip([1,2,3],['a','b','c']))
Out[]: [(1,'a'),(2,'b'),(3,'c')]
```

Lorsqu'on représente des vecteurs par des listes (ici u et v), il est alors très simple de les additionner composante par composante :

```
In []: u=[1,3,4]
In []: v=[2,-1,0]
In []: [x+y for x,y in zip(u,v)]
Out[]: [3,2,4]
```

8.8 Opérations sur les listes

Les objets de type séquentiel (dont les listes, les chaines et les tuples) ont en commun les opérations résumées dans le tableau suivant où s et t désignent deux séquences du même type et i,j,k des entiers.

L'opération	Son effet
`x in s`	`True` si s contient x, `False` sinon
`x not in s`	`True` si s ne contient pas x, `False` sinon
`s+t`	concaténation de s et t
`s*n`, `n*s`	n copies (superficielles) concaténées de s
`s[i]`	i-ième élément de s (à partir de 0)
`s[i:j]`	tranche de s de i (inclus) à j (exclu)
`s[i:j:k]`	tranche de s de i (inclus) à j (exclu) avec un pas de k
`len(s)`	longueur de s
`max(s)`, `min(s)`	plus grand, plus petit élément de s
`s.index(i)`	indice de la 1ère occurence de i dans s
`s.count(i)`	nombre d'occurences de i dans s

8.9 Copie d'une liste

```
In []: liste=[1,2,3]
In []: copie=liste
In []: liste,copie
Out[]: ([1, 2, 3], [1, 2, 3])
In []: liste[0]=5
In []: liste,copie
Out[]: ([5, 2, 3], [5, 2, 3])
In []: copie[1]=4
In []: liste,copie
Out[]: ([5, 4, 3], [5, 4, 3])
```

Les noms `liste` et `copie` désignent tous deux **un seul et même objet** en mémoire. On dit que le nom `copie` est un **alias** du nom `liste`.

Copie superficielle et copie profonde

Comment copier une liste en obtenant un objet distinct de la liste copiée ?

```
copie=liste[:]
```

ou

```
copie=list(liste)
```

Python effectue une copie superficielle de liste. En effet, ce ne sont pas les objets eux-mêmes qui sont copiés, mais les pointeurs vers les objets copiés. D'où le nom de **copie de références** donnée à ce type de copie.

```
In []: liste=[1,2,3]
In []: copie=list(liste)
In []: id(liste)==id(copie)
Out[]: False
In []: id(liste[0])==id(copie[0])
Out[]: True
In []: liste=['a',[1,2]]
In []: copie=list(liste)
In []: liste[0]='b'
In []: liste,copie
Out[]: (['b', [1, 2]], ['a', [1, 2]])
In []: liste[1][0]=2
In []: liste,copie
Out[]: (['b', [2, 2]], ['a', [2, 2]])
```

Pour effectuer une copie vraiment indépendante d'une liste (ou d'un objet en général), on peut utiliser la fonction `deepcopy` du module `copy` : cette fonction permet de copier récursivement tous les objets d'une liste

8.10 Tris : 1er aperçu

Pour une liste, on peut utiliser la méthode `sort`. Cela modifie la liste en triant ses éléments (avec l'utilisation de la fonction qui compare les éléments de la liste si ils sont comparables) :

```
In []: liste=[3,8,5,19,45,34,2]
In []: id(liste)
Out[]: 58918152
In []: liste.sort()
In []: liste
Out[]: [2, 3, 5, 8, 19, 34, 45]
In []: id(liste)
Out[]: 58918152
```

On obtient bien le même objet, avec ses éléments triés.

On peut, à la place créer une nouvelle liste triée avec la fonction `sorted`.

```
In []: liste=[3,8,5,19,45,34,2]
Out[]: 58419960
In []: l2=sorted(liste)
In []: id(l2)
Out[]: 5894984
In []: l2
Out[]: [2, 3, 5, 8, 19, 34, 45]
In []: liste
Out[]: [3, 8, 5, 19, 45, 34, 2]
```

On peut également trier un dictionnaire. Il n'y a bien entendu pas de méthode de tri pour un dictionnaire (c'est non ordonné), mais la fonction `sorted` peut s'utiliser. Le tri est alors réalisé sur les clés.

```
In []: personnes={"Pierre":['H',45],"Paule":['F',53],"Jacques":['H',57],"Marie":['F',43]}
In []: sorted(personnes)
Out[]: ['Jacques','Marie','Paule','Pierre']
```

8.11 Quelques méthodes pour trier une liste

- **Tri par sélection** : on commence par rechercher le plus grand élément de la liste que l'on va mettre à sa place, c'est-à-dire l'échanger avec le dernier élément. Puis on recommence avec le deuxième élément le plus grand, etc.
- **Tri par insertion** : on commence par prendre le premier élément à trier que l'on place en position 1. Puis, on insère les éléments dans l'ordre en plaçant chaque nouvel élément à sa bonne place. Pour procéder à un tri par insertion, il suffit de parcourir une liste : on prend les éléments dans l'ordre, ensuite, on les compare avec les éléments précédents jusqu'à trouver la place de l'élément qu'on considère. Il ne reste plus qu'à décaler les éléments de la liste pour insérer l'élément considéré à sa place dans la partie déjà triée.
- **Tri rapide** : L'idée est de prendre un élément au hasard (par exemple le premier) que l'on appelle le pivot et de le mettre à sa place définitive en plaçant tous les éléments qui sont plus petits à sa gauche et tous ceux qui sont plus grands à sa droite. On recommence ensuite le tri sur les deux sous-listes obtenues jusqu'à ce que la liste soit triée.

9 Les chaines de caractères

Les chaines de caractères comme les listes sont des séquences.

L'opération	Son effet
`x in s`	`True` si s contient x, `False` sinon
`x not in s`	`True` si s ne contient pas x, `False` sinon
`s+t`	concaténation de s et t
`s*n`, `n*s`	n copies (superficielles) concaténées de s
`s[i]`	i-ième élément de s (à partir de 0)
`s[i:j]`	tranche de s de i (inclus) à j (exclu)
`s[i:j:k]`	tranche de s de i (inclus) à j (exclu) avec un pas de k
`len(s)`	longueur de s
`max(s)`, `min(s)`	plus grand, plus petit élément de s
`s.index(i)`	indice de la 1ère occurence de i dans s
`s.count(i)`	nombre d'occurences de i dans s

Sous Python 3, les chaines de caractères (`str`) sont des chaines Unicode. Cela signifie que les identifiants numériques de leurs caractères sont uniques (il ne peut exister qu'un seul caractère typographique pour chaque code) et universels (les identifiants choisis couvrent la gamme complète de tous les caractères utilisés dans les différentes langues du monde entier).

9.1 Parcours d'une séquence : for ... in ...

Le **parcours d'une séquence** est une opération très fréquente en programmation. Python propose une structure de boucle appropriée basée sur le couple d'instructions `for ... in ...`.

L'instruction `for` permet d'écrire des boucles, dans lesquelles **l'itération traite successivement tous les éléments d'une séquence donnée**.

```
In []: nom="Cléopâtre"
In []: for car in nom:
  ...:     print(car+'*',end=' ')
  ...:
C* l* é* p* â* t* r* e*
```

9.2 Les chaines sont non modifiables et comparables

On ne peut pas modifier le contenu d'une chaine existante. On ne peut donc pas utiliser l'opérateur `[]` dans la partie gauche d'une instruction d'affectation.

```
In []: salut="bonjour"
In []: salut[0]='B' #Renvoie une erreur

In []: salut="bonjour"
In []: salut='B'+salut[1:]
In []: salut
Out[]: Bonjour
```

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux (ie les instructions `if ... elif ... else`) fonctionnent avec les chaines de caractères.

Cela peut être utilse pour trier des mots par ordre alphabétique. Cela ne fonctionne cependant bien que pour des mots qui sont tous entièrement en minuscules (ou entièrement en majuscules) et qui ne comportent aucun caractère accentué.

```
In []: "oui"<"non"
Out[]: False
In []: "été"<"hiver"
Out[]: False
```

9.3 Méthodes applicables sur les chaines

Quelques méthodes applicables :
- `split()` : convertit une chaine en une liste de sous-chaines.
- `join(liste)` : rassemble une liste de chaines en une seule. Attention : la chaine à laquelle on applique cette méthode est celle qui servira de séparateur, l'argument transmis est la liste des chaines à rassembler.
- `find(sch)` : cherche la position d'une sous-chaine `sch` dans la chaine.
- `count(sch)` : compte le nombre de sous-chaines `sch` dans la chaine.
- `replace(c1,c2)` : remplace tous les caractères `c1` par des caractères `c2` dans la chaine.
- `index(car)` : retrouve l'indice de la première occurence du caractère `car` dans la chaine.
- `lower()` : convertit une chaine en minuscules.
- `upper()` : convertit une chaine en majuscules.
- `strip()` : enlève les espaces éventuels au début et à la fin de la chaine.

9.4 La méthode split

La méthode `split` permet de découper une chaine en une liste de sous-chaines, en effectuant les coupures sur des caractères bien précis (par défaut les espaces).

```
In []: t="être ou ne pas être, là est la question"
In []: t.split()
Out[]: ['être', 'ou', 'ne', 'pas', 'être,', 'là', 'est', 'la', 'question']
In []: t.split('être') #On choisit une chaîne comme séparateur
Out[]: ['',' ou ne pas ',',là est la question']
In []: t="  plein de    vide   dans cette    chaîne  "
In []: t.split() #dans un .split() tous les espaces sont supprimés.
Out[]: ['plein', 'de', 'vide', 'dans', 'cette', 'chaîne']
In []: t="--plein-de--tirets---dans-cette--chaîne--"
In []: t.split('-')
Out[]: ['','','plein', 'de', '', 'tirets', '', '', 'dans', 'cette', '', 'chaîne', '', '']
```

9.5 La méthode join

La méthode `join` est l'inverse de la méthode `split`. L'expression `sep.join(iterable)` regroupe les chaines éléments de l'objet `iterable` (une liste de chaines, par exemple), en utilisant la chaine `sep` comme séparateur.

```
In []: ''.join(['abc','def','ghi'])
Out[]: 'abcdefghi'
In []: '+'.join(['abc','def','ghi'])
Out[]: 'abc+def+ghi'
```

La méthode `join` doit être considérée comme une propriété de la chaine qui est spécifiée comme séparateur.

La méthode `join` est économique en mémoire et doit être préférée à une boucle mettant les différentes chaines bout à bout (complexité linéaire plutôt que quadratique).

9.6 La méthode format

La méthode `format` permet de formater une chaine (pour l'aficher, le plus souvent), en utilisant un canevas contenant des champs (qui servent à spécifier le format) et des arguments pour renseigner ces champs.

La syntaxe est `canevas.format(arguments)`. Les champs de formatage (dans la chaine `canevas`) sont délimités par des accolades, et chaque champ est renseigné par l'argument qui lui correspond. Il y a énormément de variantes.

```
In []: canevas = 'Nom: {}, prénom: {}, date de naissance: {}'
In []: canevas.format('Shakespeare','Willian',1613)
Out[]: 'Nom: Shakespeare, prénom: William, date de naissance: 1613'
In []: canevas = 'Nom: {n}, prénom: {p}, date de naissance: {d}'
In []: canevas.format(d=1613,n='Shakespeare',p='Willian')
Out[]: 'Nom: Shakespeare, prénom: William, date de naissance: 1613'
In []: canevas='Le nombre décimal {0:d} s'écrit {0:b} en binaire et {0:x} en hexadécimal'
In []: canevas.format(2013)
Out[]: "Le nombre décimal 2013 s'écrit 11111011101 en binaire et 7dd en hexadécimal"
```

10 Autres types

10.1 Les tuples

- Les chaines sont composées de caractères et sont des séquences non modifiables.
- Les listes sont composées d'éléments de n'importe quel type et sont des séquences modifiables.
- Les tuples sont composées d'éléments de n'importe quel type mais sont des séquences non modifiables.

```
In []: tup1='a',1,[1,2]
In []: tup2=('a',1,[1,2])
In []: tup1==tup2
Out[]: True # par contre, tup1 et tup2 n'ont pas le même id
```

Les opérations que l'on peut effectuer sur les tuples sont syntaxiquement similaires à celles que l'on peut effectuer sur les listes, si ce n'est pas les tuples ne sont pas modifiables.

Les tuples sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les tuples sont moins gourmands en ressources système.

10.2 Les dictionnaires

Les dictionnaires sont de type **composite**. Ils ressemblent aux listes (ils sont modifiables), mais ce ne sont pas des séquences. Les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appelera une **clé**, laquelle pourra être alphabétique, numérique, etc.

Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des chaines, des listes, des tuples, des dictionnaires, etc.

```
In []: dico={}
In []: dico['computer']='ordinateur'
In []: dico['mouse']='souris'
In []: dico['keyboard']='clavier'
In []: dico
Out[]: {'keyboard': 'clavier', 'computer': 'ordinateur', 'mouse': 'souris'}
```

Dans un dictionnaire, les index s'appellent des clés, et les éléments s'appellent des paires clé-valeur.

Contrairement aux listes, pour ajouter de nouveaux éléments à un dictionnaire, il suffit de créer une nouvelle paire clé-valeur.

Opérations :
- `del dico[clé]` : supprime la paire clé-valeur de dico.
- `len(dico)` : donne le nombre de clés de dico.
- `if clé in dico:` permet de avoir si `dico` comprend une clé bien déterminée.

Méthodes spécifiques :
- `keys()` : renvoie la séquence des clés utilisées dans le dictionnaire. Cette séquence peut être utilisée telle quelle ou être convertie en liste ou en tuple avec les fonctions `list()` ou `tuple()`.
- `values()` : renvoie la séquences des valeurs mémorisées dans le dictionnaire.
- `items()` : extrait du dictionnaire une séquence équivalente de tuples.
- `copy()` : effectue une vraie copie d'un dictionnaire.

On peut utiliser une boucle `for` pour traiter successivement tous les éléments contenus dans un dictionnaire :
- au cours de l'itération, ce sont les clés utilisées qui seront successivement affectées à la variable de travail, et non les valeurs.
- l'ordre dans lequel les éléments seront extraits est imprévisible.

Si l'on souhaite effectuer un traitement sur les valeurs, il suffit de récupérer chacune d'elles à partir de la clé correspondante :

```
In []: dico={'keyboard': 'clavier', 'computer': 'ordinateur', 'mouse': 'souris'}
In []: for cle in dico:
  ...:     print(cle,':',dico[cle])
  ...:
computer : ordinateur
keyboard : clavier
mouse : souris
In []: for cle,valeur in dico.items:
  ...:     print(cle,':',valeur)
  ...:
computer : ordinateur
keyboard : clavier
mouse : souris
```

- Les dictionnaires ne sont pas des séquences : les éléments d'un dictionnaire ne sont pas disposés dans un ordre particulier. Des opérations comme la concaténation et l'extraction ne peuvent pas s'appliquer.

– Les dictionnaires sont utiles pour gérer des ensembles de données où l'on est amené à effectuer fréquemment des ajouts ou des suppressions, dans n'importe quel ordre.
– Les dictionnaires remplacent avantageusement les listes lorsqu'il s'agit de traiter des ensembles de données numérotées, dont les numéros ne se suivent pas.

10.3 Les ensembles

Les objets de type ensemble (`set`) sont des structures de données qui modélisent la notion mathématique d'ensemble. Un tel objet peut contenir des valeurs de type quelconque, mais une même valeur ne peut apparaître qu'une seule fois. De plus les éléments d'un objet de type ensemble ne sont pas ordonnés (on ne peut pas y accéder par un indice : on peut juste savoir si une valeur est ou n'est pas élément de l'ensemble).

On forme un ensemble par la séquence de ses éléments (valeurs), encadrée par { et }, ou en utilisant le constructeur `set` (appliqué à un itérable quelconque).

```
In []: e={1,5,2,3,2,7,5,2,1,3,2}
In []: e=set([1,5,2,3,2,7,5,2,1,3,2]) #même résultat en convertissant une liste
In []: e
Out[]: {1, 2, 3, 5, 7} #Tous les doublons ont été éliminés
In []: set('abracadabra')
Out[]: {'d', 'r', 'a', 'c', 'b'}
```

11 Tracé de courbes avec Matplotlib

Le tracé de courbes scientifiques peut se fait à l'aide du module matplotlib. Pour l'utiliser, il faut importer le module pylab. La référence complète de matplotlib est lisible à l'adresse : http ://matplotlib.sourceforge.net/matplotlib.pyl

Il est en particulier recommandé de regarder les captures d'écrans, qui sont donnés avec le code utilisé pour les générer. Dans un premier temps, on ne verra que la représentation de fonction 1D.

Pour tracer le graphe d'une fonction $f : [a, b] \to \mathbb{R}$, Python a besoin d'une grille de points x_i où évaluer la fonction, ensuite il relie entre eux les points $(x_i, f(x_i))$ par des segments. Plus les points sont nombreux, plus le graphe de la fonction affine par morceaux est proche du graphe de la fonction f.

Pour générer les points x_i on peut utiliser l'instruction **linspace(a,b,n+1)** qui construit la liste de $n + 1$ éléments :

$$\left[a, a + \frac{b-a}{n}, a + 2\frac{b-a}{n}, \ldots, b\right]$$

ou l'instruction **arange(a,b,h)** qui construit la liste des éléments suivants :

$$[a, a + h, a + 2h, \ldots, a + nh]$$

avec n le plus grand entier tel que $a + nh < b$.

Voici un exemple avec une sinusoïde :

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-5,5,101) # x = [-5,-4.9,-4.8,...,5]
y = np.sin(x)
plt.plot(x,y)
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-5,5,0.1) # x = [-5,-4.9,-4.8,...,4.9]
y = np.sin(x)
plt.plot(x,y)
plt.show()
```

On obtient une courbe sur laquelle on peut zoomer, modifier les marges et sauvergarder dans différents formats (jph, png, eps,...).

Tout d'abord, vous voyez qu'on importe plutôt le module `pylab` (et non pas `matplotlib`). Le module `pylab` importe lui-même toutes les fonctions (et variables) du module `numpy` (e.g. `pi, cos, arange`, etc.).
Avec le module `pyplot`, il faut importer le module `numpy` si l'on souhaite faire du calcul.

On pourra choisir d'importer par

```
from pylab import *
# ou
from matplotlib.pylab import *
# ou
import matplotlib.pyplot as plt
import numpy as np
```

Les commandes sont très intuitives.

– La fonction `plot()` va générer un graphique avec des lignes et prend comme valeurs en abscisse (x) et en ordonnées (y) des vecteurs de type `array` à une dimension ou des listes de valeurs.

– Les fonctions `xlabel()` et `ylabel()` sont utiles pour donner un nom aux axes.
– `title()` permet de définir le titre du graphique.
– `grid()` affiche une grille en filligrane.
– Pour activer l'affichage à l'écran du graphique, il faut appeler la fonction `show()`. Celle-ci va activer une boucle dite gtk qui attendra les manipulations de l'utilisateur.
– Les commandes Python éventuellement situées après la fonction `show()` seront exécutées seulement lorsque l'utilisateur fermera la fenêtre graphique.

On peut même tracer plusieurs courbes sur la même figure. Par exemple, si on veut comparer les graphes de la fonction précédente en modifiant la grille de départ, on peut écrire :

```
import matplotlib.pyplot as plt
import numpy as np
a = np.linspace(-5,5,5) # a = [-5,-3,-1,1,3,5]
fa = np.sin(a)
b = np.linspace(-5,5,10) # a = [-5,-4,-3,...,5]
fb = np.sin(b)
c = np.linspace(-5,5,101) # b = [-5,-4.9,-4.8,...,5]
fc = np.sin(c)
plt.plot(a,fa,b,fb,c,fc)
plt.show()
```

Pour tracer plusieurs courbes, on peut les mettre les unes à la suite des autres en spécifiant la couleur et le type de trait, changer les étiquettes des axes, donner un titre, ajouter une grille, une légende...

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-5,5,101) # x = [-5,-4.9,-4.8,...,5]
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x,y1,"r-",x,y2,"g.")
plt.legend(['sinus','cosinus'])
plt.xlabel('abscisses')
plt.ylabel('ordonnees')
plt.title('Comparaison de sin(x) et cos(x)')
plt.grid(True)
plt.show()
```

"r-" indique que la première courbe est à tracer en rouge avec un trait continu, et "g." que la deuxième est à tracer en vert avec des points.

`-`	solid line	`--`	dashed line
`-.`	dash-dot line	`:`	dotted line
`.`	points	`,`	pixels
`o`	circle symbols	`^`	triangle up symbols
`v`	triangle down symbols	`<`	triangle left symbols
`>`	triangle right symbols	`s`	square symbols
`+`	plus symbols	`x`	cross symbols
`D`	diamond symbols		
`b`	blue	`g`	green
`r`	red	`c`	cyan
`m`	magenta	`y`	yellow
`k`	black	`w`	white

On peut déplacer la légende en spécifiant l'une des valeurs suivantes :
`best`, `upper right`, `upper left`, `lower right`, `lower left`, `center right`, `center left`, `lower center`, `upper center`, `center`

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-pi,pi,0.05*pi)
plt.plot(x,np.sin(x),'co',x,np.cos(x),'mD')
plt.legend(['sinus','cosinus'],loc='upper left')
plt.axis([-pi, pi, -1, 1]) # axis([xmin, xmax, ymin, ymax])
plt.show()
```

12 Matrices et tableaux avec Numpy

Le module `numpy` permet d'effectuer des calculs sur des vecteurs ou des matrices, élément par élément, via un nouveau type d'objet appelé `array`. Ce module contient des fonctions de base pour faire de l'algèbre linéaire, des transformées de Fourier ou encore des tirages de nombre aléatoire plus sophistiqués qu'avec le module `random`.

Le module `scipy` est lui même basé sur `numpy`, mais il en étend considérablement les possibilités de ce dernier (e.g. statistiques, optimisation, intégration numérique, traitement du signal, traitement d'image, algorithmes génétiques, etc.).

On trouve dans le module `numpy` les outils de manipulation des tableaux pour le calcul numérique (nombreuses fonctions de manipulation de tableaux et bibliothèque mathématique importante).

Pour l'importer, on recommande d'utiliser `import numpy as np`. Toutes les fonctions NumPy seront alors préfixées par np.

Les objets de type array correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction `array()` permet la conversion d'un objet séquentiel (type liste ou tuple) en un objet de type array. Voici un exemple simple de conversion d'une liste à une dimension en objet array :

```
In[] : import numpy as np
In[] : a=np.array([1,2,3])
In[] : type(a)
Out[]: <class 'numpy.ndarray'>
In[] : a
Out[]: array([1, 2, 3])
```

Nous avons converti la liste a en array, mais cela aurait donné le même résultat si on avait converti le tuple `(1,2,3)`.

Par ailleurs, vous voyez que lorsqu'on demande à Python le contenu d'un objet array, les symboles ([]) sont utilisés pour le distinguer d'une liste [] ou d'un tuple ().

Notez qu'un objet array ne peut contenir que des valeurs numériques. Vous ne pouvez pas, par exemple, convertir une liste contenant des chaînes de caractères en objet de type array.

La fonction `arange()` est équivalente à `range()` et permet de construire un array à une dimension de manière simple.

```
In[] : np.arange(8)
Out[]: array([0, 1, 2, 3, 4, 5, 6, 7])
In[] : np.arange(2,6)
Out[]: array([2, 3, 4, 5])
```

```
In []: np.arange(10,5,-1)
Out[]: array([10, 9, 8, 7, 6])
```

Un des avantages de la fonction `arange()` est qu'elle permet de générer des objets array qui contiennent des entiers ou de réels selon l'argument qu'on lui passe.

La différence fondamentale entre un objet array à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un **vecteur**. Par conséquent on peut effectuer des opérations **élément par élément** dessus, ce qui est bien commode lorsqu'on analyse de grandes quantités de données.
Regardez ces exemples :

```
In[] : v=np.arange(4)
In []: v
Out[]: array([0, 1, 2, 3])
In[] : v+1
Out[]: array([1, 2, 3, 4])
In[] : v*2
Out[]: array([0, 2, 4, 6])
In[] : v*v
Out[]: array([0, 1, 4, 9])
```

Notez bien sur le dernier exemple de multiplication que l'array final correspond à la multiplication élément par élément des deux array initiaux. Avec les listes, ces opérations n'auraient été possibles qu'avec des boucles !

Nous vous encourageons donc à utiliser dorénavant les objets array lorsque vous aurez besoin de faire des opérations élément par élément.

Il est aussi possible de construire des objets array à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```
In[] : np.array([[1,2,3],[2,3,4],[3,4,5]])
Out[]:
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

On peut aussi créer des tableaux à trois dimensions en passant à la fonction `array()` une liste de listes de listes. La fonction `array()` peut créer des tableaux à n'importe quel nombre de dimensions. Toutefois ça devient vite compliqué lorsqu'on dépasse trois dimensions.

Retenez qu'un objet array à une dimension peut être considéré comme un vecteur et un array à deux dimensions comme une matrice.

Pour récupérer un ou plusieurs élément(s) d'un objet array, vous pouvez utiliser l'indiçage ou les tranchage, de la même manière que pour les listes.

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

Dans le cas d'un objet array à deux dimensions, vous pouvez récupérer une ligne, une colonne ou bien un seul élément.

```
In[] : a=np.array([[1,2],[3,4]])
In[] : a
Out[]:
array([[1,2],
       [3,4]])
In[] : a[:,0]
Out[]: array([1,3])
In[] : a[0,:]
Out[]: array([1,2])
In[] : a[1,1]
Out[]: 4
```

La syntaxe $a[m,:]$ récupère la ligne m-1, et $a[:,n]$ récupère la colonne n-1.

Le module numpy contient quelques fonctions commodes pour construire des matrices à partir de rien. Les fonctions `zeros()` et `ones()` permettent de construire des objets array contenant des 0 ou de 1, respectivement. Il suffit de leur passer un tuple indiquant la dimensionnalité voulue.

```
In[] : np.zeros((3,3))
Out[]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
In[] : np.zeros((3,3),int)
Out[] :
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
In[] : np.ones((3,3))
Out[]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Attention, une opération de découpage crée une **vue** de l'array initial, qui est juste une façon d'accéder aux données du tableau. Ainsi, le tableau original n'est pas copié dans la mémoire. **Lorsqu'on modifie une vue, le tableau initial est modifié également**.

Pour extraire des sous-parties d'un tableau numpy, on a vu qu'on peut faire de l'indexation simple $t[0]$ et des coupes $t[1:3]$. Une autre possibilité très pratique est de sélectionner certaines valeurs d'un tableau grâce à un autre tableau de booléens (un "masque"), de taille compatible avec le tableau d'intérêt. Cette opération s'appelle de l'indexation par masque (ou fancy indexing). **Cela crée des copies (superficielles) pas des vues**.

```
In[] : a=np.array([10,3,8,0,7,6])
In[] : (a%3==0)
Out[]: array([False,True,False,True,False,True])
In[] : a[a%3==0]
Out[]: array([3,0,6])
In[] : a[[2,1,4,2]]
Out[]: array([8,3,7,8])
In[] : a[a>5]
Out[]: array([10,8,7,6])
```

Application : calcul de pi

```
x,y=np.random.random((2,100000))
mask=x**2+y**2<1 #quart de disque
pi=4*np.mean(mask) #fraction des points dans le disque (*4)
print(pi)
print(np.pi)
```

Attention à la copie de tableau ! Pour un tableau NumPy, par defaut on ne copie que l'adresse du tableau (pointeur) pas son contenu (les deux noms correspondent alors aux mêmes adresses en mémoire).
Pour effectuer une copie des valeurs, il faut utiliser .copy()
Remarque : la même chose s'applique aux coupes.

Il existe les fonctions `reshape()` et `resize()` qui permettent de remanier à volonté les dimensions d'un array. Il faut pour cela, leur passer en argument l'objet array à remanier ainsi qu'un tuple indiquant la nouvelle dimensionnalité.

```
In[] : a=np.arange(9)
In[] : a
Out[]:array([0, 1, 2, 3, 4, 5, 6, 7, 8])
In[] : np.reshape(a,(3,3))
Out[]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Toutefois, `reshape()` attend un tuple dont la dimension est compatible avec le nombre d'éléments contenus dans l'array de départ, alors que `resize()` s'en moque et remplira le nouvel objet array généré même si les longueurs ne coincident pas.

Si vous ne vous souvenez plus de la dimension d'un objet array, la fonction `shape()` permet d'en retrouver la taille.

La fonction `transpose()` renvoie la transposée d'un array. Il y a deux syntaxes possibles : `np.transpose(A)` ou `A.T`. Attention la transposée est une vue. Par exemple pour une matrice :

```
In[] : b=np.array([[0,1,2],[3,4,5],[6,7,8]])
In[] : np.transpose(b)
Out[]:
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
```

La fonction `trace()` renvoie la trace d'une matrice.

```
In[] : a=np.array([[1,2,3],[4,5,6],[7,8,9]])
In[] : b=2*a # Multiplication de chaque terme
In[] : c=a+b # Sommation terme a terme (addition de matrices)
In[] : np.dot(a,b) # Produit de matrices
Out[]:
array([ 60,  72,  84],
      [132, 162, 192],
      [204, 252, 300]])
In[] : a*b # Produit terme a terme
Out[]:
array([[  2,   8,  18],
       [ 32,  50,  72],
       [ 98, 128, 162]])
```

Notez bien que `dot(a,b)` renvoie le produit matriciel entre deux matrices, alors que `a*b` renvoie le produit élément par élément.

Pour toutes les opérations suivantes, il faudra utiliser des fonctions dans le sous-module numpy.linalg. La fonction `inv()` renvoie l'inverse d'une matrice carrée, `det()` son déterminant.

```
In[] : a=np.array([[0,1],[2,3]])
In[] : np.linalg.inv(a)
Out[]:
array([[-1.5, 0.5],
       [ 1. , 0. ]])
In[] : numpy.linalg.det(a)
Out[]: -2.0
```

La fonction `matrix__power(A,n)` renvoie A^n.

12.1 Transformer les tableaux en matrices

On pourra transformer explicitement les tableaux en matrices avec :

```
A=np.matrix(A)
```

lorsque l'on fera du calcul matriciel ou de l'algèbre linéaire. Cela évitera de confondre les opérateurs éléments par éléments et les opérateurs matriciels.

- $A * B$: produit élément par élément pour des array.
- $np.dot(A, B)$: produit matriciel pour des array.
- $A * B$: produit matriciel pour des matrix.

12.2 Opérateurs vectorialisés

Chaque fonction mathématique usuelle possède une fonction `numpy` qui lui est homonyme et qui calcule la même chose sur les objets de type **float**. Par exemple, il existe `math.sin` et `np.sin`. L'avantage des fonctions numpy `f`, c'est que si `a` est un tableau numpy `a=array[$a_0,...,a_{n-1}$]`, l'appel `f(a)` renvoie le tableau `array[$f(a_0),...,f(a_{n-1})$]`.
Ce mécanisme s'applique aussi aux matrices.

Appliquer une fonction à un tableau : 2 possibilités

- Utiliser les fonctions de numpy :

```
import numpy as np
def f(t):
    return np.sin(5*t)*np.exp(t)
x=np.linspace(-np.pi,np.pi,100)
y=f(x)
```

- Vectorialiser f pour en faire une fonction application à un tableau :

```
import numpy as np
import math as m
def f(t):
    return m.sin(5*t)*m.exp(t)
x=np.linspace(-np.pi,np.pi,100)
fv=np.vectorize(f)
y=fv(x)
```

12.3 Sauver et charger des tableaux

On peut facilement sauver un tableau NumPy sous deux formats :

- en texte ascii avec np.savetxt (si on a besoin de voir le tableau dans un éditeur de texte) pour les tableaux de dimension ≤ 2.
- en format binaire de NumPy avec np.save (pour des fichiers moins gros en mémoire). Pour des tableaux de dimension trois ou plus, il faut utiliser np.save

De la même manière, il existe deux fonctions pour charger un tableau numpy à partir d'un fichier

```
a=np.array([[0,1],[2,3]])
np.savetxt('tableau.txt',a)
np.save('tableau.npy',a)
c=np.loadtxt('tableau.txt')
d=np.load('tableau.npy')
```

12.4 Calcul polynomial sous Numpy

Numpy offre des outils pour travailler sur des polynômes $P = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$ définis et manipulés à travers le vecteur $[a_n, a_{n-1}, \ldots, a_0]$ de leurs coefficients (dans l'ordre des degrés décroissantes).

Par exemple, pour le polynôme $P = X^5 + 4X^4 + 2X^2 + 7X + 6$:

```
In[] : p=np.poly1d([1,4,0,2,7,6])
In[] : p.order # le degre de p
Out[]: 5
In[] : p.c # les coefficients de p
Out[]: array([1, 4, 0, 2, 7, 6])
In[] : p[1] # le coefficient du terme de degre 1
Out[]: 7
In[] : p(10) # la valeur de p en 10
Out[]: 140276
In[] : p.r # les racines de p (reelles ou complexes)
```

Table des matières