Résumé Python

Table des matières

1 Commentaires

Tout ce qui suit # sur une ligne est un commentaire

Tout ce qui est entre des triples guillemets """ un commentaire

```
2 + 4 # Ceci est une addition
6

"""
Un long commentaire
sur plusieurs lignes
2+3 ne sera pas exécuté !
"""
```

2 Types

Les types de base sont

`str`	Chaine de caractères : "une chaine" ou 'une chaine'.
`int`	Nombres entiers
`float`	Nombres à virgule flottante : `3.14`
`bool`	Valeurs `True` et `False`
`list`	Listes d'items : `[3,1,4]`
`tuple`	Listes immuable d'items : `(3,1,4)`
`dict`	Dictionnaire `{"A":3,"B":1,"C":4]}`
`Nonetype`	Le type de la valeur `None`.

Certaines bibliothèques définissent de nouveaux types.

`type(expression)`	Retourne le type d'une expression

2.1 Conversion de type

On peut convertir d'un type à un autre quand cela a un sens.

```
int(A)
float(A)
str(A)
list(A)
```

```
>>> list(Bonjour)
['B', 'o', 'n', 'j', 'o', 'u', 'r']

>>> float(5)

>>> int("42")
42

>>> float("42.5")
42.5
```

3 Entrée-sortie

3.1 Print

`print(A, B, C, ...)` imprime les chaines *A*, *B*, *C*, etc.

```
>>> print("Ceci est un exemple")
Ceci est un exemple

>>> print("Ceci", 10, sqrt(4), "Exemple")
Ceci 10 2 Exemple
```

La commande print termine une ligne par un retour de chariot. Pour changer le dernier caractère écrit par la commande print, on utilise le paramètre *end*.

```
>>> for compteur in range(4):
...     print(compteur,end=", ")
0, 1, 2, 3,
```

3.2 Input

`input(Invite)` demande d'enter quelque chose en affichant la chaîne de caractère `Invite`. Le message d'invite peut être laissé vide ou être une chaine formatée.

```
input("Quel est votre nom ?")
```

Note : le résultat est toujours de type `srt`, donc une chaine de caractères. Si nécessaire, on peut le convertir la chaine entrée en entier avec `int` et en nombre à virgule avec `float`.

`int(input())`	Demande un nombre entier
`float(input())`	Demande un nombre nombre à virgule

3.3 Fichiers

On ouvre un fichier avec `A=open(NOM,OPTIONS)`. Le fichier est assigné à variable A qui est utilisée par la suite pour référer au fichier ouvert. Pour économiser les ressources du système, on doit fermer le fichier quand il n'est plus utilisé avec `A.close()`.

Les options d'ouverture sont les suivantes :

- `'r'` Ouvrir en lecture (*Read*). C'est l'option par défaut de la commande open.
- `'w'` Ouvre en écriture (*Write*). Le contenu du fichier ouvert est remplacé.
- `'a'` Ouvre en écriture . Si le fichier existe déjà, le nouveau contenu sera ajouté (*Append*) au contenu existant.
- `'x'` Ouvre un nouveau fichier en écriture . Si le fichier existe déjà, on obtient une erreur.

`A=open(NOM,OPTIONS)`	Ouvre le fichier NOM en lecture et l'assigne à variable A
`A.read()`	Retourne une chaine correspondant au contenu du fichier A
`A.readline()`	Retourne une chaine correspondant à la ligne actuelle du fichier A et passe à la ligne suivante.
`A.readlines()`	Retourne une liste de chaines, un item par ligne du fichier A.
`A.write(CHAINE)`	Écrit la chaine CHAINE dans le fichier A.
`A.close()`	Ferme le fichier A

4 Chaines de caractères

Une chaine de caractère doit être délimitée par `" "` ou `' '`.

```
"Ceci est un phrase"
'ceci est une phrase'
```

Les caractères ne pouvant être entrés directement, comme `"` et `'`, doivent être précédés par \.

- `\n` Pour une nouvelle line
- `\\` Pour les barres obliques inverse \
- `\"` Pour les guillemets doubles
- `\'` Pour les guillemets simples

```
>>> print("Gullemets \" et backslash \\")
Gullemets " et backslash \

>>> print("Retour de chariot:\nsuite de la phrase")
Retour de chariot:
suite de la phrase
```

`len(CHAINE)`	Retourne le nombre de caractères (la longueur) de CHAINE
`CHAINE.upper()`	Transforme toute les lettres de CHAINE en majuscules
`CHAINE.lower()`	Transforme toute les lettres de CHAINE en minuscuscules

4.1 Sélection de sous-chaines

On sélectionne une sous-chaîne en utilisant les crochets. Cette opération fonctionne que la sélection de sous-listes.

```
>>> phrase="Ceci est une phrase"
>>> phrase[:4]
Ceci
>>> phrase[4:]
' est une phrase'
>>> phrase[5:8]
'est'
>>> phrase[-6:]
'phrase'
```

4.2 Trouver une séquence dans une chaine

`SOUS-CHAINE in CHAINE`	Teste si la chaine SOUS-CHAINE est présente dans la chaine CHAINE ; retourne une valeur booléenne.
`CHAINE.find(SOUS-CHAINE)`	Retourne l'index de la première occurence de SOUS-CHAINE dans la chaine CHAINE ou -1 s'il n'y en a pas.
`CHAINE.find(SOUS-CHAINE,n,m)`	Retourne l'index de la première occurence de SOUS-CHAINE dans la chaine CHAINE entre les indices n et m ; ou -1 s'il n'y en a pas.

```
>>> chaine="Bonjour Yannick !"
>>> "Yannick" in chaine
True
>>> "Sophie" in chaine
False
>>> chaine.find("Yannick")
8
>>> chaine.find("Sophie")
-1
```

4.3 Chaines formatées (*f-string*)

Une chaine peut être une *f-string* (ou **chaine formatée**). La chaine doit être précédée d'un caractère "f".

Le format général d'une f-string est le suivant : `fCHAINE`. La CHAINE de caractère, les expressions de la forme `{ }` sont remplacées en les évaluant et formatant le résultat selon une spécification précise.

La forme générale d'une expresion entre accolades dans une *f-string* est la suivante :

`{INSTRUCTIONS: REMPLISSAGE ALIGNEMENT LARGEUR PRÉCISION TYPE}`

Les INSTRUCTIONS peuvent être n'importe quelle expression python dont le résultat d'évaluation peut être affiché.

REMPLISSAGE : Caractère à utiliser pour remplir la chaine de caractère pour qu'elle ait la LARGEUR voulue.

ALIGNEMENT :

- `<` à droite ;
- `>` à gauche ;
- `^` centré.

LARGEUR : nombre de caractères à occuper

PRÉCISION : précision de l'affichage des nombres décimaux ou nombre de caractères d'une chaine à afficher

TYPE :

- `f` Virgule flottante
- `G` Notation scientifique si grand nombre
- `E` Notation scientifique
- `d` Nombre entier
- `b` Nombre entier affiché en binaire
- `X` Nombre entier affiché en hexadécimal
- `%` Pourcentage

4.3.1 Chaine de caractères

Si `nom = "Python"`, alors

Exemple sortie	Remplacement	Remplissage	Alignement	Largeur
`'Python'`	`{nom:>20}`		`>`	20
`'Python '`	`{nom:<20}`		`<`	20
`'  Python  '`	`{nom:^20}`		`^`	20
`'**********Python'`	`{nom:*>20}`	`*`	`>`	20

```
>>> nom = "Yannick"
>>> print(f"Le nom est |{nom:>10}|")
Le nom est |   Yannick|
>>> print(f"Le nom est |{nom:<10}|")
Le nom est |Yannick   |
>>> print(f"Le nom est |{nom:^10}|")
Le nom est | Yannick  |
```

4.3.2 Nombres entiers

Si `nombre = 1325`, alors

Exemple sortie	Remplacement	Remplissage	Largeur	Alignement	Type
`'00001325'`	`{nombre:08d}`	0	8		d
`'    1325'`	`{nombre:8d}`		8		d
`'1325****'`	`{nombre:*<8d}`	*	8	<	d
`'10100101101'`	`{nombre:b}`				b
`'52D'`	`{nombre:X}`				X

4.3.3 Nombres à virgule flotante

Pour arrondr avec des chiffres significatifs, utiliser la précision et `f`.

Si `nombre = 3.14159265357989`, alors

Exemple sortie	Remplacement	Larg.	Align.	Précision	Type
`'3.1416'`	`{nombre:.5}`			5	
`'3.14159'`	`{nombre:.5f}`			5	f
`'   3.142'`	`{nombre:8.3f}`	8		3	f
`'3.14E+00'`	`{nombre:.2E}`			2	E
`'3.14'`	`{nombre:.3G}`			3	G
`'5.19E+49'`	`{nombre**100:.3G}`			3	G
`'25.67%'`	`{.2567:.2%}`			2	%

4.3.4 Signes des nombres

+ signe + ou - toujours présent .
(Espace) signe + sont remplacés par des espaces, signes - sont affichés.

Exemple sortie	Remplacement	Larg.	Align.	Précision	Type
'-231'	{-231:d}				d
'231'	{231: d}				d
'-231'	{-231: d}				d
'+231'	{231:+d}				d
'-231'	{-231:+d}				d
' +231'	{ 231:+6d}	6			d
' 231'	{ 231: 6d}	6			d
' -231'	{-231:+6d}	6			d
'-231 '	{-231:<+6d}	6	<		d
'+3.14'	{3.14159:+.2f}			2	f
'-3.14'	{-3.14159:+.2f}			2	f
' 3.14'	{3.14159: .2f}			2	f
'-3.14'	{-3.14159: .2f}			2	f

4.3.5 Affichage d'expressions évaluées

Un type de remplacement est utile pour afficher des expressions de la forme `x = 3.1415`. Toute expression dans une spécification de remplacement suivie de = sera remplacée par elle-même, suivie de '=' et de la valeur de l'expression. Il est aussi possible de formater la valeur de l'expression.

Si `x = 3.1415` et `nom = "Python"`, alors

Exemple sortie	Remplacement	Larg.	Align.	Précision	Type
'x = 3.1415'	{x =}				
'x=3.1415'	{x=}				
"nom = 'Python'"	{nom =}				
'x**2 = 9.8690225'	{x**2 = }				
'x**2 = 9.869'	{x**2 = :.3f}			3	f
'x**2 = 9.869E+00'	{x**2 = :.3E }			3	E
'x**2 = 9.869'	{x**2 = :>8.3f}	8	>	3	f
"nom = 'Python'"	{nom = }				
'nom = Python'	{nom = :}				
'nom = Python '	{nom = :<10}	10	<		

5 Variables

5.1 Nom de variables

- Les noms peuvent être aussi longs que l'on veut (mais on évite habituellement les noms trop long, car ils sont difficiles à lire).
- Les noms peuvent contenir des chiffres, des lettres majuscules et minuscules, mais doivent toujours débuter par une lettre.
- Les noms ne peuvent contenir d'espace, mais on peut utiliser le caractère « _ » (souligné).
- Les noms ne peuvent pas être des mots réservés, comme les noms des commandes pythons. Il y a 35 mots réservés qui ne peuvent pas être utilisés comme nom de variable.

5.1.1 Liste les mots réservés

```
False None True and as assert async await break class continue
def del elif else except finally for from global if import in is
lambda nonlocal not or pass raise return try while with yield
```

5.2 Assignation

`var=EXPRESSION`	Évalue EXPRESSION et assigne le résultat à la variable var.
`del VARIABLE`	détruire une variable (et ainsi libérer l'espace utilisé par son contenu)

5.2.1 Notes sur l'assignation

L'expression est évaluée avant d'être assignée à la variable.

On peut changer la valeur assignée à une variable plusieurs fois.

On veut assigner des valeurs de n'importe quel type à une variable.

```
>>> variable1 = 2
>>> variable2 = 3
>>> variable1+variable2
5
```

```
>>> variable1 = "Bonjour"
>>> variable2 = "Yannick"
>>> variable1+variable2
'BonjourYannick'
```

5.3 Raccourcis pour assignations fréquentes

`x+=1`	x=x+1
`x-=1`	x=x-1
`x*=2`	x=2x
`x/=2`	x=x/2
`x//=2`	x=x//2

```
>>> x=2
>>> x+=1
>>> print(x)
3
```

6 Expressions arithmétiques

La priorité des opérations est respectée et les parenthèses peuvent être utilisées.

`A+B`	Somme
`A-B`	Différence
`A*B`	Produit
`A/B`	Division
`A**B`	Puissance A^B
`pow(A,B)`	Puissance A^B
`A//B`	Division entière de A par B.
`A%B`	A modulo B (reste de division de A par B).
`abs(A)`	Valeur absolue de A.
`round(A)`	Entier le plus proche de A.
`round(A,n)`	Arrondissement de A à n chiffres après le point décimal.
`sum(A1,A3,A3)`	Somme des éléments d'une énumération, comme une liste
`min(A1,A2,A3`	Minimum des arguments
`max(A1,A2,A3)`	Maximum des arguments

```
>>> x=2
>>> y=3.5
>>> print(2**3-y)
4.5
>>> print(11//3)
3
>>> print(11/3)
3.6666666666666665
>>> sum([2,3,1])
6
>>> round(3.14159265357989,4)
3.1416
```

7 Blocs

Un **bloc d'instructions** est partie d'un programme qui débute par `:`, suivie par des lignes intentées et se termine quand l'indentation revient au niveau initial. Un bloc est similaire à une expression entre parenthèse : c'est une partie d'un programme qui est considéré comme un tout.

Dans le code suivant, il y a un bloc constitué des lignes 2,3 et 4.

```
1 def f(x):
2     a = x**2
3     b = a+x
4     return a+b
```

Comme les parenthèses, les blocs peuvent être imbriqués, c'est à dire qu'il peut y avoir des blocs à l'intérieur d'autres blocs. Dans le code suivant, il y a deux blocs imbriqués : un bloc associé à la commande `while` allant de la ligne 5 à la ligne 9 et un bloc associé à la commande `if` constitué des lignes 7 et 8.

```
1 a = 0
2 b = 1
3 while a < 10:
4     a = a+1
5     if a%3 == 0:
6         print(a)
7         b = b +1
8     print(a,b)
```

8 Fonctions

Forme de la définition d'une fonction

```
def Nom(ARGUMENTS):
    COMMANDES
    ...
    [return EXPRESSION]
```

```
>>> def succ(n):
        return n+1
```

On peut spécifier les types attendus en entrée et en sortie, ce qui peut prévenir certaines erreurs

```
>>> def division0(x:float,y:float) -> float:
        if y!=0:
            return x/y
        else:
            return 0
```

9 Opérations booléennes et comparaisons

9.1 Opérations logiques

`A and B`	A et B
`A or B`	A ou B
`not A`	non-A

9.2 Opérateurs de comparaisons

Les opérateurs de comparaisons retournent toujours une valeur booléenne `True` ou `False`.

`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 plus petit ou égal à y ?
`x > y`	x est strictement plus grand que y ?
`x >= y`	x est plus grand ou égal à y ?
`x is y`	id(x)=id(y) ?
`x in y`	x est un élément de y ?

10 Structures conditionnelles

Exécution d'un bloc de commande si une certaine condition est satisfaite. CONDITION doit retourner une valeurs booléenne `True` ou `False`.

```
if CONDITION:
  BLOC 1
[elif condition2:
  BLOC 2]
[else:
  BLOC 3]
```

11 Listes et énumérations

Une *liste* est une collection ordonnées d'éléments. Ces éléments peuvent être de type différents. On entre les listes en mettant ses éléments entre crochets, séparés par des virgules.

Une liste : `[1,2,3,4,23,"Blah"]`

La liste vide : `[]`

11.1 Indexation

Si A est une liste ou un objet énumérable, on peut référer au n-ième item de l'énumération avec `A[n+1]`.

L'indexation en python commence toujours à 0. `A[n]` est donc le $n+1$-ième élément d'une liste A

Si n est négatif, on compte à partir de la fin. `A[-1]` est donc le dernier élément.

`A[m:n]` est la liste des éléments de la liste A du m-ième au $n+1$-ième

Si m est laissé vide, l'énumération débute au début de la liste

Si n est laissé vide, l'énumération se termine à la fin de la liste

`A[m:n:k]` est la liste des éléments de la liste A du m-ième au $n+1$-ième par sauts de grandeur k (appelé le **pas**).

```
>>> A=['a','b','c','d','e','f']
>>> A[2]
'c'

>>> A[1:3]
['b', 'c']

>>> A[2:]
['c', 'd', 'e', 'f']

>>> A[:2]
['a', 'b']

>>> A[-3]
'b'
```

Pour énumérer une liste en ordre inverse, on peut utiliser un pas négatif.

```
>>> A=['a','b','c','d','e','f']
>>> A[::-1]
['f', 'e', 'd', 'c', 'b', 'a']
```

`len(L)`	Donne la longueur de la liste L.
`L1+L2`	Donne une nouvelle liste en ajoutant L2 à la fin de L1
`k*L1`	Donne une nouvelle liste constituée de k copies de L1
`L.append(A)`	Ajoute l'élément A à la fin de la liste L.
`L.insert(n,A)`	Ajoute l'élément A à la liste L pour que sont indice soit n.

`L.pop()`	Retourne et retire le denier élément de la liste L.
`L.pop(n)`	Retourne et retire l'élément d'indice n de la liste L.
`range(n)`	liste des entiers de 0 à n.
`range(m,n)`	liste des entiers k tels que $m \leq k < n$.
`range(m,n,d)`	liste des entiers k tels que $m \leq k < n$ par sauts de d.
`map(F,L)`	Applique la fonction F à chacun des éléments de la liste L.
`L.count(A)`	Donne le nombre d'occurrences de l'élément A dans la liste L.
`L.sort()`	Met la liste L en ordre croissant ou alphabétique.
`L.sort(reverse=True)`	Met la liste L en ordre décroissant ou alphabétique.
`sorted(L)`	Retourne une copie en ordre croissant de la liste L.
`sorted(L,reverse=True)`	Retourne une copie en ordre décroissant de la liste L.
`L.copy()`	Retourne une copie de la liste L.
`L.deepcopy()`	Retourne une copie d'un tableau imbriqué L.

11.2 Listes de listes et tableaux

Une liste peut contenir d'autres listes.

```
>>> A=[[0],[1,2],[3,4,5],[6,7,8,9]]
>>> A[2][1] # Liste no 2, entrée no 1
4
```

Il est possible de faire des « listes de listes de listes » de cette manière avec autant de niveaux que l'on veut.

Un tableau $n \times m$ est une liste de n listes de m éléments.
On sélectionnes des entrées particulières comme avec les listes.

```
>>> A=[[0,1,2,3],[4,5,6,7],[8,9,10,11]] # Un tableau 3x4
>>> A[1][2] # Ligne no 1, colonne no 2
6
```

11.3 Générer des listes

La commande *range* créé des énumérations. On peut convertir ces énumérations en lites avec la commande *liste*.

```
>>> list(range(2,6))
[5, 6, 7, 8, 9]
>>> list(range(0,10,2))
[0, 2, 4, 6, 8]
```

12 Dictionnaires

Un **dictionnaire** est une structure de donnée (type `dict`) associant des clefs artibraires à des valeurs.
Syntaxe : `{ CLEF1: VALEUR1, CLEF2: VALEUR2, ...}`
Il peut y avoir autant d'association que l'on veut. Les clefs doivent être des chaines de caractères ou des nombres et les valeurs peuvent être de n'importe quel type.

```
>>> Population_parc={"Tigre": 4, "Lion": 6, "Éléphant": 25, "Zebre":33}
>>> Population_parc["Lion"]
6
>>> Population_parc["Lion"]=5
>>> Population_parc["Lion"]
5
```

`D.keys()`	Retourne un énumérateur des clefs du dictionnaire D.
`D.copy()`	Retourne une copie du dictionnaire D (avec un id différent)
`D.pop(CLEF)`	Retourne la valeur de `Dictionnaire[CLEF]` et retire cette valeur du dictionnaire D

On peut énumérer les clefs d'un dictionnaire pour une boucle for comme avec une liste.

```
>>> for Animal in Population_parc:
...     print(Animal)
...
Tigre
Lion
Éléphant
Zebre
>>> for Animal in Population_parc:
...     print(f"{Animal}: {Population_parc[Animal]}")
...
Tigre: 4
Lion: 6
Éléphant: 25
Zebre: 33
```

13 Tuples

Les **tuples** sont des objets Python similaires aux listes, mais donc les items ne peuvent pas être modifiés. En conséquence, les fonctions `append()`,`pop()`, `insert()` et `sort()` ne peuvent pas être utilisées avec des tuples.

```
>>> Tuple = (3,1,2)
>>> Liste = [3,1,2]
>>> Tuple[0]
3
>>> Liste[0]
3
```

```
>>> Tuple[0] = 5
TypeError: 'tuple' object does not support item assignment
>>> Liste[0] = 5
>>> Liste
[5, 1, 2]
```

14 Boucles

14.1 Boucles limitées (*for*)

Une boucle `for` répète un bloc d'instruction en donnant à une variable toutes les valeurs possibles d'un numérateur. Le nombre de répétition du bloc est déterminé par le nombre de valeurs possibles dans l'énumération.

```
for VARIABLE in ENUMÉRATION:
  BLOC
```

La commande `range` construit des énumérations de nombres entiers.

```
>>> for compteur in range(3,9):
...     print(compteur)
3
4
5
6
7
8
```

On peut aussi énumérer les éléments d'une liste, une chaîne de caractères ou n'importe quel objet que python considère comme "énumérable".

```
>>> for nom in ["Yannick","Marcella","Dim","Cath"]:
...     print(nom)
Yannick
Marcella
Dim
Cath
```

14.2 Boucles conditionnelles (*while*)

La commande **while** permet de répéter d'exécution d'un bloc d'instructions tant qu'une condition est satisfaite. Cette condition doit retourner une valeur booléenne `True` ou `False`.

```
while CONDITION:
  BLOC
```

Note Il est possible que la condition d'une boucle while soit toujours satisfaite. Dans ce cas le bloc d'instruction est répété à l'infini.

On utilise souvent les boucles `while` avec des variables auxiliaires comme des compteurs.

```
>>> compteur = 3
... while compteur <=9:
...     compteur += 1
...     print(compteur)
4
5
6
7
8
9
10
```

15 Bibliothèques

Pour importer une bibliothèque comme la bibliothèque `math` :

```
import math
```

Pour importer une bibliothèque comme la bibliothèque `math` mais en lui donnant un autre nom :

```
import math as ma
```

16 Bibliothèque math

Commandes mathématiques qui nécessitent de charger la bibliothèque `math` : `import math`

```
>>> import math
>>> math.cos(math.pi)
-1.0
```

Constantes mathématiques

`math.pi`	Constante $\pi = 3.14159265357989\ldots$
`math.e`	Constante e $= 2.718281\ldots$
`math.inf`	Constante « infinie » ∞
`math.nan`	Constante « pas un nombre » (*Not a number*)

Fonctions mathématiques.

Si x : `float`, n : `int`, alors

`math.sqrt(x)`	Racine carrée $\sqrt{x}$
`math.cbrt(x)`	Racine cubique $\sqrt[3]{x}$
`math.exp(x)`	Exponentielle e^x
`math.log(x)`	Logarithme $\ln(x)$
`math.log(x,b)`	Logarithme à base b : $\log_b(x)$
`math.exp2(x)`	Puissance de 2 : 2^x
`math.log2(x)`	Logarithme à base 2 : $\log_2(x)$ (plus précis que `math.log(x,2)`).
`math.degree(x)`	Conversion rads $\rightarrow$ degrés
`math.radians(x)`	Conversion degrés $\rightarrow$ rads
`math.sin(x)`	Sinus : $\sin(x)$
`math.cos(x)`	Cosinus : $\cos(x)$
`math.tan(x)`	Tangente : $\tan(x)$
`math.acos(x)`	Arcsinus : $\arccos(x)$
`math.asin(x)`	Arccosinus : $\arcsin(x)$
`math.atan(x)`	Arctangente : $\arctan(x)$
`math.floor`	Partie entière
`math.ceil(x)`	Plus petit entier plus grand que l'argument
`math.gcd`	Plus grand commun diviseur des arguments
`math.lcm`	Plus petit commun multiple des arguments
`math.comb(n,k)`	Combinaisons de k dans n
`math.perm(n,k)`	Permutations de k dans n
`math.prod`	Produit des éléments d'une liste
`math.factorial(n)`	Factorielle $n!$

17 Biblohtèque scipy

Cette bibliothèque contient des définitions de plusieurs constantes utiles en science. Toutes les constantes sont données en unités SI.

Après `import scipy.constants as cst`

`cst.g`	Accélération gravitationnele g
`cst.G`	Constante gravitationnele G
`cst.Avogadro`	Nombre d'Avogadro N_A
`cst.c`	Vitesse de la lumière dans le vide c
`cst.e`	Charge élémentaire e
`cst.m_p`	Masse du proton m_p
`cst.m_e`	Masse de l'électron m_e
`cst.m_n`	Masse du neutron m_n
`cst.mu_0`	Constante magnétique μ_0
`cst.epsilon_0`	Constante électrique ε_0
`cst.h`	Constante de Planck h

Quelques constantes utiles pour les conversions d'unités.

`cst.astronomical_unit`	Unité astronomique en mètres
`cst.calorie`	Une calorie en Joules
`cst.eV`	Un électron-Volt en Joules
`cst.minute`	une minute en secondes
`cst.hour`	une heure en secondes
`cst.day`	un jour en secondes
`cst.year`	une année en secondes

18 Bibliothèque random

Cette bibliothèque contient des foncions permettant de générer des nombres et des séquences (pseudo)aléatoire.

Charger cette bibliothèque avec `import random`

`random.random()`	Génère un nombre décimal aléatoire entre 0 et 1.
`random.uniform(A,B)`	Génère un nombre décimal aléatoire entre A et B.
`random.randint(A,B)`	Génère un nombre entier aléatoire entre les entiers A et B.
`random.normavariate(mu,sigma)`	Génère un nombre décimal aléatoire selon une distribution normale de moyenne mu et d'écart type sigma.
`random.choice(L)`	Génère un élément aléatoire de la liste *L*.
`random.choice(L,k=NB_CHOIX)`	Génère une liste de *NB_CHOIX* éléments aléatoires tirées avec remise de la liste L.
`random.sample(L,k=NB_CHOIX)`	Génère une liste de *NB_CHOIX* éléments aléatoires tirées sans remise de la liste *L*.
`random.shuffle(L,k=NB_CHOIX)`	Génère une liste de *NB_CHOIX* éléments aléatoires tirées sans remise de la liste *L*.
`random.seed(A)`	Règle de germe du générateur de nombre aléatoire à la valeur *A*.

```
>>> # Génère un nombre aléatoire dans l'intervalle [0,100]
>>> import random
>>> 100*random.random()
94.80261695959705
```

```
>>> liste = ["Bonjour", "Allo", "Salut"]
>>> print(random.choice(liste))
Allo
>>> print(random.choice(liste))
Salut
```

19 Bibliothèque numpy

Commandes mathématiques qui nécessitent de charger la librairie `numpy` : `import numpy as np`

19.1 Fonctions mathématiques

Les fonctions et constantes de la bibliothèque math sont aussi définies dans la bibliothèque *numpy*.

19.2 Tableaux

Un tableau rectangulaire de données numériques est un objet défini dans la bibliothèque numpy appelé **array**.

Un tableau unidimensionnel de longueur 5 :

```
>>> A = np.array([[1,2,3,4,5])
print(A)
[1,2,3,4,5]
```

Un tableau multidimensionnel 2×3 :

```
>>> A = np.array([[1,2,3],[4,5,6]])
>>> print(A)
[[1 2 3]
 [5 6 7]]
```

19.2.1 Application de fonctions à toutes les valeurs d'un tableau

On peut appliquer une fonction simultanément à toutes les entrée d'un tableau.

```
>>> A = np.array([1,2,3,4,5])
>>> print(A**2)
[ 1  4  9 16 25]
```

19.3 Modification de tableaux à une dimension

`np.append(T,val)`	Retourne un nouveau tableau où on ajoute la valeur *val* à la fin du tableau *T*.
`np.insert(T,val,pos)`	Retourne un nouveau tableau où on ajoute la valeur *val* à la position *pos* du tableau *T*.
`np.column_stack((x,y))`	Créé un tableau à prenant les tableaux unidimensionnels *x* et *y* comme colonnes.

Note : il est préférable d'utiliser la version d'*append* des listes en convertir en tableau numpy si nécessaire, car la version liste est plus efficace pour ajouter un élément.

19.3.1 Création de tableau à une dimension avec valeurs constantes

`np.empty(dimentions)`	Création d'un tableau vide dont les dimentions sont dimentions.
`np.full(n,valeur)`	Créé un *array* de longueur n où toutes les entrées sont *valeur*.
`np.full_like(A,valeur)`	Créé un *array* de même dimension que A où toutes les entrées sont *valeur*.
`np.zeros(n)`	Créé un *array* de longueur n où toutes les entrées sont 0 (zéro).
`np.zeros_like(A)`	Créé un *array* de même dimensions que A où toutes les entrées sont 0 (zéro).

19.3.2 Intervalles subdivisés

`np.linspace(a,b,n)`	Liste de n points séparant l'intervalle $[a,b]$ en $n-1$ intervalles égaux
`np.arange(a,b,delta)`	Liste de points séparant l'intervalle $[a,b]$ en intervalles égaux de largeur delta.

19.3.3 Analyse de tableau

`A.prod()`	Retourne la somme des entrées du tableau A.
`A.sum()`	Retourne la somme des entrées du tableau A.
`np.sum(A)`	Retourne la somme des entrées du tableau A.
`A.prod()`	Retourne le produit des entrées du tableau A.
`np.prod(A)`	Retourne le produit des entrées du tableau A.
`A.mean()`	Retourne la moyenne des entrées du tableau A.
`A.std()`	Retourne l'écart type des entrées du tableau A.
`A.max()`	Retourne la plus grande valeur du tableau A.
`A.min()`	Retourne la plus petite valeur du tableau A.

19.3.4 Manipulation de tableaux

`np.sort(A)`	Retourne une copie triée du tableau A.
`A.sort()`	Trie le tableau A.
`A.flip()`	Inverse l'ordre du tableau A.
`np.append(A,x)`	Ajoute x à la fin du tableau A.
`np.insert(A,x,n)`	Ajoute x à la position n du tableau A.
`np.delete(A,n)`	Enlève l'entrée du tableau A à la position n.

19.4 Tableau multidimensionnels

Un tableau *numpy* peut avoir plusieurs dimensions. On créé de tels tableaux avec les listes imbriquées. Par exemple, un tableau 2×3 :

```
>>> A = np.array([[1,2,3],[4,5,6]])
>>> print(A)
[[1 2 3]
 [4 5 6]]
```

19.4.1 Création de tableau à plusieurs dimensions avec valeurs constantes

`np.empty(dimentions)`	Création d'un tableau vide dont les dimensions sont dimensions.
`np.zeros((n,m))`	Créé un *array* $n \times m$ où toutes les entrées sont 0 (zéro).
`np.ones((n,m))`	Créé un *array* $n \times m$ où toutes les entrées sont 0 (zéro).
`np.full((n,m),valeur)`	Créé un *array* $n \times m$ où toutes les entrées sont *valeur*.
`np.zeros_like(A)`	Créé un *array* de même dimensions que A où toutes les entrées sont 0 (zéro).

19.5 Analyse de données

On peut utiliser ces commandes par ligne ou par colonnes avec l'argument optionnel *axis*. Dans ce cas, le résultat est un *array* des résultats. Lignes = `axis=0`, Colonnes = `axis=1`.

`np.sum(A,axis=0)`	Sommes par ligne
`A.sum(axis=1)`	Sommes par colonnes
`A.mean(axis=0)`	Moyenne par linges
`A.mean(axis=1)`	Moyenne par colonnes
	etc.

Remarque : on peut utiliser l'une ou l'autre forme des commandes données.

19.6 Courbes polynomiales de tendance

`np.polyfit(A,B,d)`	Retourne la liste des coefficients du polynôme de degré d qui s'ajuste le mieux aux données des tableaux A (en x) et B (en y).
`np.poly1d(Liste_coef)`	Retourne la fonction définie par le polynôme dont les coefficients sont dans la liste `Liste_coef`. Une telle liste est retournée par `polyfit`.

Pour définir la fonction polynomiale de degré 1 décrivant le mieux les valeur de B en fonction de A :

```
coefs=np.polyfit(A,B,1)
f=np.poly1d(coef)
```

19.7 Modifications de tableaux multidimensionnels

On utilise les commandes de base *append*, *insert* et *delete* sur les tableau multidimensionnels en spécifiant si elles doivent s'exécuter sur des ligne ou des colonnes avec le paramètre *axis*.

`np.append(A,vals,axis=0)`	Ajoute la ligne *vals* au tableau *A*.
`np.insert(A,vals,pos,axis=1)`	Ajoute la colonne *vals* au tableau *A* à la position *pos*.
`np.delete(A,pos,axis=1)`	Enlève la colonne *pos* du tableau *T*.

19.8 Charger des données à partir d'un fichier

Si le contenu du fichier `données.csv` est le suivant :

```
1, 3.4
2, 4.2
3, 5.3
4, 6.1
```

alors la commande *loadtxt* de la biliothèque numpy permet de les charger dans un *array* numpy.

```
>>> import numpy as np
>>> données = np.loadtxt("données.csv",delimiter=",")
>>> print(données)
[[1.  3.4]
 [2.  4.2]
 [3.  5.3]
 [4.  6.1]]
```

Calcul de la moyenne, de l'écart type et de la médiane de la 2e colonne des données

```
>>> colonne = données[:,0] # Sélectionne la 2e colonne
>>> colonne.mean()
2.5
>>> colonne.std()
1.118033988749895
>>> np.median(colonne)
2.5
```

19.8.1 Écrire des données dans un fichier csv

La commande `np.savetxt` permet de sauver les données d'un tableau dans un fichier csv.

```
>>> import numpy as np
>>> données = np.array([[1,2,3,4,5],[3.4,2.3,4.5,5.3,6.2]]).T
>>> np.savetxt("données2.csv",données,delimiter=",")
```

Librarie matplotlib

On charge la bibliothèque mathplotlib avec `import mathplotlib.pyplot as plt`.

On construit le graphique à l'aide de commande en spécifiant le contenu et on l'affiche ensuite avec `show()` ou on sauve le graphique dans un fichier avec `savefig()`.

19.9 Courbe de fonctions

19.10 Nuage de point (*scatter*)

`plt.scatter(X,Y)` où `X` et `Y` sont des `array` de données.

```
>>> import mathplotlib.pyplot as plt
>>> plt.title("Titre du graphique")
>>> X = array([1,2,3,4,5,6])
>>> Y = array([1.2,1.9,3.2,4.3,5.0,6.2])
>>> plt.scatter(X,Y)
>>> plt.xlabel=("x")
>>> plt.xlabel=("y")
>>> plt.show()
```

19.10.1 Types de lignes et points

`-`	Ligne pleine
`-`	Ligne à tirets
`.`	Petits points
`o`	Gros points
`-.`	Ligne avec points et tirets
`:`	Ligne pointilée
`+`	Signe « + »

19.11 Affichage et exportation

19.12 Dimensions du graphique

`plt.xlim(xmin,xmax)`	Change les limites du graphique pour le limiter à xmin $\leq x \leq$ xmax
`plt.ylim(ymin,ymax)`	Change les limites du graphique pour le limiter à ymin $\leq y \leq$ ymax
`plt.grid(bool)`	Si `bool` est `True`, ajoute un grille au graphique.

19.12.1 Titre et légendes

`plt.title(titre)`	Ajoute la chaine `titre` comme titre du graphique
`plt.xlabel(chaine)`	Utilise la chaine de caractère `chaine` comme étiquette sous l'axe des abscisses.
`plt.ylabel(chaine)`	Utilise la chaine de caractère `chaine` comme étiquette pour l'axe des ordonnées.
`plt.show()`	Affiche le graphique actuel et le remet à zéro.

`plt.savefig("NOM_FICHIER.EXT")`	Exporte le graphique dans le fichier nommé "NOM_FICHIER.EXT" dans le format associé à l'extension EXT. Les formats suivants sont supportés : `png`, `pdf` et `svg`. (D'autres formats peuvent être supportés selon les plateformes).

19.13 Ajouter des droites à un graphique

`plt.axline((x1,y1),(x2,y2)`	Ajoute une ligne infinie passant par les points `(x1,y1)` et `(x2,y2)`
`plt.axline((x1,y1),slope=m`	Ajoute une ligne infinie passant par le point `(x1,y1)` et de pente `m`.
`plt.axhline(C)`	Ajout d'une ligne infinie horizontale d'équation $y = C$.
`plt.axvline`	Ajout d'une ligne infinie verticale d'équation $x = C$.

20 Débuggage et efficacité

20.1 Commandes spéciales de Jupyter

On peut mettre sur la première ligne d'une cellule d'une feuille de calcul jupyter certaines commandes spéciales permettant d'étudier le fonctionnement d'un programme.

`%pdb on`	Active le lancement automatique de pdb en cas d'erreur.
`%pdb off`	Désactive le lancement automatique de pdb en cas d'erreur.
`%debug`	Lance le débugger s'il y a un message d'erreur auparavant. À utiliser *après* l'erreur, dans une nouvelle cellule.
`%time`	Exécute en déterminant le temps d'exécution
`%timeit`	Éxécute le programme plusieurs fois et calcule la moyenne et l'écart type du temps d'exécution.

20.2 Débugger pdb

Le débugger interne de Python est `pdb`. Dans ce mode, l'interpréteur Python attend avant d'exécuter chaque instruction et permet d'examiner le contenu des variables, ce qui peut aider à déterminer la cause d'un problème.

Les commandes de pdb les plus utiles sont les suivantes.

`h`	Aide ((*Help*).
`n`	Exécute la prochaine instruction
`s`	Exécute la prochaine instruction en entant dans la fonction
`c`	Continuer jusqu'au prochain `breakpoint`.
`r`	Continuer jusqu'à la prochaine instruction `return`.
`p VAR`	Affiche la la valeur de la variable VAR.
`l`	Afficher les lignes de code autour de la ligne actuelle
`a`	Afficher les arguments de la fonction actuelle
`q`	Quitter le débugger.

21 Messages d'erreur fréquents

21.1 Parenthèse manquante

Correctif général : ajouter la parenthèse manquante.

```
>>>3*2+5)
SyntaxError: unmatched ')'
```

Correction : `3*(2+5)`

```
>>> 5*(2+3
SyntaxError: '(' was never closed
```

Corretion : `5*(2+3)`

```
>>> math.sin(0
SyntaxError: '(' was never closed
```

Corretion : `math.sin(0)`

21.2 Variable non définie

Utilisation d'une variable qui n'a pas encore de valeur définie

```
>>> print(x)
NameError: name 'x' is not defined

>>> x = 3
>>> print(x)
3
```

21.3 Problème d'indentation

Indentation de trop

```
>>> x = 3
>>>    print(x)
IndentationError: unexpected indent
```

Correctif :

```
>>> x = 3
>>> print(x)
3
```

Indentation manquante

```
>>> if condition:
... print(x)
IndentationError: expected an indented block after 'if' statement on line 1
```

Correctif :

```
>>> if condition:
...     print(x)
IndentationError: expected an indented block after 'if' statement on line 1
```

```
>>> def f(x):
...     y = x + 1
... return y
SyntaxError: invalid syntax
```

Correctif :

```
>>> def f(x):
...     y = x + 1
...     return y
```

21.4 Problème de types – opération non valide

On tente de faire une opération qui n'a pas de sens.

```
>>> "Bonjour" + 2
TypeError: can only concatenate str (not "int") to str
```

Correctif : s'assurer que les types des entrée de l'opération sont adéquats :

```
>>> "Bonjour" + str(2)
'Bonjour2'
```

21.5 Confondre '=' et '=='

Convusion entre assignation '=' et comparaison '==' :

```
>>> if x = 1:
...     print("condition satisfaite")
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Correctif :

```
>>> if x == 1:
...     print("x=1")
```

21.6 Division par zéro

Il y a un division par zéro lors de l'exécution du programme

```
>>> x = 2
>>> y = 1/(x**2-4)
ZeroDivisionError: division by zero
```