PYTHON
Version du lundi 13 octobre

Table des matières

1 Premiers pas **5**
1.1 Les opérations de base . . . 5
1.2 Les variables . . . 6
1.3 Le mot clé **del** . . . 7
1.4 Types de données . . . 7
1.4.1 Les entiers : **int** . . . 8
1.4.2 Les réels : **float** . . . 9
1.4.3 Les nombres complexes : **complex** . . . 9
1.4.4 Les booléens : **bool** . . . 10
1.4.5 Les chaînes de caractères : **str** . . . 10
1.4.6 Le type **tuple** . . . 13
1.4.7 Les listes : **list** . . . 15
1.5 Écrivons le programme dans un fichier ! . . . 18
1.6 Le transtypage . . . 19
1.7 Les commentaires . . . 21

2 Les structures conditionnelles **21**
2.1 Différence entre Expression et Instruction . . . 21
2.2 Expression booléenne . . . 22
2.3 Structure if . . . 23
2.4 La structure if...else . . . 24
2.5 Structure if...elif...else . . . 24

3 Les boucles **25**
3.1 La boucle **for** . . . 25
3.2 La boucle **while** . . . 28
3.3 Le mot-clé **break** . . . 29
3.4 Compréhension de liste avec l'instruction **for** . . . 30

4 Les fonctions **30**
4.1 Définition d'une fonction . . . 30
4.2 Passage de paramètres à la fonction . . . 31
4.3 Valeur retournée par une fonction . . . 33
4.4 La portée des variables . . . 34
4.5 Utilisation de bibliothèques . . . 37

5 Python est un langage orienté objet **40**
5.1 Qu'est-ce qu'un objet ? . . . 40
5.2 Comment connaître les attributs et les méthodes d'un objet ? . . . 41
5.3 Comment accéder à un attribut ? . . . 43
5.4 À quoi sert une méthode ? . . . 43
5.5 Les objets de la classe **str** . . . 45
5.5.1 Introduction aux objets **str** . . . 45
5.5.2 La méthode **count** . . . 45
5.5.3 La méthode **lower** . . . 46
5.5.4 La méthode **replace** . . . 46
5.5.5 La méthode **split** . . . 46
5.6 Les objet de la classe **list** . . . 47
5.6.1 La méthode **append** . . . 47
5.6.2 La méthode **insert** . . . 47
5.6.3 La méthode **pop** . . . 47
5.6.4 La méthode **remove** . . . 48

6 La bibliothèque numpy **48**
6.1 Qu'est-ce que numpy ? . . . 48
6.2 Création de tableaux (ndarrays) numpy . . . 49
6.2.1 À partir d'une liste numérique . . . 49
6.2.2 À partir de fonctions de numpy . . . 51
6.3 Les attributs **size** et **shape** . . . 53
6.4 La méthode **reshape** . . . 54
6.5 Parcours de tableau . . . 54

6.5.1 Cas d'un tableau unidimensionnel 54
6.5.2 Cas d'un tableau bi-dimensionnel (matrice) . . 55
6.6 Opérations sur les tableaux 56
6.7 Fonctions agissant sur un tableau 59

7 Lire et écrire un fichier 59
7.1 Arbre des fichiers 59
7.2 Chemin relatif et chemin absolu 61
7.2.1 Le chemin absolu 61
7.2.2 Le chemin relatif 61
7.2.3 Comment connaître et changer le répertoire de travail courant ? 62
7.3 Lire et écrire du texte dans un fichier 62
7.3.1 Ouvrir un fichier 63
7.3.2 Écrire dans un fichier 64
7.3.3 Lecture d'un fichier 64
7.3.4 Écrire plusieurs lignes de texte dans un fichier . 65
7.3.5 La méthode readlines 67
7.4 Écrire autre chose que du texte 67
7.4.1 Enregistrer un objet dans un fichier 67
7.4.2 Récupérer nos objets enregistrés 68

8 Annexe : représentation d'un nombre réel en machine 69
8.1 Notation décimale d'un réel 69
8.2 Notation d'un réel en base deux 69
8.3 Codage binaire des nombres réels 70
8.4 Codage de l'exposant 70
8.5 Exemples 71
8.6 Codage d'un réel en précision simple 72
8.7 Codage d'un réel en double précision 72

Python est un langage de programmation dont la première version est sortie en 1991. Il fut créé par **Guido van Rossum** qui travaillait dans un centre de recherche en informatique au Pays-bas. Depuis 2001, une organisation à but non lucratif à été créée pour développer ce langage : la **Python Software Foundation**. Le nom du langage a été choisi en hommage à la troupe de comique des "Monthy Python". Ce langage est actuellement très utilisé en recherche - développement.

Nous allons utiliser une implémentation de Python dénommée "Pyzo". On peut la télécharger gratuitement en allant sur le site *pyzo.org*. Il suffit de télécharger la version adaptée à votre ordinateur : Windows, linux ou Mac OS X. "Pyzo" utilise une version 3 de Python.

Lorsque vous lancez l'application, vous voyez apparaître sur l'écran de votre ordinateur quelque chose qui ressemble à la figure de la page 4. Globalement, votre écran est divisé en 3 zone :

1. **l'interpréteur** en haut ;
2. **la zone d'écriture du fichier** (on écrit ici le programme Python que l'on pourra sauvegarder dans un fichier) en bas à gauche ;
3. **la zone d'aide** en bas à droite.

Il y a deux façon d'écrire un programme en Python :

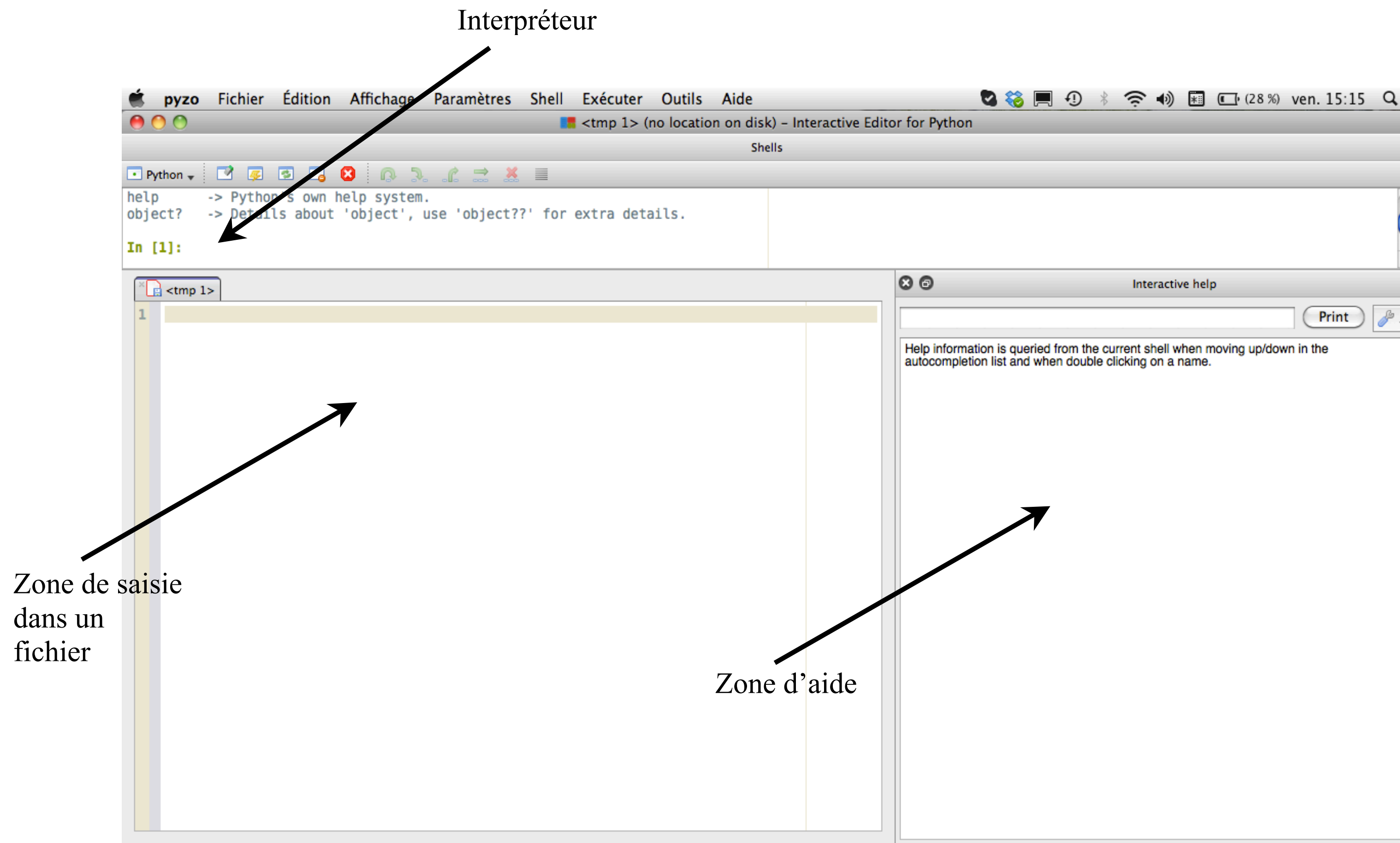
1. Vous pouvez l'écrire directement dans l'interpréteur. Chaque ligne de l'interpréteur commence par les signes **In [*numero*] :** où *numero* commence à 1 et augmente au fur et à mesure que vous tapez des lignes d'instruction.
   Dans l'espace qui suit les deux points " : " vous pouvez taper une ligne d'instructions en code Python. Vous terminez la ligne en tapant sur la touche "Entrée" de votre clavier : Python se met alors tout de suite à travailler et exécute l'instruction que vous venez d'entrer. Il affiche éventuellement le résultat dans l'interpréteur (si toutefois vous lui avez demandé de le faire...).

   Dans ce mode de programmation, vous ne pouvez entrer les instructions que *ligne par ligne* et Python les exécute au fur et à mesure : il n'est pas possible d'entrer plusieurs lignes d'instruction en un seul bloc avant de lancer le travail de Python (on verra qu'il y a néanmoins des exceptions à cela mais, en gros, c'est comme cela que çà se passe avec l'interpréteur).

   C'est donc un peu limité mais cela reste néanmoins très intéressant lorsqu'on apprend le langage ou qu'on veut tester une instruction pour voir si elle fonctionne correctement.

2. Vous pouvez aussi rédiger tout un fichier Python en utilisant la "zone de rédaction dans un fichier". Dans ce cas, vous pourrez écrire autant d'instructions que vous voulez, avec des tas de lignes si vous le souhaitez. Vous créez ainsi un **programme Python**.
   Pour exécuter ce programme, il faut d'abord enregistrer le fichier en lui donnant un **nom** : le fichier sera enregistré avec l'extension ".py" dans le répertoire que vous aurez choisi. Vous aurez par exemple : mon_beau_programme.py

   Une fois enregistré, vous pourrez exécuter toutes les instructions de votre fichier en une seul fois grâce au menu : *Exécuter > Exécuter le fichier*. Si un résultat doit s'afficher, il le sera dans l'interpréteur.


Interpréteur
Zone de saisie dans un fichier
Zone d'aide
pyzo Fichier Édition Affichage Paramètres Shell Exécuter Outils Aide
<tmp 1> (no location on disk) – Interactive Editor for Python
Shells
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
In [1]:
Interactive help
Print
Help information is queried from the current shell when moving up/down in the autocompletion list and when double clicking on a name.

1 Premiers pas

Dans cette partie, nous recommençons à zéro. Désolé pour ceux qui ont déjà des bases, voire qui se débrouillent bien. Pour se familiariser avec les opérations de base, nous allons nous placer dans l'interpréteur : si vous trouvez que la zone d'affichage est trop petite, vous pouvez toujours faire disparaître la "zone de saisie dans un fichier" ainsi que la "zone d'aide" pour faire de la place et étendre la zone de l'interpréteur au moyen de la souris.

1.1 Les opérations de base

Les symboles des opérations fondamentales sont : " * " (multiplication), " + " (addition), " - " (soustraction) et " / " (division). A ces opérations, il convient d'ajouter : " ** " (élévation à une puissance), " // " (division entière) et " % " (reste de la division entière).

Lorsqu'on tape l'instruction dans une ligne commençant par **In [*numéro*] :** et qu'on appuie sur la touche "Entrée", le résultat s'affiche en dessous, sur une ligne qui commence par **Out [*numéro*] :**

In [1] : 2 + 3
Out [1] : 5

In [2] : 5/3
Out [2] : 1.6666666666666667

Un nombre réel s'écrit avec un point " . " (notation anglo-saxonne pour la virgule) : **2.3456** par exemple. Notez que dans l'exemple de la ligne [2], Python arrondit le résultat : 1.6666666666666667 au lieu de 1.66666666666666666666...etc...

On peut utiliser la *notation scientifique* pour écrire un nombre réel. Par exemple 6,456 $\times 10^5$ s'écrira en Python : 6.456e+5 ou 6.456e5 (on peut omettre le signe + si l'exposant est positif).

In [3] : 2.567e36 * 4.56
Out [3] : 1.1705519999999998e+37

In [4] : 1.38e-7 * 2.3e-22
Out [4] : 3.1739999999999995e-29

L'élévation à une puissance se fait grâce à l'opérateur " ** " :

In [5] : 2**3
Out [5] : 8

On dispose en outre de la ***division entière*** " // " et du ***reste*** de cette division " % " :

In [6] : 7 // 2
Out [6] : 3

In [7] : 7 % 2
Out [7] : 1

On a bien : $7 = 3 \times 2 + 1$.

Les priorités entre ces opérations sont les *priorités usuelles*. Dans le cas où on veut les changer, on utilise des parenthèses :

In [8] : (7 + 3) * 2
Out [8] : 20

Sachez enfin que Python n'accorde aucune importance aux "espaces"

In [9] : 2   **   3+ 1
Out [9] : 9

Il convient cependant de ne pas abuser de cela pour des raisons de lisibilité et surtout de ne pas mettre d'espace entre deux signes du même opérateur : " * * " à la place de " ** " ou " / / " à la place de " // " : cela, Python **ne le comprend pas** !

In [10] : 2 * * 3
File "<ipython-input-17-fe628e899910>", line 1
2 * * 3
^
SyntaxError : invalid syntax

Python râle mais il est vraiment sympa car il a le bon goût de vous indiquer l'endroit où cela ne va pas...

1.2 Les variables

En informatique, une **variable** désigne un *emplacement* de la mémoire de l'ordinateur qui contient des *données*. Ces données peuvent changer au cours du déroulement du programme (d'où le nom de variable...).

En pratique, une variable est désignée par son **nom** et la donnée qu'elle contient s'appelle sa **valeur**.

Dans le langage Python, un nom de variable doit respecter la syntaxe suivante :

- Le nom de la variable ne peut être composé que de lettres minuscules ou majuscules (non accentuées), de chiffres et du symbole souligné "_" (appelé underscore en anglais). Ainsi, un nom de variable ne peut être composé que par des caractères appartenant à l'ensemble {a, ..., z, A,..., Z, 0, 1, ..., 9,_ }.
- Le nom de la variable ne peut pas commencer par un chiffre.
- Le langage python est sensible à la casse, ce qui signifie qu'il fait la différence entre majuscules et minuscules : **AGE**, **aGe** ou **age** sont trois noms différents.

Pour affecter une valeur à une variable, on utilise le symbole " = " : ce procédé s'appelle **affectation**.

Prenons par exemple une variable appelée **age** et décidons de lui donner la valeur 19 : en langage informatiqur, on dira qu'on affecte la valeur 19 à la variable **age**. En python on écrira : age = 19. De façon générale, le procédé d'affectation d'une valeur à une variable s'écrit :

nom_de_la_variable = valeur

In [1] : age = 19
In [2] :

Ici, vous pourrez constater que Python n'affiche pas de **Out [1] :** : c'est normal, il n'a rien à calculer...et il considère qu'une simple affectation ne justifie pas d'afficher quoi que ce soit ! Python passe donc tranquillement à la ligne suivante.

On peut cependant demander l'affichage du contenu de la variable **age** simplement en tapant son nom :

In [2] : age
Out [2] : 19

Vous remarquez que Python *crée* la variable **age** lorsqu'il exécute pour la première fois une instruction d'affectation de celle-ci. À ce moment, il réserve un emplacement dans la mémoire de l'ordinateur et y dépose le contenu de la variable : 19.

On peut ensuite faire des calculs en utilisant le nom de la variable à la place de son contenu.

In [3] : age + 3
Out [3] : 21

On peut aussi créer d'autres variables à partir de variables déjà connues :

In [4] : Grand_age = age + 50
In [5] : Grand_age
Out [5] : 69

Bien entendu, la valeur d'une variable peut être changée au cours du déroulement du programme : on dit que la valeur est **écrasée** ou encore que la variable est **ré-affectée**. Notez bien qu'une ré-affectation ne re-crée pas la variable : cela change seulement son contenu. Par exemple :

In [6] : age = 19
In [7] : age = 21.5
In [7] : age
Out [7] : 21.5

On peut ré-affecter une valeur à une variable déjà créée en utilisant le nom de cette même variable (auto-référencement) :

In [8] : age = age - 6
In [9] : age
Out [9] : 15.5

Au cours de cette opération, Python extrait la valeur de **age**, lui soustrait 6 et, finalement, place la nouvelle valeur obtenue dans **age** : il s'agit donc bien d'une ré-affectation.

1.3 Le mot clé del

Vous pouvez détruire une variable (déjà créée) à l'aide du mot-clé **del**.

In [10] : a = 36
In [11] : a
Out [11] : 36

In [12] : del a
In [13] : a

NameError
<ipython-input-23-60b725f10c9c> in <module>()
—-> 1 a
NameError : name 'a' is not defined

La variable **a** a été détruite et Python ne la connaît donc plus !

1.4 Types de données

En informatique, les **valeurs** sont des données qui sont caractérisées par leur **type**. Ces valeurs peuvent être affectées à des variables. Python travaille avec plusieurs types. Nous allons commencer par décrire 5 types importants, à savoir :

- Les entiers : type **int**
- les réels : type **float**
- les complexes : **complex**
- les booléens : type **bool**
- les chaînes de caractères : type **str**
- le type **tuple**
- les listes : type **list**

Ce ne sont pas les seuls types de valeurs gérées par Python : nous verrons par la suite les types **dict** (dictionnaire), **function** (fonction), etc...

Lorsque vous manipulez une valeur, vous pouvez toujours voir son type grâce à l'instruction : **type**(*valeur*).

In [1] : type(236)
Out [1] : int

In [2] : type(2.345e5)
Out [2] : float

Lorsqu'une **valeur** est affectée à une variable, nommée **a** pour notre exemple, cette variable prend le même type que la valeur qu'elle contient. Pour le voir, on peut utiliser l'instruction **type**(*nom_de_la_variable*) :

In [3] : a = 236
In [4] : type(a)
Out [4] : int

Rien ne vous empêche par la suite de ré-affecter à **a** une valeur d'un autre type au cours du déroulement du programme. Python s'adapte au fur et à mesure au type de la valeur : on dit que le *typage est dynamique*. Par exemple :

In [5] : a = -123
In [6] : type(a)
Out [6] : int
In [7] : a = 2.3456e5
In [8] : type(a)
Out [8] : float

Passons maintenant en revue les caractéristiques de ces types de données.

1.4.1 Les entiers : int

En Python 3, une valeur entière $k \in \mathbb{Z}$ est du type **int**. Sa taille peut être en théorie illimitée [1] et, en pratique, la seule limitation correspond à la taille de la mémoire attribuée à Python par le système !

1. Dans les versions antérieures de Python, il y avait deux types d'entiers, les **int** (entiers courts) et les **long** (entiers de taille illimitée)

Quel est le principe du codage d'un entier ?

Selon la taille de celui-ci, Python réserve une quantité plus ou moins importante de mémoire (32 bits, 64 bits ou plus).

Supposons pour simplifier que ce codage se fasse sur 8 bits, du genre : 01111001. Le bit situé à l'extrême gauche ("0" ici) est appelé bit de *poids fort* : c'est lui qui code le signe de l'entier.

- Pour un entier positif, ce bit est toujours 0. Le plus grand entier positif que l'on pourra coder ainsi sera donc 01111111, c'est à dire en base dix :

$$k_{max} = 1 \times 2^0 + 1 \times 2^1 + ... + 1 \times 2^6 = 2^7 - 1 = 127$$

  De façon générale, le plus grand entier positif codé sur N bits a pour valeur $k_{max} = 2^{N-1} - 1$. Par exemple, avec $N = 32$, cela donne : $k_{max} = 2^{31} - 1 = 2\ 147\ 483\ 647$.

- Passons maintenant aux entiers négatifs. Dans ce cas, le bit de poids fort est nécessairement "1". Cependant, la façon de coder est un peu spéciale et se fait ***en complément à deux***. Comment çà marche ? Prenons l'exemple de -5 avec un codage sur 8 bits :

  * On commence par coder son opposé + 5 : 00000101
  * On complémente à deux, c'est à dire qu'on inverse tous les bits : les "0" deviennent des "1" et les "1" des "0" : 11111010
  * On ajoute 1 : 11111010 + 1 = 11111011

  Le codage de - 5 dans la machine est donc 11111011 et le bit de poids fort (bit de signe) est bien égal à 1. La particularité de ce codage est qu'on retrouve bien 5 + (-5) = 0 y compris dans l'écriture binaire. En effet, en se rappelant que, au niveau de l'addition des bits, on a : 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1 et

$1 + 1 = 0$ avec une retenue de 1, on obtient :

$$\begin{array}{cc} & 00000101 \\ + & 11111011 \\ & -------- \\ = & 00000000 \end{array}$$

avec une retenue de 1 sur le bit de poids fort, dont on ne tient pas compte. Avec ce type de codage, le plus petit entier négatif que l'on peut obtenir sur 8 bits est $-2^7 = -128$ et, de façon plus générale $k_{min} = -2^{N-1}$ sur N bits. Avec $N = 32$, cela donne : $k_{min} = -2\,147\,483\,648$.

1.4.2 Les réels : float

En Python, une valeur réelle est de type **float**. Ce type permet de gérer tous les nombres à virgule, par exemple 2.365 ou -1.34e-56. Même lorsque la virgule n'est pas explicite, comme par exemple avec 4/5, le type est **float**.

In [1] : a = 4/5
In [2] : type(a)
Out [2] : float

Python code ces données sur 64 bits : c'est ce qu'on appelle un format double (un format simple étant codé sur 32 bits). Reportez vous à l'annexe pour le principe général de ce codage : ce principe est un des points abordés dans le programme d'informatique de MPSI.

1.4.3 Les nombres complexes : complex

Un nombre complexe est du type **complex**. Python les code comme un couple de nombres réels qui représentent sa partie réelle et sa partie imaginaire. La syntaxe utilisée est :

$$a + b\text{j}$$

où a est la partie réelle et b la partie imaginaire du nombre complexe. Remarquez **qu'il faut coller** b et j : bj et non pas b j ou j b, ce qui génèrerait une erreur (essayez pour voir). Le nombre imaginaire pur " i " des mathématiques s'écrit : 1j et pas j.

En revanche, vous pouvez mettre des espaces ailleurs : 1+2j, 1 + 2j ou 1 +   2j sont acceptés.

On peut ensuite additionner, soustraire, multiplier et diviser deux nombres complexes. Ces opérations sont décrites par les symboles usuels : " + ", " − ", " * " et " / ". Bien entendu, ces valeurs complexes peuvent être affectées à des variables :

In [1] : c1 = 1 + 2j
In [2] : c2 = 2 + 3j
In [3] : c1 + c2
Out [3] : (3 + 5j)
In [4] : c1 / c2
Out [4] : (0.6153846153846154+0.07692307692307691j)

Une fois définie une variable nommée **c** de type **complex**, on peut accéder à sa partie réelle et à sa partie imaginaire grâce aux instructions **c.real** et **c.imag**. La syntaxe est :

nom_variable.real   ou   nom_variable.imag

In [5] : nombreC = 1.03 − 2.4j
In [6] : nombreC.real
Out [6] : 1.03
In [7] : PartieIm = nombreC.imag
In [8] : PartieIm
Out [8] : − 2.4

1.4.4 Les booléens : bool

Une valeur de type **bool** ne peut prendre que l'une des deux expressions suivantes : **True** ou **False**. Ces deux mots sont des *mots réservés* du langage Python qui signifient "vrai" ou "faux". Remarquez que ces deux mots commencent par une **majuscule**.

Dans la machine, "True" est codé comme le nombre entier 1 et "False" comme le nombre entier 0.

Nous reparlerons des booléens lorsque nous étudierons les **instructions conditionnelles** et les boucles **while**.

1.4.5 Les chaînes de caractères : str

Une *chaîne de caractères* permet de stocker des lettres : c'est une valeur de type **str** qui représente un mot ou une phrase. En Python, on peut écrire une chaîne de caractères de trois façons :

- entre deux guillemets : "Bonjour tout le monde !"
- entre deux apostrophes : 'Bonjour tout le monde !'
- entre 3×2 guillemets : """Bonjour tout le monde !"""

In [1] : type("Bonjour tout le monde !")
Out [1] : str

Ces trois façons d'écrire sont équivalentes. Pourquoi multiplier les façons d'écrire une chaîne de caractère ? Eh bien, tout simplement pour pouvoir écrire des phrases du style :

J'aime le chocolat

ou encore

Monsieur Jacques m'a dit : "Taisez-vous !".

On voit que dans ces exemples, l'apostrophe ou les guillemets sont déjà utilisés en tant que caractères à l'intérieur de la chaîne : on ne peut donc plus les utiliser comme **délimiteurs** de chaîne. Dans l'exemple 1, on pourra écrire en Python :

"J'aime le chocolat" ou encore """J'aime le chocolat"""

et dans l'exemple 2, on n'aura pas le choix et il faudra écrire :

"""Monsieur Jacques m'a dit : "Taisez-vous !"."""

On peut toutefois quand même utiliser des apostrophes ou des guillemets comme délimiteurs de chaîne même si celle-ci en contient déjà à condition d'utiliser le **caractère d'échappement** " \" (caractère "anti-slash"). Reprenons les deux exemples précédents : Python accepte l'écriture :

'J\'aime le chocolat' ou "Monsieur Jacques m'a dit : \"Taisez-vous !\"."

On voit dans ces deux cas comment indiquer à Python qu'une apostrophe (') ou un guillemet (") constitue un des caractères de la chaîne et pas le délimiteur : il suffit de placer le signe \devant ce caractère. Cependant, si on ne veut pas s'embêter avec çà, on peut toujours utiliser une des trois formes possibles décrites plus haut !

Tout comme les autres valeurs déjà vues, la valeur chaîne de caractère peut être affectée à une variable :

In [2] : Ma_chaine = "Bonjour !"
In [3] : Ma_chaine
Out [3] : 'Bonjour !'

Remarques :

- Une chaîne de caractères peut être *vide* et on l'écrit "" ou '' : dans ce cas, elle ne contient rien. Attention, il faut **coller** les deux guillemets ou apostrophes pour ne pas confondre avec ' ' qui est une chaîne qui contient 1 caractère : le caractère Espace.

- On peut aussi avoir des chaînes de caractère à un seul élément, par exemple "a" ou 'ù'.

Nombre de caractères dans une chaîne : le mot-clé **len**

Vous pouvez connaître le nombre de caractères dans une chaîne sans avoir à les compter : il vous faut simplement écrire **len**(nom_chaine) :

In [**4**] **:** Ma_chaine = "Bonjour"
In [**5**] **: len**(Ma_chaine)
Out [**5**] **:** 7

ou encore plus simplement, en utilisant directement la valeur de la chaîne :

In [**6**] **: len**("Bonjour")
Out [**6**] **:** 7

Chaque caractère contenu dans une chaîne de N caractères est caractérisé par son **rang** (encore appelé **index** en anglais) : il s'agit d'un numéro compris entre 0 et $N-1$. Le caractère le plus à gauche a le **rang** 0, le suivant le **rang** 1, etc... et le dernier caractère de la chaîne possède le **rang** $N-1$.

rang	0	1	2	3	4	5	6
chaîne	B	o	n	j	o	u	r

On peut accéder individuellement à chacun des caractères d'une chaîne en écrivant **nom_chaine**[*rang*] (rang écrit entre deux crochets) :

In [**7**] **:** Nom = "Dalila"
In [**8**] **:** Nom[0]
Out [**8**] **:** 'D'
In [**9**] **:** Nom[2]
Out [**9**] **:** 'l'

On peut aussi écrire directement :

In [**10**] **:** "Dalila"[0]
Out [**10**] **:** 'D'

Une écriture comme "Dalila"[0] étant une **valeur**, on peut l'affecter à une variable :

In [**11**] **:** Lettre = "Dalila"[0]
In [**12**] **:** Lettre
Out [**12**] **:** 'D'

Par la suite, je n'utiliserai pas cette syntaxe avec la valeur de la chaîne de caractères. Je préfèrerai utiliser une variable pour stocker la valeur de la chaîne et écrire par exemple : **Nom** = "Dalila", puis **Nom**[0], etc...

En revanche, il n'est pas possible de modifier un caractère d'une chaîne par cette méthode ! Essayez cela pour voir :

In [**13**] **:** Nom = 'Papa'
In [**14**] **:** Nom[1] = 'o'

TypeError

<ipython-input-51-0df53d28322d> in <module>()
—-> 1 Nom[1] = 'o'
TypeError : 'str' object does not support item assignment

Un avertissement important en ce qui concerne les manipulations de chaînes : faites attention car pour Python, le premier caractère d'une chaîne de N lettres porte toujours le numéro 0 et le dernier porte le numéro $N-1$. C'est une source d'erreur fréquente.

Si vous n'avez pas envie de compter jusqu'au dernier caractère, sachez que Nom[-1] affiche le dernier caractère de la chaîne **Nom** :

In [15] : Nom = "Bonjour !"
In [16] : Nom[-1]
Out [16] : ' !'

Que se passe-t-il si vous cherchez à accéder à un caractère au delà de la fin de la chaîne ? Essayez !

In [17] : Nom = "Dalila"
In [18] : Nom[15]

IndexError
<ipython-input-48-f75c80e800de> in <module>()
—-> 1 Nom[15]

IndexError : string index out of range

Python n'accepte pas ! Il vous indique une erreur de dépassement d'index de chaîne.

Parcours de chaîne de caractère
Il est possible d'accéder à une sous-chaîne contenue à l'intérieur d'une chaîne au moyen de l'instruction :

nom_chaine[n_1 : n_2]

Cela va créer une sous-chaîne dont le premier caractère est le numéro n_1 et dont le dernier caractère est le numéro $n_2 - 1$ (**attention !**) de la chaîne initiale.

Par exemple, si Nom = "Noémie", alors Nom[2 : 5] est "émi" : c'est la sous-chaîne dont le premier caractère a le **rang** 2 ("é") et le dernier le **rang** 4 (c'est à dire "i") dans Nom.

Voyons un autre exemple :

In [13] : Ma_chaine = "Bonjour tout le monde !"
In [14] : Nouvelle_chaine = Ma_chaine[0 : 7]
In [15] : Nouvelle_chaine
Out [15] : 'Bonjour'

Nouvelle_chaine contient les caractère numéros 0 à 6 de Ma_chaine, c'est à dire 'Bonjour'.

Deux cas particuliers :

- **nom_chaine**[: n] est une sous - chaîne qui contient les caractères de **nom_chaine** dont les **rangs** sont compris entre 0 et $n-1$.

- **nom_chaine**[n :] est une sous - chaîne qui contient tous les caractères de **nom_chaine** à partir du **rang** n.

In [16] : ch = "Hello World"
In [17] : ch[: 5]
Out [17] : 'Hello'
In [18] : ch[6 :]
Out [18] : 'World'

Une syntaxe encore plus complète pour l'extraction de sous - chaînes est :

$$\textbf{nom_chaine}[n_1 : n_2 : \textit{pas}]$$

ce qui donne une sous-chaîne formée des caractères de **rang** n_1 à $n_2 - 1$, en sautant *pas* caractères à chaque fois.

In [19] : ch[6 : : 2]
Out [19] : 'Wrd'

On extrait ici les caractères de "Hello World" du **rang** 6 jusqu'à la fin de la chaîne, avec un *pas* de 2.

Concaténation de deux chaînes de caractères

Avant de passer à la suite, voyons une autre opération sur les valeurs de type **str** : la **concaténation**. C'est une opération qui consiste à *fusionner* deux chaînes de caractères l'une derrière l'autre. Elle se réalise à l'aide de l'opérateur addition " + ".

In [20] : ch1 = "Bonjour "
In [21] : ch2 = "Dalila"
In [22] : ch1 + ch2
Out [22] : 'Bonjour Dalila'

ch1 + ch2 est la **valeur** de la chaîne *concaténée* : on peut donc aussi l'affecter à une variable et écrire :

In [23] : ch3 = ch1+ ch2
In [24] : ch3
Out [24] : 'Bonjour Dalila'

Les deux chaînes sont vraiment *collées* l'une à l'autre, donc attention aux espaces de séparation.

1.4.6 Le type tuple

Le langage Python dispose d'un type de données appelé **tuple**. Il s'agit d'une **collection d'éléments**. Une valeur de type **tuple** s'écrit en plaçant les différents éléments de la collection, séparés les uns des autres par des virgules, entre deux parenthèses. Voyons cela sur un exemple :

In [1] : a = (1, 2.35, "arbre", 's')
In [2] : type(a)
Out [2] : tuple

En [1], on crée une variable **a** et on lui affecte une valeur de type **tuple**. 1, 2.35, "arbre" et 's' sont les *éléments de cette collection.*

Dans cette collection, chaque élément possède un **rang** (on dit aussi un **index**) : le **rang** de l'élément le plus à gauche est 0, celui de l'élément suivant est 1, et ainsi de suite...

rang		0		1		2		3	
Éléments	(	1	,	2.35	,	"arbre"	,	's'	)

Les éléments peuvent avoir des *types différents* : cela ne pose aucun problème !

Enfin, on peut accéder aux différents éléments de la collection grâce à leur **rang**, en écrivant par exemple :

In [3] : a[0]
Out [3] : 1
In [4] : a[2]
Out [4] : 'arbre'

C'est donc simple, lorsqu'une valeur de type **tuple** est affectée à une variable nommée **a** par exemple, on accède à l'élément de **rang** n en écrivant a[n].

Le problème avec ce type de donnée est qu'une fois créée on ne peut pas la modifier ! Essayez cela pour voir :

In [**5**] **:** a[0] = 345

TypeError
<ipython-input-7-0584e945bc6e> in <module>()
—-> 1 a[0] = 345

TypeError : 'tuple' object does not support item assignment

Python n'accepte pas du tout ce genre de manipulation : une fois le "tuple" créé, il est impossible de modifier un de ses éléments par une affectaction.

À quoi sert un "tuple" alors ? En pratique, on peut s'en servir pour réaliser une **affectation multiple**. Supposez que vous vouliez affecter "arbre" à une variable **Mot** et 23 à une variable **b**. Au lieu de réaliser l'affectation séparément, vous pouvez très bien utiliser la syntaxe suivante :

In [**6**] **:** (Mot, b) = ("arbre", 23)

Cela crée d'un seul coup deux variables **Mot** et **b** que l'on peut manipuler séparément par la suite. Ces variables sont tout à fait normales et vous pouvez faire des calculs avec, leur affecter d'autres valeurs, etc...

In [**7**] **:** Mot
Out [**7**] **:** 'arbre'
In [**8**] **:** b
Out [**8**] **:** 23
In [**9**] **:** Mot = 'cheval'
In [**10**] **:** Mot
Out [**10**] **:** 'cheval'
In [**11**] **:** b = b + 1
In [**12**] **:** b
Out [**12**] **:** 24

Mais vous venez de dire qu'on ne pouvait pas modifier un tuple ! C'est vrai, et c'est un point délicat à comprendre...mais ce n'est pas la même chose d'écrire **a** = ('arbre', 23) et (**Mot**, **b**) = ('arbre', 23).

- Dans le premier cas, on crée une variable **a** de type **tuple** et ses éléments *ne sont pas modifiables.* **a**[0] = 'cheval' n'est pas possible !

- Dans le second cas *on utilise une syntaxe autorisée par Python* pour affecter simultanément deux variables **Mot** et **b**, cette syntaxe s'écrivant :
  (**Mot**, **b**) = valeur de type **tuple**
  Cela ne veut pas dire qu'on a créé une variable **tuple** dont la valeur serait (**Mot**, **b**).

Une dernière chose : Python étant très souple sur ce point, vous pouvez même réaliser une affectation multiple en omettant les parenthèses ! Python comprendra tout seul la syntaxe (les parenthèses des tuples sont ici implicites et sous-entendues) :

In [**13**] **:** Mot, b = 'arbre', 23
In [**14**] **:** Mot
Out [**14**] **:** 'arbre'
In [**15**] **:** b
Out [**15**] **:** 23

Nous reverrons encore une fois les valeurs de type **tuple** lorsque nous aborderons les **fonctions** renvoyant plusieurs valeurs.

1.4.7 Les listes : list

Dans le langage Python, les *listes* sont des données fondamentales. Tout comme les "tuples", les listes sont des collections d'éléments, mais contrairement aux "tuple", les éléments d'une liste sont **modifiables**. Une liste est une valeur de type **list**. On l'écrit en précisant tous ses éléments, séparés les uns des autres par une virgule, entre *deux crochets*.

Cela donne par exemple :

In [1] : Ma_liste = [1, -2.65e-7, "Julie", 1+7j]
In [2] : Ma_liste
Out [2] : [1, -2.65e-07, 'Julie', (1+7j)]
In [3] : type(Ma_liste)
Out [3] : list

En **In [1]**, on crée une variable nommée **Ma_liste** et on lui affecte la valeur [1, -2.65e-7, 'Julie', 1+7j] qui est une valeur de type **list**.

- Vous pouvez faire des listes de toute longueur.
- Les listes peuvent contenir n'importe quel type d'éléments.

Tous comme dans les valeurs de type **tuple**, chaque élément d'une liste possède un **rang** (ou un **index**) : le premier élément de la liste (à gauche) a le rang 0, le suivant a le rang 1, etc...

rang		0		1		2		3	
Élément	[	1	,	-2.65e-7	,	'Julie'	,	1+7j	]

On peut alors accéder à chaque élément au moyen de son **rang** et on peut le modifier aussi ! (contrairement aux tuples)

In [4] : Ma_liste[2]
Out [4] : 'Julie'

In [5] : Ma_liste[3] = "Jacques"
In [6] : Ma_liste
Out [6] : [1, -2.65e-07, 'Julie', 'Jacques']

On peut aussi accéder directement aux éléments d'une liste en écrivant :

In [7] : [1, -2.65e-7, 'Julie', 'Jacques'][2]
Out [7] : 'Julie'

C'est le même type d'écriture que celui que nous avions utilisé avec les chaînes de caractères quand nous écrivions par exemple "Dalila"[0]. Par la suite, nous n'utiliserons plus cette syntaxe et nous affecterons la valeur de type **list** à une variable. Nous écrirons donc plutôt des choses du style :

Nom_variable = valeur type **list**
puis Nom_variable[1] par exemple

Passons à la suite ! On peut aussi utiliser des variables pour définir les éléments d'une liste. La syntaxe ci-dessous est tout à fait permise :

In [8] : a = 1
In [9] : b = 4
In [10] : c = "Sacha"
In [11] : Liste1 = [a, b, c]
In [12] : Liste1
Out [12] : [1, 4, 'Sacha']

Vous pouvez aussi utiliser une variable pour modifier un élément d'une liste : la valeur de cette variable sera affectée à l'élément :

In [13] : Nom = "Jules"
In [14] : Liste1[0] = Nom

In [15] : Liste1
Out [15] : ['Jules', 4, 'Sacha']

Dans **In [14]** on affecte la valeur de la variable **Nom** (c'est à dire "Jules") à l'élément de **rang** 0 de Liste1 (qui était jusqu'à présent 1).

Un même élément peut se trouver plusieurs fois dans la même liste : cela ne pose aucun problème.

In [16] : ListeNoms = ["Julie", "Paul", "Paul", "Monica"]
In [17] : ListeNoms[1]
Out [17] : 'Paul'
In [18] : ListeNoms[2]
Out [18] : 'Paul'

Les deux éléments de valeur 'Paul' sont bien différentiés par leurs **rangs** qui sont différents.

Comme les éléments d'une liste sont de type quelconque, un élément d'une liste peut être lui-même une liste : on peut donc créer des **listes de listes**. Voyons un exemple :

In [19] : Liste = [[1, 2] , [5, 9]]
In [20] : type(Liste)
Out [20] : list

Liste est une variable de type **list** qui contient deux éléments qui sont eux-mêmes des listes. Dans ce cas, Liste[0] est la liste [1,2] et Liste[1] est la liste [5, 9]. Essayez :

In [21] : Liste[0]
Out [21] : [1, 2]
In [22] : Liste[1]
Out [22] : [5, 9]

Comment faire pour accéder à l'entier 5 qui est le premier élément de Liste[1] ? Eh bien, comme d'habitude, au moyen de son **rang** en écrivant Liste[1][0] : on accède alors à l'élément de **rang** 0 de la liste Liste[1]. Essayons !

In [23] : Liste[1][0]
Out [23] : 5

Sachez que les **listes de listes** sont très utilisées en Python pour modéliser des **matrices** $n \times p$. Par exemple, la matrice :

$$\mathrm{A} = \begin{pmatrix} 1 & 4 & 7 & 2 \\ -3 & 6 & 2 & 5 \\ 1 & 9 & 8 & 4 \end{pmatrix}$$

est écrite sous la forme A = [[1, 4, 7, 2], [−3, 6, 2, 5], [1, 9, 8, 4]].

Le premier élément de A est la première ligne de la matrice, écrite sous forme d'une liste, le second élément de A est la deuxième ligne, etc...

On accède alors à l'élément A_{ij} de la matrice A en écrivant : A[i][j]. En effet [2], A[i] est une liste qui est la ligne de **rang** i et A[i][j] est l'élément de **rang** j dans A[i]. Par exemple A[1][0] sera −3

Nombre d'éléments d'une liste : le mot-clé **len**

Vous pouvez toujours connaître le nombre des éléments d'une liste en tapant **len**(nom_liste) :

In [24] : Ma_liste = [3, 2.4, "pomme", 18, 'r']
In [25] : **len**(Ma_liste)

2. Il faut cependant toujours faire attention au numéro du rang. Python commence toujours à 0.

Out [25] : 5

Parcours de liste

Comme pour les chaînes de caractères, il est possible d'extraire une sous-liste à partir d'une liste. Il faut utiliser l'instruction :

nom_liste[n_1 : n_2 : *pas*]

Cela va créer une sous liste dont le premier élément était de rang n_1 et le dernier élément était de rang $n_2 - 1$ (attention !) dans **nom_liste**, en sautant *pas* éléments à chaque fois. Par défaut *pas* = 1 : ce sera sa valeur implicite si vous ne l'écrivez pas, en tapant simplement **nom_liste**[n_1 : n_2].

Par exemple :

In [26] : L1 = [1, 5, "cheval", "chameau", 4]
In [27] : L2 = L1[1 : 3]
In [28] : L2
Out [28] : [5, "cheval"]
In [29] : L3 = L1[0 : 4 : 2]
In [30] : L3
Out [30] : [1, 'cheval']

Cas particuliers :

- nom_liste[n_1 :] est une sous-liste avec les éléments situés à partir du **rang** n_1 jusqu'à la fin de nom_liste.

- De même, nom_liste[: n_2] est une sous-liste constituée des éléments du **rang** 0 jusqu'au **rang** $n_2 - 1$ de nom_liste.

In [31] : L4 = L1[2 :]
In [32] : L4
Out [32] : ["cheval", "chameau", 4]
In [33] : L5 = L1[: 4]
In [34] : L5
Out [34] : [1, 5, "cheval", "chameau"]

Opération de concaténation de deux listes

Deux listes, appelons-les **Liste1** et **Liste2**, peuvent être **concaténées**, comme les chaînes de caractères. Cette opération s'effectue à l'aide du symbole "+ ". Quand nous faisons **L** = **Liste1** + **Liste2**, **L** est une liste qui contient d'abord les éléments de **Liste1**, puis ceux de **Liste2**. Voyons un exemple :

In [35] : Liste1 = [1, 2, 3]
In [36] : Liste2 = [4, 5, 6, 7]
In [37] : L = Liste1 + Liste2
In [38] : L
Out [38] : [1, 2, 3, 4, 5, 6, 7]

Destruction d'un élément d'une liste : le mot-clé **del**

Nous avons déjà vu le mot-clé **del** lorsqu'on a parlé des variables à la section 1.2 : ce mot **détruit** une variable.

Cela fonctionne de la même façon pour les éléments d'une liste : vous pouvez détruire n'importe lequel de ses éléments à l'aide du mot-clé **del**. Vous détruisez l'élément en y accédant grâce à son **rang** dans la liste :

In [39] : Liste_fruits = ["pomme", "couteau", "poire"]
In [40] : Liste_fruits
Out [40] : ['pomme', 'couteau', 'poire']
In [41] : del Liste_fruits[1]
In [42] : Liste_fruits
Out [42] : ['pomme', 'poire']

En **In** [**41**] nous détruisons l'élément de rang 1 de Liste_fruits, c'est à dire "couteau". Python réorganise alors automatiquement la liste[3].

La **liste vide** s'écrit [] : deux crochets sans rien entre (attention, même pas d'espace, les deux crochets ouvrant et fermant sont collés). Une liste vide ne contient *aucun élément* !

In [**43**] **: del** Liste_fruits[1]
In [**44**] **:** Liste_fruits
Out [**44**] **:** ['pomme']
In [**45**] **: del** Liste_fruits[0]
In [**46**] **:** Liste_fruits
Out [**46**] **:** []

Après cela, on ne peut plus continuer à détruire : il n'y a d'ailleurs plus rien à détruire...(essayez pour voir).

On peut aussi ajouter des éléments à une liste mais on verra cela plus tard, à la section **5 Les méthodes d'objets**.

1.5 Écrivons le programme dans un fichier !

L'interpréteur c'est bien, mais ce n'est pas du tout pratique quand on veut écrire un long programme, sans l'exécuter ligne après ligne. Il vaut mieux alors écrire directement le programme dans un fichier. Pour cela :

- Écrivez le programme dans la **zone de saisie dans un fichier** de Pyzo (voir page 3). Les lignes sont alors numérotées automatiquement.
- Enregistrez votre fichier à l'aide du menu *Fichier* > *Enregistrer*. Vous pourrez choisir le nom du fichier ainsi que le répertoire dans lequel il sera placé. Cela va générer un fichier avec l'extension .py
- Lancez l'exécution du programme à l'aide du menu *Exécuter* > *Exécuter le fichier*. Le programme est exécuté en bloc et les éventuels résultats s'affichent dans l'interpréteur.

3. Attention ! Si vous tapez **del** Liste_fruits, vous détruirez la variable Liste_fruits d'un seul coup, avec tous ses éléments.

Cependant, si vous voulez afficher le contenu d'une variable, vous ne pourrez plus vous contenter de taper son nom...car ici, rien ne se passera. Il vous faudra maintenant utiliser la fonction **print**.

—> La fonction **print**

print est une fonction qui permet d'afficher sur l'écran de l'ordinateur des valeurs ou le contenu de variables, avec une mise en forme si nécessaire.

Essayez en tapant (pas besoin de numéroter les ligne, c'est automatique) :

1 a = 345
2 **print**(a)
3

Enregistrez votre fichier en lui donnant par exemple le nom Mon_programme.py et lancez son exécution. Vous verrez le résultat s'afficher dans l'interpréteur de la manière suivante :

In [**1**] **:** (executing lines 1 to 2 of "Mon_programme.py")
345

Avec la fonction **print**, vous pouvez afficher plusieurs valeurs sur la même ligne. Il suffit par exemple d'écrire :

1 age = 19

2 **print**("J'ai", age, "ans")
3

ce qui devrait donner :

In [1] : (executing lines 1 to 2 of "Mon_programme.py")
J'ai 19 ans

Les différentes valeurs (ou les contenus des variables), séparées par une virgule, sont affichées les unes derrière les autres et Python laisse un espace entre chaque valeur (il n'est donc même pas nécessaire d'inclure des espaces au début ou à la fin des chaînes de caractères pour gérer la séparation des mots).

La fonction **print** peut afficher tout type de valeurs : des listes, des complexes, des booléens, etc... Essayez ceci :

```
1   L = [ 'a', 'b', 1, 6 ]
2   c = 1 + 9j
3   print(L, c, "et voilà !")
```

Exécution du programme :

In [1] : (executing lines 1 to 3 of "Mon_programme.py")
['a', 'b', 1, 6] (1+9j) et voilà !

—> La fonction **input**

Il se peut que vous soyez amenés à demander à l'utilisateur de faire une saisie au clavier : votre programme va devoir capter cette saisie afin de pouvoir l'utiliser. Pas de problèmes : la fonction **input** gère cela. La syntaxe est :

input(chaine_affichee)

où chaine_affichee est une chaîne de caractères qui s'affiche à l'écran, dans l'interpréteur. Vous pouvez ensuite taper votre nom (toujours dans l'interpréteur) et la fonction **input** renvoie cette chaîne de caractères saisie, que vous pourrez sauvegarder dans une variable. Prenons un exemple :

1 nom = **input**("Veuillez entrer votre nom : ")
2 **print**("Vous vous appelez", nom)

À la ligne 1, on demande à l'utilisateur de taper son nom sur le clavier. Le résultat forme une chaîne de caractères que l'on affecte à la variable **nom** pour la sauvegarder. Ligne 2, on affiche la phrase "Vous vous appelez" suivie du nom qui vient d'être entré, grâce à la fonction **print**. Voici le résultat de l'exécution :

In [1] : (executing lines 1 to 2 of "Mon_programme.py")
Veuillez introduire votre nom :

Vous tapez alors tranquillement ce que vous voulez...

Veuillez introduire votre nom : Personne

Vous appuyez ensuite sur la touche "Entrée" de votre clavier et voici ce qui est affiché à la fin du processus :

In [1] : (executing lines 1 to 2 of "Mon_programme.py")
Veuillez introduire votre nom : Personne
Vous vous appelez Personne

1.6 Le transtypage

Notez que la valeur renvoyée par la fonction **input** est toujours de type **str**. C'est une chaîne de caractères, même si vous demandez à l'utilisateur de rentrer un nombre ! Dans ce cas, le nombre sera

transformé en chaîne de caractères, avec ses différents chiffres en tant que lettres. Essayez pour voir :

```
1 k = input("Veuillez entrer un nombre entier : ")
2 print(type(k))
```

À la ligne 1, on demande de saisir un nombre entier que l'on sauvegarde dans la variable **k**. On affiche ensuite le type de cette variable : souvenez-vous que si vous écrivez le programme dans un fichier, vous ne pouvez pas vous contenter d'écrire **type**(k) pour afficher le type de **k** ... cela ne marchera pas ! Il faut utiliser **print** ! Voici ce que cela donne :

```
In [1] : (executing lines 1 to 2 of "Mon_programme.py")
Veuillez entrer un nombre entier : 345
<class 'str'>
```

Python affiche le type : 'str' (il utilise aussi le mot *class* dont on verra la signification plus tard). Il s'agit bien d'une chaîne de caractères !

Comment faire pour réaliser des calculs avec **k** ? Il faut d'abord convertir la valeur de cette variable en entier en utilisant le **transtypage**. Pour cela, on écrit :

k1 = **int**(k) ou encore **k** = **int**(k)

ce qui signifie : "Prend le contenu de **k**, convertit-le en entier et place le résultat dans **k1** ou dans **k**".

Il est possible de transtyper toutes les valeurs que l'on veut dans la limite du bon sens : on peut transtyper une chaîne de caractère en entier ou en réel si celle-ci a le format d'un entier ou d'un réel. On peut aussi transtyper un entier en un réel et un réel en un complexe.

Cela ne marche pas toujours, souvent pour des raisons évidentes : Python génère alors une erreur.

Essayez par exemple, directement dans l'interpréteur :

```
In [1] : ch = "2.023e5"
In [2] : r = float(ch)
In [3] : r
Out [3] : 203000.0
```

Continuons :

```
In [4] : c = complex(r)
In [5] : c
Out [5] : (203000 + 0j)
```

Allons plus loin...

```
In [6] : ch2 = str(c)
In [7] : ch2
Out [7] : '(203000 + 0j)'
```

En revanche :

```
In [8] : ch = "Hello !"
In [9] : k = int(ch)

ValueError
...
—-> 1 k = int(ch)
ValueError : invalid literal for int() with base 10 : 'Hello'
```

Là non plus :

```
In [10] : c = 12.0 + 4j
```

In [11] : r = **float**(c)

TypeError
...
—-> 1 r = float(c)
TypeError : can't convert complex to float

En conclusion : là où vous ne sauriez pas faire la conversion, Python ne sait pas gérer non plus. Le transtypage est donc une question de bon sens.

1.7 Les commentaires

Terminons cette section par quelque chose qui est souvent négligé mais qui a une très grande importance lorsque vous écrivez votre programme dans un fichier : il s'agit des **commentaires**.

Un commentaire est une phrase précédée du signe " # " (dièse) et qui sert seulement à documenter votre programme. Toute ce qui suit le #, jusqu'à la fin de la ligne, ne sera pas exécuté. Ce commentaire servira simplement à vous rappeler pourquoi vous avez appelé votre variable **toutankamon** ou encore à quoi servent ces 32 lignes de code qui suivent. Cela peut être utile si vous laissez passer quelques jours ou semaines avant de reprendre votre programme et que vous vous dites : "Qu'est ce que j'ai bien pu écrire ici ?" ou encore si c'est une autre personne qui doit terminer votre programme.

Un exemple :

1#Ce programme demande d'entrer un nombre
2 #et affiche ce nombre +1
3 k = **input**("Entrez un nombre : ") #Saisie du nombre
4 a = **float**(k) + 1 #Ne pas oublier de transtyper !
5 **print**(a) #On affiche le nombre + 1

Vous pouvez aussi laisser des lignes vides dans votre programme : Python n'en tient pas compte et saute directement à la ligne suivante. Cela permet d'aérer le texte...Réécrivons donc le programme précédent, en l'aérant un peu :

1#Ce programme demande d'entrer un nombre
2 #et affiche ce nombre +1
3
4 k = **input**("Entrez un nombre : ") #Saisie du nombre
5 a = **float**(k) + 1 #Ne pas oublier de transtyper !
6
7 **print**(a) #On affiche le nombre + 1

En conclusion : les commentaires sont très utiles, donc ne les négligez pas...mais n'en faites pas trop non plus. Pensez aussi à aérer votre texte : il sera beaucoup plus agréable à lire.

2 Les structures conditionnelles

Les **structures conditionnelles** permettent d'exécuter un bloc d'instructions si une condition est vérifiée et un autre bloc d'instructions dans le cas contraire. Ce sont des structures très importantes dans les langages de programmation.

2.1 Différence entre Expression et Instruction

Parmi les notions qui doivent être développées dans le programme de MPSI, figurent celles d'**expression** et d'**instruction**.

- Une **expression** est une suite de signes qui ont **du sens** pour le langage de programmation utilisé et qui possède **une valeur**. Par exemple :

  2 + 4 est une expression. Sa valeur est 6

"Hel" + "lo" est une expression, de valeur "Hello"
4 > 3 est une expression booléenne de valeur **True**.

En revanche, 2 + "Papa" ou 1 < "Coucou" ou encore z"%tr""+2,(Ẑ ne sont pas des expressions car elles n'ont pas de sens et on ne peut leur donner aucune valeur : Python ne sait pas faire ! (mais peut être qu'un autre langage le saurait...).

L'interpréteur est un très bon détecteur d'expressions : quand vous tapez 2 + 4, il affiche directement la valeur de cette expression...et il génère une erreur dans le cas contraire. Essayez !

In [1] : 2 + 4
Out [1] : 6
In [2] : 4 > 3
Out [2] : True

In [3] : aert2.5
File "<ipython-input-21-ea4d0ec1afca>", line 1
aert2.5
^

SyntaxError : invalid syntax

In [4] : a > 3
NameError
...
—-> 1 a > 3
NameError : name 'a' is not defined

En revanche, vous pouvez toujours écrire :

In [5] : a = 1
In [6] : a > 3
Out [6] : False

Dans ce dernier cas, a > 3 devient une expression car **a** a été défini avant et Python sait lui donner une valeur.

- Une **instruction** est une *action dans un programme*. Pour détecter si on a affaire à une instruction, il faut se poser la question "Donne-t-on l'ordre au programme de faire quelques chose à l'aide de cette suite de signes ?". Par exemple :

  print("Bonjour") est une instruction. Son ordre : "affiche "Bonjour" sur l'écran".

  a = 1.2 est aussi une instruction : "prend la valeur 1.2 et mets-la dans la variable **a**".

  En revanche, si vous écrivez votre programme dans un fichier, et que vous tapez sur une ligne : 4 > 3 ce n'est pas une instruction car Python ne sait pas quoi faire de cette ligne de programmation : aucun ordre n'est donné ici (Rappelez-vous qu'écrire dans un fichier n'est pas la même chose qu'écrire directement dans l'interpréteur)

2.2 Expression booléenne

Revenons maintenant aux expressions. Parmi celle-ci, on appelle **expression booléenne** une expression dont la valeur peut être soit **True** (vrai), soit **False** (faux). Afin de générer des expressions booléennes, Python autorise l'utilisation des signes suivants :

- " == " : identique à
- " != " : différent de
- " > " : strictement supérieur à
- " < " : strictement inférieur à
- " <= " : inférieur ou égal à
- " >= " : supérieur ou égal à

On peut aussi utiliser des parenthèses ouvrantes ou fermantes pour clarifier les expressions booléennes. Par exemple (3.4 >= 2) est une expression booléenne dont la valeur est **True**.

De même, si on a pris soin de définir une variable **c** au préalable, c == 3 devient une expression booléenne qui vaut **True** si la valeur de **c** est égale à 3 et qui vaut **False** dans le cas contraire [4].

Les mots-clés and, or et not

Python dispose de trois mots-clés destinés à fabriquer des expressions booléennes. Dans ce qui suit, *Exp1* et *Exp2* sont deux expressions booléennes :

- **and** : équivalent à "et". *Exp1* **and** *Exp2* est une expression booléenne qui vaut **True** si et seulement si *Exp1* et *Exp2* valent **True** toutes les deux. Dans le cas contraire, elle vaut **False**.

- **or** : équivalent à "ou". *Exp1* **or** *Exp2* est une expression booléenne qui vaut **True** si au moins l'une des deux expressions *Exp1* ou *Exp2* vaut **True**. Dans le cas contraire, *Exp1* **or** *Exp2* vaut **False**.

- **not** : c'est la négation de l'expression. **not** *Exp1* vaut **False** si *Exp1* a pour valeur **True** et vice versa (**True** si *Exp1* vaut **False**).

2.3 Structure if

La structure conditionnelle la plus simple est la structure **if**. Elle s'écrit de la manière suivante :

4. Vous pouvez mettre des espaces si vous voulez, Python comprendra, mais vous ne pouvez pas séparer deux signes comme les deux == de "identique à". Par exemple (2 == 4) est correct mais (2 = = 4) ne l'est pas.

if *Expression booléenne* **:**
bloc d'instructions

Son fonctionnement est le suivant : si *Expression booléenne* a pour valeur **True**, le bloc d'instructions qui suit est exécuté. Sinon, le programme saute tout ce bloc d'instruction. Notez que vous devez *impérativement mettre deux points* " **:** " après l'expression booléenne et que vous devez mettre un espace devant chacune des instructions du bloc : on dit que les instructions doivent être **indentées** [5]. C'est la seule manière pour Python de voir quelles instructions font partie du bloc **if**.

Prenons un exemple. Dans ce qui suit, j'utiliserai les commentaires pour préciser le fonctionnement du programme. Écrivons donc les lignes suivantes dans un fichier, nommé Programme.py :

```
1  # Premier exemple de structure if
2  a = 5
3  if a > 0 : #Si a est strictement supérieur à 0
4     print(a, "est strictement positif")
```

Remarquez le **décalage vers la droite** de l'instruction **print**(a,"est strictement positif") : c'est cela l'indentation. Si vous ne décalez pas les instructions, Python ne reconnaîtra pas ce qu'il doit faire si la condition a > 0 est vraie, ou alors il génèrera une erreur.

Voici le résultat après lancement du programme par *Exécuter > Exécuter le fichier*.

```
In [1] : (executing lines 1 to 4 of "Programme.py")
5 est strictement positif
```

5. De toute façon, quand vous écrivez dans le fichier avec Pyzo, il indente automatiquement les instructions qui suivent les deux points :

Lorsque le bloc d'instructions qui est exécuté dans la **structure if** contient plusieurs instructions, il faut qu'elles aient **toutes la même indentation** (c'est à dire le même décalage vers la droite).

```
1 # Autre exemple de structure if
2 a = 5
3 b = 0
4 if (a > 0) and (a < 10) :
5   print(a, "appartient à l'intervalle ]0,10[")
6   b = b + 1 # augmenter b d'une unité
7   print("Le test d'appartenance a réussi", b, "fois")
```

En voici le résultat :

In [1] : (executing lines 1 to 7 of "Programme.py")
5 appartient à l'intervalle]0,10[
Le test d'appartenance a réussi 1 fois

2.4 La structure if...else

Passons à une structure plus complète : la **structure if - else**. Sa syntaxe est la suivante :

if *Expression booléenne* **:**
  bloc n°1 d'instructions
else :
  bloc n°2 d'instructions

Son fonctionnement est le suivant :

- Python évalue si *Expression booléenne* a pour valeur **True**.
- Si oui, il exécute le bloc n°1 d'instructions.
- Sinon (*Expression booléenne* a donc pour valeur **False**), c'est le bloc n°2 qui sera exécuté.

Remarquez que **if** et **else** ont le même décalage (la même indentation) et que **else** se termine aussi par deux points " : " . Ces deux points marquent le début du bloc n°2 d'instructions. D'autre part, chaque bloc n°1 ou n°2 doit être indenté par rapport au **if** ou au **else**.

Exemple :

```
1 # Structure if - else
2 age = 15
3 if age >= 18 :
4   print("Vous êtes majeur !")
5 else : #dans ce cas, age < 18
6   print("Vous êtes mineur !")
7   reste = 18 - age #nombre d'années avant majorité
8   print("Il vous reste", reste, "ans à attendre")
```

Voici le résultat :

In [1] : (executing lines 1 to 8 of "Programme.py")
Vous êtes mineur !
Il vous reste 3 ans à attendre

2.5 Structure if...elif...else

La structure conditionnelle la plus complète est la structure **if - elif - else** où **elif** est l'abbréviation de "else if = sinon si". La syntaxe générale est :

if *Exp1* **:**
  bloc n°1 d'instructions
elif *Exp2* **:**
  bloc n°2 d'instructions
else :
  bloc n°3 d'instructions

- Si *Exp1* vaut **True**, Python exécute le bloc n°1 d'instructions
- Si *Exp1* vaut **False**, Python passe à l'évaluation de l'expression booléenne *Exp2*.
- Si *Exp2* vaut **True**, il execute le bloc n°2 d'instructions
- Si *Exp2* vaut **False**, Python exécute le bloc n°3 d'instruction qui suivent le **else :**

Notez que **if**, **elif** et **else** sont au même niveau d'indentation et qu'il y a toujours les deux points " : " derrière.

Écrivons par exemple un programme qui demande à l'utilisateur d'entrer un nombre entier ou réel. Ce programme teste le signe de ce nombre et affiche une phrase qui donne ce signe.

```
1  # Structure if - elif - else
2  ch = input("Veuillez entrer un nombre : ")
3  a = float(ch) # Penser à transtyper
4  if a > 0 :
5     print("Le nombre est positif")
6  elif a < 0 :
7    print("Le nombre est négatif")
8  else :
9     print("Le nombre est nul")
```

Résultat à la fin du processus :

In [1] : (executing lines 1 to 9 of "Programme.py")
Veuillez entrer un nombre : -5.45
Le nombre est négatif

3 Les boucles

On utilise des boucles lorsqu'on a besoin de répéter plusieurs fois un bloc d'instructions. En Python, il y a deux types de boucles :

1. La boucle **for**
2. La boucle **while**

3.1 La boucle for

Il existe dans le langage Python des types de données qui sont appelés **itérables** : en pratique, ce sont les chaînes de caractères (type **str**) et les listes (type **list**). Ces données contiennent des éléments qui possèdent un **rang**.

- Dans le cas d'une chaîne de caractères, les éléments sont les différents caractères de la chaîne. Ex : ch = "Confiture", "C" a le rang 0, "o" le rang 1, etc...
- Dans le cas d'une liste, les éléments sont ceux de la liste. Ex : Liste = [1, "banane", 3], 1 a le rang 0, "banane" le rang 1, etc...

On utilise ces types de données pour construire la boucle **for**. Vous aurez alors besoin d'un **itérateur de boucle**, que vous appellerez comme vous voudrez, et qui va parcourir tous les éléments de la donnée, du premier au dernier. La syntaxe est la suivante :

for itérateur **in** valeur itérable **:**
    Bloc d'instructions

ce qui signifie : "*Pour* itérateur *allant du premier élément au dernier élément de* valeur itérable*, exécuter le bloc d'instructions qui suit les deux points " : " et qui est décalé (*indenté*) par rapport à for*".

En voici quelques exemples. Écrivons dans un fichier que nous sauvegarderons sous le nom "Programme.py" les lignes suivantes :

```
1 #Premier exemple de boucle for, avec une chaîne
2  for c in "Robert" :
3     print(c)
```

Éxécution du programme :

In [1] : (executing lines 1 to 3 of "Programme.py")
R
o
b
e
r
t

Dans ce cas, **l'itérateur** est nommé **c** : c'est un caractère qui prend successivement *toutes les valeurs des caractères* de la chaîne "Robert", du premier au dernier.

Vous pouvez aussi placer votre **valeur itérable** dans une variable et ré-écrire la boucle **for** sous la forme :

```
1 #Autre version du premier exemple
2   Nom = "Robert"
3   for c in Nom :
4     print(c)
```

Essayez, cela donne le même résultat !

Une liste est aussi une valeur itérable et qui convient tout à fait pour construire une boucle **for** ! Voyons cela sur un deuxième exemple :

```
1 #Deuxième exemple avec une liste
2   for el in [1, "chou", 2.3e5] :
3     print(el)
```

ou encore

```
1 #Autre version du deuxième exemple
2   Liste = [1, "chou", 2.3e5]
3   for el in Liste :
4     print(el)
```

L'exécution de ce programme conduit à :

In [1] : (executing lines 1 to 4 of "Programme.py")
1
chou
230000.0

D'une façon générale :

- Vous pouvez choisir le *nom* de l'**itérateur** comme vous voulez ! **c**, **el**, **toto**, **Mon_iterateur**, etc...La seule contrainte est que ce nom soit compatible avec les règles qui régissent les noms de variables en Python (voir section **1.2**).
- **L'itérateur** est une véritable variable qui est créée lors du premier passage dans la boucle. On peut donc l'utiliser à l'intérieur du bloc d'instructions. À chaque nouveau passage dans la boucle, sa valeur est modifiée en passant à l'élément suivant de la **valeur itérable**.
- Pour chaque valeur de l'**itérateur**, le bloc d'instructions est exécuté.
- Noter les deux points " **:** " (obligatoires) et le décalage à droite du bloc d'instruction (son indentation) par rapport à **for** : obligatoire aussi.

Écrivez maintenant un programme qui demande à l'utilisateur d'entrer une chaîne de caractères. Le programme doit ensuite compter le nombre de fois où la lettre 'r' est dans la chaîne.

Voici ma solution :

```
1  #Programme de comptage de la lettre 'r'
2  chaine = input("Veuillez entrer une chaîne de caractères : ")
3  compteur = 0    #Initialisation du compteur
4  for c in chaine :
5    if c == 'r' :
6        compteur = compteur + 1
7
8  print("La lettre 'r' est présente", compteur, "fois")
```

Bilan de l'exécution, à la fin de la saisie :

In [1] : (executing lines 1 to 8 of "Programme.py")
Veuillez entrer une chaîne de caractères :Terra incognita
La lettre 'r' est présente 2 fois

Ligne **4**, l'itérateur **c** prend successivement toutes les valeurs des caractères de **chaine**. La structure **if** (lignes **5** et **6**) fait partie de la boucle **for** : on compare chaque valeur de **c** avec 'r' et, si elles sont égales, on incrémente la variable **compteur** d'une unité. Le programme finit par afficher le nombre de fois que 'r' est présente. Noter que l'instruction **print** de la ligne **8** n'est plus dans la boucle **for**, ni dans la structure **if** car elle n'est pas *indentée*. D'autre part, j'ai choisi la forme "....." (2 guillemets) pour la chaine à afficher car je voulais y mettre 'r'. Enfin, j'ai laissé la ligne **7** vide afin d'aérer le programme.

En général, lorsqu'on est un bon matheux, on aime bien construire des boucles **for** où l'**itérateur** sera un entier, appelé **i** par exemple, qui va varier d'un valeur *min* à un valeur *max*, en augmentant d'une unité à chaque fois.

Pas de problème ! Python dispose de la fonction **range**(...) qui renvoie une *séquence d'entiers ordonnés* pouvant servir de **valeur itérable**.

La syntaxe de la fonction **range** est l'une des trois formes suivantes :

range(*fin*) ou **range**(*debut*, *fin*) ou **range**(*debut*, *fin*, *textitpas*)

où *debut*, *fin* et *pas* sont des entiers.

Avec cette fonction **range**(...), la boucle **for** peut s'écrire de plusieurs façons, selon le résultat que l'on souhaite :

- **for i in range**(*fin*) : signifie que **i** va varier de 0 à *fin* - 1 en augmentant de 1 à chaque fois. Dans ce cas, *fin* est forcément un entier > 0. Par exemple avec : **for i in range**(5) , **i** va aller de 0 à 4.

- Avec la deuxième syntaxe de **range**, on peut écrire une boucle **for** sous la forme : **for i in range**(3, 12) ce qui signifie que **i** va varier de 3 (inclu) à 11 (donc attention aux bornes de **i**, la dernière valeur, 12, est exclue).

- Vous pouvez aussi préciser le *pas* de variation de **i** : ce sera la troisième syntaxe de la fonction **range**. Avec cette écriture, la boucle **for** devient : **for i in range**(3, 15, 2) ce qui signifie que **i** va varier de 3 à 14, en augmentant de 2 unités à chaque fois. **i** prendra donc successivement les valeurs 3, 5, 7, 9, 11 et 13.

- Enfin, vous pouvez faire en sorte que la boucle **for** soit décrite par *valeurs décroissantes* de i : dans ce cas, il faut indiquer un *pas* **négatif**. La boucle **for** s'écrit alors, par exemple, **for i in range**(10, 3, -1) et **i** va diminuer de 10 à 4, par *pas* de 1 (et la valeur finale est toujours exclue).

Prenons un exemple en réécrivant notre fichier Programme.py de la façon suivante :

```
1 #Boucle for avec un itérateur entier
2 for i in range(5) :   #Pour i allant de 0 à 4
3   print("La boucle est exécutée", i + 1, "fois)
```

En voici le résultat :

```
In [1] : (executing lines 1 to 3 of "Programme.py")
La boucle est exécutée 1 fois
La boucle est exécutée 2 fois
La boucle est exécutée 3 fois
La boucle est exécutée 4 fois
La boucle est exécutée 5 fois
```

Les entiers *debut* et/ou *fin* de la fonction **range** peuvent être négatifs...cela marche aussi !

```
1 # Testons range avec des entiers négatifs
2 for k in range(-3, 4) :
3   print(k)
```

```
In [1] : (executing lines 1 to 3 of "Programme.py")
-3
-2
-1
0
1
2
```

Prenons un autre exemple. On veut calculer et afficher deux sommes S1 et S2. S1 est la somme des inverses des 30 premiers entiers à partir de 1, c'est à dire :

$$S1 = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{30}$$

S2 est la somme alternée :

$$S2 = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + ... - \frac{1}{30}$$

```
1 # Autre programme avec range(...)
2 S1 = 0
3 S2 = 0
4 for i in range(1, 31) :   #Pour i allant de 1 à 30
5   S1 = S1 + 1 / i
6   S2 = S2 + ( (-1)**(i+1) ) / i
7 print("Fin de la boucle for")
8 print("Somme S1 =", S1)
9 print("Somme S2 = ", S2)
```

Enregistrons le fichier et lançons l'exécution pour obtenir :

```
In [1] : (executing lines 1 to 9 of "Programme.py")
Fin de la boucle for
Somme S1 = 3.9949871309203906
Somme S2 = 0.6767581376913979
```

- Le bloc d'instruction de la boucle **for** peut naturellement contenir plusieurs instructions : lignes **5** et **6** ici. Dans ce cas, toutes ces lignes doivent avoir la même indentation.
- Les lignes **7**, **8** et **9** ne sont plus indentées par rapport à **for** : *elles ne font donc plus partie de la boucle* et ne sont exécutées qu'à la fin de celle-ci.

3.2 La boucle while

Le deuxième type de boucle que vous pouvez construire en Python est la boucle **while**. Sa syntaxe est la suivante :

while *Exp* :

bloc d'instructions

- *Exp* est une expression booléenne. Tant que sa valeur est **True**, Python exécute le bloc d'instructions qui suit les deux points.

- Il faut *placer une condition* dans le bloc d'instructions qui change la valeur de *Exp* à **False**. Sinon la boucle ne s'arrêtera jamais ! De même, si vous voulez que la boucle s'exécute au moins une fois, il faut que *Exp* soit initialisée à **True**.

- Remarquez encore les deux points " **:** " et l'indentation du bloc d'instruction par rapport à **while** : obligatoires en Python.

Finalement, une boucle **while** peut être formulée en pseudo-langage de la manière suivante : "Tant que *Exp* est vraie, exécute le bloc d'instructions qui est indenté".

En voici un exemple :

```
1  # Une boucle while
2  i = 0   #Initialisation
3  while i <= 5 :   #Tant que i plus petit que 5
4    print(i)
5    i = i + 1 #Ne pas oublier de changer la valeur de i
6  print("Nous voici sortis de la boucle !")
```

ce qui donne :

```
In [1] : (executing lines 1 to 6 of "Programme.py")
0
1
2
3
4
5
Nous voici sortis de la boucle !
```

L'instruction i = i + 1 de la **ligne 5** est primordiale car elle assure que l'on va sortir de la boucle. Remarquez que l'instruction de la **ligne 6** *n'est pas indentée* : elle ne fait pas partie du bloc d'instructions while et ne sera exécutée qu'une fois la boucle terminée.

3.3 Le mot-clé break

Il existe en Python un mot-clé **break** qui permet d'interrompre immédiatement une boucle (même si elle n'est pas finie). En voici un exemple :

Le programme suivant demande à l'utilisateur d'entrer une lettre via le clavier. Il y a 15 essais en tout. Si l'utilisateur trouve la lettre gagnante (choisie à l'avance par le programmateur) le programme s'interrompt et affiche "Bravo, vous avez gagné !". Dans cas contraire, il affiche le nombre d'essais restants.

En voici une implémentation

```
1  # Un petit jeu
2  lettre_gagnante = 's'
3  for essai in range(1, 16) :   #essai varie de 1 à 15 = 15 essais
4    c = input("Veuillez taper une lettre : ")
5    if (c == lettre_gagnante) :
6        print("Bravo, vous avez gagné !")
7        break   # Ici la boucle va s'interrompre
8    else :   #La bonne lettre n'a pas été trouvée
9        print("Il vous reste", 15 - essai, "essais")
```

Essayez pour voir ! Remarquez bien les *différentes indentations* des blocs d'instructions qui appartiennent à la boucle **for** et à la structure conditionnelle **if - else**.

Le même mot **break** peut être utilisé dans une boucle **while** : il donnera le même résultat : sortie de boucle immédiatement !

En conclusion, je vous ai cité la possibilité d'utiliser ce mot **break** mais je vous déconseille de l'utiliser. On peut toujours s'en sortir sans.

3.4 Compréhension de liste avec l'instruction for

Lors de l'étude des listes à la section 1.3, nous avions dû créer des listes en tapant chacun des éléments de celles-ci les uns après les autres. Lorsque le nombre d'éléments de la liste devient important, cela devient vite fastidieux !

Heureusement, Python dispose d'une syntaxe qui permet de générer automatiquement une liste en utilisant une boucle **for**. C'est ce qu'on appelle une **compréhension de liste**.

En voici quelques exemples, que l'on écrira dans l'interpréteur pour voir comment çà marche :

In [1] : Liste = [c **for** c **in** "Bonjour"]
In [2] : Liste
Out [2] : ['B', 'o', 'n', 'j', 'o', 'u', 'r']

On génère ainsi automatiquement une liste formée des caractères de la chaine "Bonjour". Continuons avec un deuxième exemple :

In [3] : Liste2 = [(2*i + 1) **for** i **in** **range**(4)]
In [4] : Liste2
Out [4] : [1, 3, 5, 7]

On génère ici la liste des quatre premiers nombres impairs (i étant allé de 0 à 3). Un dernier exemple : une liste des entiers compris entre 1 et 10, en en sautant 1 sur 2 :

In [5] : Liste3 = [i **for** i **in** **range**(1, 11, 2)]
In [6] : Liste3
Out [6] : [1, 3, 5, 7, 9]

La fonction **range** utilisée dans ce dernier exemple étant **range**(1, 11, 2), **i** va varier de 1 à 10 par *pas* de 2.

On peut même introduire des structures conditionnelles dans la compréhension de liste :

In [7] : Liste4 = [x **for** x **in** **range**(10) **if** x%2 == 0]
In [8] : Liste4
Out [8] : [0, 2, 4, 6, 8]

Cela génère une liste formée des entiers compris entre 0 et 9, à condition qu'ils soient pairs (reste de la division entière par 2 égal à 0).

Je vous laisse essayer plusieurs possibilités de créer des listes avec cette méthode très puissante et très utilisée.

Exercices :

1. Créer une liste d'entiers contenant tous les multiples de 7 compris entre 1 et 100.
2. Créer une liste contenant tous les nombres entiers compris entre 1 et 100 et dont le reste de la division par 3 vaut 2.

4 Les fonctions

4.1 Définition d'une fonction

Tous les langages de programmation évolués permettent d'écrire des **fonctions**. Une fonction est un bloc d'instruction que l'on désigne par

un *nom* : c'est le nom de la fonction. À chaque fois que vous écrirez ce nom, le bloc d'instructions sera exécuté.

Voyons un premier exemple. Nous allons écrire une fonction qui affiche tous les nombres pairs compris entre 1 et 20. Écrivons donc dans notre fichier Programme.py les lignes suivantes :

```
1   # Définition d'une fonction
2   def affiche_nombres_pairs() :
3     for n in range(1, 21) :   # Pour n allant de 1 à 20
4         k = n % 2   # k est le reste de la division entière de n par 2
5         if (k == 0) :   # Reste nul ? n est donc pair !
6             print(n)
7
8   # Début du programme
9   print("Le programme commence")
10  affiche_nombres_pairs()
```

Enregistrez le fichier et lancez l'exécution pour obtenir :

```
In [1] : (executing lines 1 to 10 of "Programme.py")
Le programme commence
2
4
6
8
10
12
14
16
18
20
```

La ligne 2 commence par le mot-clé **def** qui permet de définir une **fonction**. La syntaxe générale est la suivante :

def nom_de_la_fonction() **:**
bloc d'instructions

- Vous devez baptiser votre fonction en lui donnant un *nom*. Vous pouvez choisir le nom que vous voulez, pourvu qu'il respecte la syntaxe des noms de variables autorisés par Python (voir page 5).
- Ce nom doit *impérativement* être suivi de deux parenthèses ouvrante et fermante () ainsi que des deux points " : "
- Le bloc d'instructions qui compose la fonction (et qui sera répété à chaque fois qu'on appellera le nom de cette fonction) est écrit en dessous, avec **une indentation**.

Dans notre exemple, le bloc d'instructions associé à la fonction affiche_nombres_pairs() est écrit aux lignes **3**, **4**, **5** et **6**.

Dans la suite de votre programme, à chaque fois que vous écrirez affiche_nombres_pairs(), c'est tout le bloc d'instructions écrit aux lignes **3**, **4**, **5** et **6** qui sera exécuté. C'est ce qui est fait à la ligne **9**. Dans le langage de l'informatique, on dit qu'*on appelle la fonction* affiche_nombres_pairs().

4.2 Passage de paramètres à la fonction

Une fonction qui se contente d'exécuter un bloc d'instructions, c'est sympa, mais c'est encore un peu pauvre. On aimerait pouvoir transmettre des informations à la fonction pour qu'elle puisse faire des calculs avec. Pas de problèmes, il suffit de modifier la façon d'écrire la fonction.

Le programme suivant affiche tous les nombres pairs compris entre 1 et un nombre N qu'on définira au cours du déroulement du programme.

```
1 # Définition d'une fonction avec paramètre
```

```
2 def affiche_nombres_pairs(N) :
3   for n in range(1,N+1) :    # Pour n allant de 1 à N
4     k = n % 2    # k est le reste de la division entière de n par 2
5     il (k == 0) :    # Reste nul ? n est donc pair !
6       print(n)
7
8 # Début du programme
9 print("Le programme commence")
10 affiche_nombres_pairs(11)
11 print("Suite...")
12 affiche_nombres_pairs(5)
```

En voici le résultat :

```
In [1] : (executing lines 1 to 12 of "Programme.py")
Le programme commence
2
4
6
8
10
Suite...
2
4
```

- À la ligne 2, nous avons modifié la définition de la fonction : à la place des parenthèses vides (), nous avons indiqué **affiche_nombres_pairs(N)**. N est ce qu'on appelle un **paramètre** : c'est une *valeur que l'on transmet à la fonction.* Le bloc d'instructions qui compose cette fonction peut alors se servir de N pour effectuer tout un tas de choses.
- Aux lignes **10** et **12**, nous appelons la fonction en lui transmettant diverses valeurs du paramètres : **affiche_nombres_pairs(11)** et **affiche_nombres_pairs(5)**. Dans le premier cas, la fonction s'exécute avec N = 11. Dans le second cas, elle s'exécute avec N = 5.

Lorsqu'une fonction a été définie avec un paramètre, vous pouvez aussi mettre sa valeur dans une variable et appeler la fonction en lui transmettant le nom de cette variable : çà marche aussi !

```
1 # Définition d'une fonction avec paramètre
2 def affiche_nombres_pairs(N) :
3   for n in range(1,N+1) :    # Pour n allant de 1 à N
4     k = n % 2    # k est le reste de la division entière de n par 2
5     il (k == 0) :    # Reste nul ? n est donc pair !
6       print(n)
7 # Suite du programme
8 print("Le programme commence")
9 a = 11
10 affiche_nombres_pairs(a)
11 print("Suite...")
12 b = 5
13 affiche_nombres_pairs(b)
```

Essayez ! Vous verrez que cela donne le même résultat. Remarquez aussi que le nom donné à la variable est complètement libre : vous n'avez pas besoin de l'appeler N (nom du paramètre) !

Sachez enfin que vous pouvez transmettre *plusieurs paramètres* à une fonction. Il faut alors les séparer par une virgule dans la définition. L'écriture générale se fait alors de la manière suivante :

def nom_fonction(*param1*, *param2*, ...) **:**
  bloc d'instructions

Étudions un exemple où on définit une fonction prenant comme paramètres l'age et le nom d'une personne. Cette fonction se charge ensuite d'afficher une phrase du type "Nom" "est agé de " age "ans.

```
1 # Fonction avec deux paramètres
2 # On suppose que age est un entier et nom une chaîne de caractères
3 def affiche(age, nom) :
4     print(nom, "est âgé de ", age, "ans")
5
6 print("Début du programme")
7 affiche(21, "Paul")
8 # Suite du programme
9 mon_age = 7
10 nom = "Jéronimo"
11 affiche(mon_age, nom)
```

Résultat :

In [1] : (executing lines 1 to 11 of "Programme.py")
Début du programme
Paul est âgé de 21 ans
Jéronimo est âgé de 7 ans

Remarquez qu'à la ligne **10**, on définit une variable **nom** qui porte le *même nom* qu'un des paramètres de la fonction affiche : cela est tout à fait permis !

4.3 Valeur retournée par une fonction

Allons plus loin avec les fonctions ! Il serait dommage qu'une fonction fasse des calculs sans renvoyer un résultat. Cela est possible à l'aide du mot-clé **return**.

Écrivons une fonction qui prend un nombre réel en paramètre et qui renvoie son carré. C'est très simple car il suffit d'écrire :

```
1 #Fonction carrée
2 def carre(x) :
3     y = x**2
4     return y
5
6 #Début du programme
7 print("Début du programme")
8 a = carre(4)
9 print(a)
10 b = 3
11 a = carre(b)
12 print(a)
```

In [1] : (executing lines 1 to 12 of "Programme.py")
Début du programme
16
9

- La fonction **carre** est définie aux lignes **2**, **3** et **4**. Elle renvoie le carré du nombre réel qu'on lui passe en paramètre.

- Au cours du programme, on peut alors utiliser la valeur renvoyée : par exemple aux lignes **8** et **11**. Ligne **8**, la fonction **carre** est appelée avec le paramètre 4 et le résultat (la valeur retournée) est affecté à une variable nommée **a**. C'est le même processus ligne **11**.

Fonction retournant plusieurs valeurs

Ce qui est bien avec Python, c'est que, contrairement à beaucoup d'autres langages de programmation, les fonctions peuvent renvoyer **plusieurs valeurs**. Voyons un exemple

```
1 #Fonction renvoyant 2 valeurs
```

```
2 def carre_cube(x) :
3   y = x**2
4   z = x**3
5   return y, z
6
7 #Début du programme
8 a, b = carre_cube(2)
9 print("première valeur : ", a, "seconde valeur : ", b)
```

ce qui donne :

In [1] : (executing lines 1 to 9 of "Programme.py")
première valeur : 4 seconde valeur : 8

Ligne **5**, la fonction retourne 2 valeurs grâce à l'instruction **return** y, z. Ces deux valeurs peuvent être affectées à deux variables **a** et **b**, selon la syntaxe de la ligne **8**. Les deux variables doivent être séparées par une virgule :

a, b = **carre_cube**(2)

La première valeur (y) est renvoyée dans **a** et la seconde (z) dans **b**.

En fait, le fonctionnement de Python dans ce cas consiste à créer un "tuple" contenant les différentes valeurs retournées. La syntaxe rigoureuse devrait dont être : (a, b) = **carre_cube**(2) qui marche de la même façon que la syntaxe (**Mot**, **b**) = ('arbre', 23) que nous avions vue à la section 1.3, lors de l'étude du type **tuple**.
Comme Python est souple sur ce point, on peut se permettre de ne pas écrire les parenthèses.

Cependant, comme la valeur retournée par la fonction est de type **tuple**, on peut aussi écrire le programme précédent de la façon suivante. Je n'ai pas réécrit les lignes **2** à **6** du programme et elles sont sous-entendues :

```
1   #Fonction renvoyant 2 valeurs. Autre façon d'écrire
...
7   #Début du programme
8   t = carre_cube(2)    # t variable de type tuple
9   print("première valeur : ", t[0], "seconde valeur : ", t[1])
```

Essayez pour voir, cela marche aussi ! A la ligne **9**, **t**[0] est le premier élément du "tuple" **t**, c'est à dire 4 (x**2) et **t**[1] est le second élément de **t**, ie 8 (x**3)

Une dernière chose avant de passer à la suite. On aurait pu écrire les fonctions **carre** et **carre_cube** de façon beaucoup plus compacte, sans définir les variables y et z :

```
def carre(x) :
  return x**2
```

et

```
def carre_cube(x) :
  return x**2, x**3
```

Cela marche très bien aussi ! Si j'ai introduit les variables **y** et **z**, c'est pour aborder la notion de **variable locale**, étudiée dans la section suivante.

4.4 La portée des variables

Toute variable est définie soit dans un programme que vous rédigez dans un fichier, soit à l'intérieur d'une fonction. Autrement dit, toute variable est définie à l'intérieur d'un *espace* (programme, fonction ...) et on dit que c'est une **variable locale** à cet espace.

Dans le cas d'une **fonction**, les **variables locales** à celle-ci sont les *paramètres* de la fonction et *toutes les variables définies à l'intérieur*

de cette fonction. Par exemple, la variable **y** qui est définie dans le bloc d'instructions de la **fonction carre** est une **variable locale** à cette fonction.

Python suit les règles suivantes en matière d'utilisation des variables :

1. Une variable locale à une **fonction** ne peut être utilisée que dans celle ci. Elle est totalement inaccessible à une instruction située en dehors de la **fonction**.
2. Une variable définie dans un programme, mais en dehors d'une **fonction** est accessible et manipulable dans tout le reste du programme, c'est à dire dans les lignes qui suivent la définition de cette variable...tant que vous restez en dehors des fonctions.
3. Si une variable, appelons-la **a**, est définie dans un programme (à la ligne **5** par exemple), et que vous définissez par la suite une **fonction** (à partir de la ligne **27** par exemple), **a** sera accessible en lecture dans la fonction mais jamais en écriture ! Cela signifie qu'une instruction située dans la **fonction** ne peut pas modifier **a** !

Voyons cela sur quelques exemples :

```
1 #Programme illustrant la portée des variables
2 def carre(x) :
3   y = x**2   #y variable locale à carre tout comme x
4   return y
5
6 a = 4 #a variable locale au programme, utilisable à partir
7 #de la ligne 6
8
9 def affiche() : #définition d'une autre fonction utilisant a
10   print("J'affiche une variable du programme : ", a)
11 #Ligne 10, a est accessible en lecture seulement
12
13 b = carre(3) # b est aussi une variable locale au programme
14 #À partir de la ligne 11, on peut utiliser b
15 print(b)
16 affiche()
17 a = "Julie"
18 affiche()
```

En voici le résultat :

```
In [1] : (executing lines 1 to 18 of "Programme.py")
9
J'affiche une variable du programme : 4
J'affiche une variable du programme : Julie
```

La fonction **affiche**() utilise la variable **a**, locale au programme, mais uniquement en lecture : elle ne la modifie pas.

Un deuxième point important qu'il faut bien comprendre est qu'une variable locale à une **fonction** *n'est pas accessible* en *dehors de la fonction où elle est définie*. On peut le voir à partir du programme suivant :

```
1 #Fonction carre modifiée pour illustrer portée variable locale
2 def carre(x) :
3   y = x**2
4   print("On est dans la fonction et y = ", y)
5
6 #Début du programme
7 carre(4)
8 print("On est en dehors de la fonction et y = ", y)
```

Voici ce que cela donne :

In [1] : (executing lines 1 to 8 of "Programme.py")
On est dans la fonction et y = 16

NameError File "Programme.py", line 8, in <module>
print("On est en dehors de la fonction et y = ", y)
NameError : name 'y' is not defined

Python ne reconnait plus la variable **y** en dehors de la fonction ! Tentez la même chose avec **x**, en écrivant par exemple une instruction du genre **print**(x) en dehors de la fonction carre, cela va provoquer la même erreur : **NameError** : name 'x' is not defined.

Retenez qu'*une variable locale à une fonction, c'est à dire qui a été définie dans le corps d'une fonction, n'est accessible qu'à l'intérieur de celle-ci.* Cette variable est **détruite** lorsque la fonction se termine.

Une troisième chose importante est qu'une variable définie dans un programme en dehors d'une fonction et avant celle-ci est uniquement accessible en lecture dans la fonction mais n'est pas modifiable par une instruction située dans la fonction.

Exemple :

```
1 #Modification d'une variable
2 a = 4   # a locale au programme
3 def changer() :
4   a = 6 #Essayons de modifier a
5
6 #Début du programme
7 changer()
8 print(a)
```

En voici le résultat :

In [1] : (executing lines 1 to 8 of "Prog.py")
4

Comme vous le voyez, la variable **a** n'a pas été modifiée ! La fonction **changer**() est totalement impuissante sur ce point.

Bon, alors comment faire pour changer le contenu de **a** depuis une fonction ? C'est simple, il y a deux méthodes :

1. Vous passez **a** en paramètre à la fonction : méthode fortement conseillée !
2. Vous déclarez **a** comme **variable globale** à l'aide du mot-clé **global**. Cà, c'est vraiment très déconseillé !

Première méthode :

```
1 #On passe a en paramètre
2 a = 4   # a locale au programme
3 def changer(x) :
4   x = 6
5
6 #Début du programme
7 changer(a)
8 print(a)
```

qui donne :

In [2] : (executing lines 1 to 8 of "<Prog.py>")
6

Cà a marché !

Deuxième méthode (Bôf) :

```
1 #On déclare a comme variable globale
2 a = 4    # a locale au programme
3 def changer() :
4   global a
5   a = 6 #On peut maintenant modifier a
6
7 #Début du programme
8 changer()
9 print(a)
```

et voici le résultat :

In [3] : (executing lines 1 to 9 of "Prog.py")
6

Et voilà ! Vous avez pu modifier **a** depuis l'intérieur de la fonction grâce à la ligne **4** où vous l'avez déclaré comme *variable globale*.

De façon générale, il est ***fortement déconseillé*** d'utiliser des variables globales à l'intérieur des fonctions. En effet, si ces variables sont modifiées par une fonction et qu'une autre ligne d'instruction dans le programme s'attend à ce que la variable ait gardé sa valeur initiale, cela risque d'altérer fortement le comportement du programme.

Supposez par exemple que votre programme définisse **a** = 3. Vous avez ensuite besoin d'utiliser une fonction importée d'une bibliothèque écrite par quelqu'un d'autre. Cette fonction utilise une instruction du genre **global a** pour écrire **a** = "Julie". Vous ne le savez pas et vous vous attendez toujours à ce que **a** soit égale à 3. La prochaine fois que vous écrirez **a** = **a** + 1 par exemple, ce sera le plantage du programme garanti !

4.5 Utilisation de bibliothèques

Dans sa configuration standard, Python sait faire beaucoup de calculs, mais il y a des limites...Essayez pour voir de calculer cos(30). Vous pouvez taper directement dans l'interpréteur :

In [1] : cos(30)

```
NameError
<ipython-input-31-0e53b60bf11e> in <module>()
----> 1 cos(30)
NameError : name 'cos' is not defined
```

Manifestement, Python ne connaît pas bien sa trigonométrie !

En fait, cela est normal : Python a été conçu de *façon modulaire*. Lorsque vous lancez Python, seul le noyau de l'application est chargé dans la mémoire de l'ordinateur : cela permet déjà de faire pas mal de calculs, mais ce n'est pas suffisant.

Python dispose aussi de **modules** (on peut aussi parler de **bibliothèques**) : ce sont des fichiers stockés sur le disque dur qui contiennent des définitions de **fonctions** et de **constantes**. Pour pouvoir accéder à ces fonctions et constantes, il faut d'abord charger le **module** dans la mémoire de l'ordinateur. Cela se fait à l'aide de la méthode **import**.

Prenons un exemple avec le **module** math : c'est un fichier qui contient les définitions des fonctions **cos**, **sin**, **tan**, **atan** (arc tangente), **exp**, ...et bien d'autres fonctions encore, ainsi que des constantes comme **pi** par exemple.

Pour voir toutes les fonctions et constantes d'un module, le plus efficace est d'aller dans l'interpréteur et de taper :

In [1] : import math

Vous appuyez ensuite sur la touche "Entrée" de votre clavier. Apparemment, il ne se passe rien...mais en fait, Python vient de charger le **module** math dans la mémoire de l'ordinateur. Vous pouvez alors voir ce qu'il contient en tapant :

In [2] : help(math)

Vous avez alors intérêt à agrandir la fenêtre de l'interpréteur car Python va vous afficher toutes les fonctions et constantes que contient le **module** math, avec leurs syntaxes, et cela peut fait pas mal de choses !

Une fois le **module** chargé en mémoire, vous pouvez l'utiliser. Voyons cela sur un exemple en écrivant le programme dans un fichier, que vous pouvez sauvegarder sous le nom que vous voulez, programme.py par exemple.

Une bonne habitude à prendre consiste à importer le **module** au début du fichier. Comme cela, s'il n'est pas déjà chargé, Python va le faire dès le début de sa lecture du fichier. Voici ce que cela donne :

```
1  #Programme avec importation de module
2  import math
3  a = math.sqrt(3) #fonction racine carrée
4  b = math.cos(math.pi/3) #fonction cos et utilisation de pi
5  print("Racine : ", a, "Cosinus : ",b)
```

En voici le résultat après exécution :

In [1] : (executing lines 1 to 5 of "programme.py")
Racine : 1.7320508075688772 Cosinus : 0.5000000000000001

- Les fonctions et constantes du module math doivent impérativement s'écrire **math.nom_fonction** ou **math.nom_constante**. Si vous oubliez, Python ne les reconnaîtra pas ! Vous serez donc obligé d'écrire : **math.sin**(3) ou **math.pi** par exemple.

- Remarquez l'erreur d'arrondi dans le calcul du cosinus. La valeur exacte est $\cos(\pi/3) = 1/2$ mais Python laisse traîner un petit 1 à la fin du résultat...eh oui, personne n'est parfait (cela est dû à la façon d'implémenter les nombres réels dans la machine).

Bon, écrire à chaque fois **math.cos** ou **math.sqrt**, cela devient rapidement lourd...

Une façon de résoudre ce problème est de changer la syntaxe de l'importation. Il faut écrire dans le fichier :

from math **import** *

Cela aura le même résultat que **import** math : Python va charger toutes les fonctions et constantes de ce **module**, mais vous n'aurez plus besoin de mettre des **math.** partout pour les utiliser. Essayez pour voir :

```
1  #Programme avec une autre façon d'importer
2  from math import *
3  a = sqrt(3) #fonction racine carrée
4  b = cos(pi/3) #fonction cos et utilisation de pi
5  print("Racine : ", a, "Cosinus : ",b)
```

Cela marche très bien. On obtient le même résultat :

In [2] : (executing lines 1 to 5 of "programme.py")
Racine : 1.7320508075688772 Cosinus : 0.5000000000000001

Pour être précis, la syntaxe **from** math **import** * est un cas particulier d'une syntaxe où vous donnez la liste des fonctions et constantes à importer dans le module math. Vous pourriez écrire aussi :

from math **import** cos, sqrt, pi

Dans ce cas, Python ne charge en mémoire que ce que vous lui indiquez, à savoir : cos, sqrt et pi. Vous pouvez ensuite les utiliser sans mettre **math.** devant

```
1 #Encore une autre façon d'importer
2 from math import cos, sqrt, pi
3 a = sqrt(3) #fonction racine carrée
4 b = cos(pi/3) #fonction cos et utilisation de pi
5 print("Racine : ", a, "Cosinus : ",b)
```

Ce programme fonctionne aussi ! Simplement, lorsque vous placez une étoile " * " à la place de la liste, vous ordonnez à Python d'importer *toutes les fonctions et constantes du module math.*

Alors, la vie est belle, non ? Eh bien pas tout à fait ! En fait, les développeurs de Python et moi aussi, nous vous déconseillons fortement d'utiliser cette syntaxe. Pourquoi ? Tout simplement parce que Python dispose de centaines de modules dans lesquels il y a des fonctions qui portent le même nom, mais qui ne font pas la même chose.

Il y a par exemple les **modules** *math* et *cmath.* Ce dernier est une bibliothèque pour les calculs avec les nombres complexes. Dans ces deux modules, il y a une fonction nommée *exp*, mais qui ne fait pas la même chose : celle du module *math* renvoie une valeur réelle et celle du module *cmath* une valeur complexe. Donc si vous écrivez :

```
1 #Attention : risque de conflit
2 from math import *
3 from cmath import *
4 a = exp(3)
5 ...
```

Ligne **4**, Python va choisir à votre place quelle fonction *exp* il va utiliser. Vous ne contrôlez plus rien et il se peut même que vous plantiez le système si vous supposez que **a** est réel alors que Python a renvoyé une valeur complexe...

Bilan : pour éviter les *conflits de noms*, évitez d'utiliser la syntaxe **from** *nom_module* **import** *, surtout si vous importez plusieurs modules.

Rassurez-vous cependant, tout n'est pas perdu : il y a une troisième façon d'importer un module qui consiste à le **rebaptiser**. La syntaxe est par exemple :

import math **as** m

Dans ce cas, vous importez toutes les fonctions et constantes du **module** *math* et, en même temps, vous le rebaptisez *m.* En langage informatique, on dit que vous créez un *alias* du nom de ce module. Par la suite, les fonctions seront accessibles sous la forme : **m.cos**, **m.sqrt** ou encore **m.pi**.

Bien sûr, vous pouvez renommer votre module comme vous voulez (sous réserve que la syntaxe du nom convienne à Python : voir noms des variables section 1.2), par exemple : ma, toto, Nabuchodonosor, etc... mais l'idée est quand même de faire court.

C'est cette dernière syntaxe qui est conseillée par tout le monde.

Écrivons alors une dernière fois notre fichier programme.py

```
1 #Un programme bien écrit
2 import math as m
```

```
3 a = m.sqrt(3)
4 b = m.cos(m.pi/3)
5 print("Racine : ", a, "Cosinus : ",b)
```

Pour finir cette section, sachez qu'il y a quelques modules intéressants à utiliser dans les calculs scientifiques. Il s'agit des :

- **module** *math* : la plupart des fonctions réelles usuelles des maths et les constantes *pi* et *e*.

- **module** *cmath* : les fonctions dédiées aux calculs avec les nombres complexes.

- **module** *numpy* : c'est un module qui permet faire des calculs sur des matrices de réels ou de complexes. Ces calculs sont très rapides et performants en temps et en espace mémoire car le code a été complètement optimisé.

- **module** *matplotlib.pylab* : il permet de tracer des courbes, des surfaces, des graphes de toute sorte...bref, de faire de beaux dessins.

Nous reparlerons des deux derniers plus précisément au cours de l'année. Si vous êtes curieux, vous pouvez aller chercher de la documentation sur internet à leur sujet. N'oubliez pas non plus la fonction **help** dans l'interpréteur pour avoir tous les renseignements sur ce que contient un module.

5 Python est un langage orienté objet

5.1 Qu'est-ce qu'un objet ?

Sans que vous vous en rendiez compte, Python est une langage *orienté objet*, tout comme les langages C++ ou Java. D'autres langages comme le C ne sont pas orientés objet.

En Python, **tout est objet**...

Que signifie ce terme ? Je ne vais pas vous faire un cours de langage orienté objet car il y en aurait pour des dizaines de pages et des heures, mais je vais vous en indiquer l'essentiel pour bien comprendre et appliquer toutes les possibilités de Python.

En informatique, un **objet** est un espace de la mémoire de l'ordinateur qui peut contenir à la fois des *données* et du *code*, c'est à dire des **valeurs** et des **fonctions** (instructions exécutables par la machine). Il faut vous représenter un objet comme une commode avec des tiroirs, qui est placée quelque part dans la mémoire de la machine.

- Certains de ces tiroirs sont spécialisés et contiennent uniquement des *données* : nombres entiers, nombres réels, chaînes de caractères, etc... Ces tiroirs spéciaux réservés aux données sont appelés les **attributs** de l'objet.

- D'autres tiroirs sont réservés à des tâches d'exécution et ne contiennent que des **fonctions**, c'est à dire des instructions exécutables par le processeur de la machine. Ces **fonctions** sont appelées les **méthodes** de l'objet.

Un objet est identifié par son **nom** et par son **type** mais dans le cas d'un langage orienté objet, on parle plutôt de **classe** à la place du **type** : un objet sera donc identifié par son **nom** et par sa **classe**.

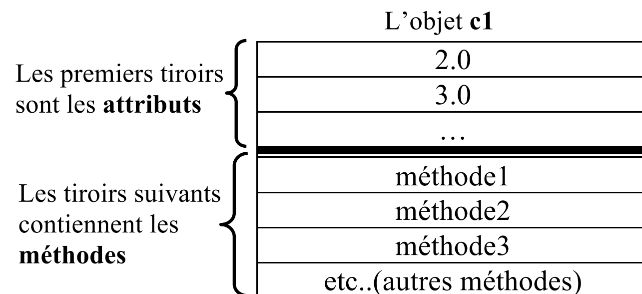
- Le **nom** d'un objet suit les règles énoncées à la section 1.2 , page 5. Il ne peut être composé que des caractères de l'ensemble {a, b, c, ..., y, z, A, B, C, ..., Y, Z, 1, 2, 3, ..., 9, _ } et ne peut pas commencer par un chiffre. Il est sensible à la casse : ABC et AbC ne forment donc pas le même nom.

- Quant à la **classe** de l'objet, vous en avez déjà rencontré plein d'exemples. On peut rencontrer des objets de classe **int**, **float**, **complex**, **str**, **tuple**, **list**. Python vous permet aussi de définir vous-même d'autres classes...mais de cela, nous ne parlerons pas ici.

Soyons concrets ! Vous avez déjà utilisé l'instruction **c1** = 2+3j et vous aviez l'impression de créer une simple variable de type **complex** (c'est du moins ce que je vous avait dit)...Eh bien cela n'est pas tout à fait vrai !

En tapant cette instruction, vous avez créé un **objet** dont le *nom* est **c1** et dont la *classe* est **complex**. Cet objet contient beaucoup plus que la donnée 2+3j : c'est quelque chose de beaucoup plus riche !

En fait, il faut vous représenter l'objet **c1** de la façon suivante : une commode avec plusieurs tiroirs comme cela est dessiné ci-dessous :



En particulier, on voit que l'objet **c1** possède, entre autres, deux **attributs** (les deux premiers tiroirs sur la figure) qui contiennent respectivement la partie réelle 2.0 et la partie imaginaire 3.0 de **c1**, stockées sous la forme de deux nombres réels. En vérité, ce ne sont pas les seuls attributs de **c1** mais on ne va s'intéresser qu'à ces deux là.

Les tiroirs suivants contiennent des **méthodes** dont nous parlerons par la suite.

Il faut vous représenter la **classe** d'un objet comme le plan de fabrication de cet objet. Ce plan indique le nombre de tiroirs **attributs** et la nature des données que contiennent ces tiroirs, puis le nombre de tiroirs réservés aux méthodes, ainsi que les instructions (code) de ces méthodes.

5.2 Comment connaître les attributs et les méthodes d'un objet ?

Tous les objets d'une classe donnée sont fabriqués sur le même modèle. Leurs **attributs** (je vous rappelle que ce sont des tiroirs qui contiennent des données) et leurs **méthodes** (là, ce sont des fonctions) sont *repérés par leurs noms*. Comment connaître ces noms ?

C'est très simple, vous allez dans l'interpréteur et vous créez un objet. On va prendre l'exemple d'un objet de classe **complex** :

In [1] : c1 = 2 + 3j

L'objet **c1** vient donc d'être créé. Pour connaître les noms de tous ses attributs et méthodes, il faut ensuite utiliser la fonction **dir**, en tapant :

In [2] : dir(c1)
Out [2] :
['__abs__' ,
'__add__' ,
'__bool__'
...
'__truediv__'
'conjugate' ,

'imag' ,

'real']

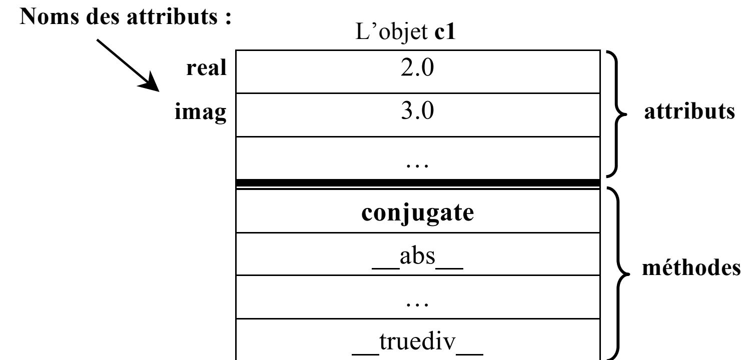
Python vous renvoie sous la forme d'une liste de chaînes de caractères, de la forme ['nom1', 'nom2', ...] , les noms de tous les attributs et de toutes les méthodes de l'objet **c1**. Il y en a beaucoup, une quarantaine environ ! Malheureusement, Python ne vous indique pas lesquels sont des noms de méthodes et lesquels sont des noms d'attributs. Pas grave pour le moment, je vais vous l'indiquer

- Certains de ces noms (la plupart) sont de la forme ___nom___ : ils commencent et finissent par 2 tirets de soulignement "___". Ce sont des **attributs spéciaux** ou des **méthodes spéciales** dont nous ne parlerons pas et que nous n'utiliseront pas pour le moment. Vous pouvez les oublier.

- À la fin de la liste vous avez les noms **real** et **imag**, sans tirets de soulignement : ce sont deux attributs "ordinaires" de **c1** qui contiennent respectivement la partie réelle et la partie imaginaire de **c1**.

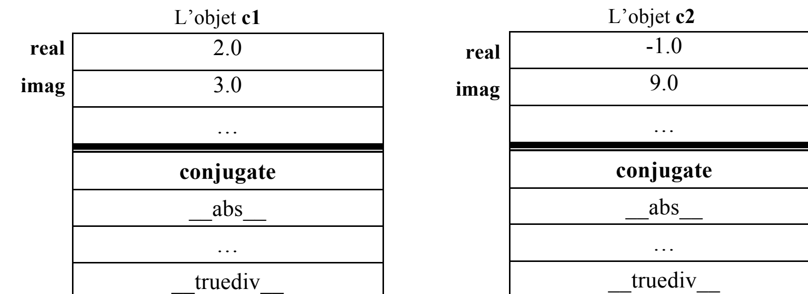
- Enfin, vous avez le nom **conjugate**, sans tirets de soulignement non plus, qui désigne une méthode "ordinaire" de **c1** (ordinaire au sens de : pas spéciale).

À ce niveau, vous pouvez vous représenter l'objet **c1** comme sur la figure ci-dessous :



Il faut bien comprendre que les noms des attributs et des méthodes *sont fixés par la classe de l'objet*, c'est à dire son modèle de fabrication. Deux objets d'une même classe, par exemple **c1** = 2 + 3j et **c2** = -1 + 9j auront les mêmes noms d'attributs et de méthodes.

Ce qui va faire la différence entre **c1** et **c2**, c'est que vous aurez deux commodes fabriquées sur le même modèle mais situées dans deux endroits différents de la mémoire de l'ordinateur, et *dont les contenus des attributs seront différents.*



En conclusion de cette section : ne vous inquiétez pas et ne pensez pas que tout ce qu'on a dit avant sur les variables, les types, les boucles, les structures conditionnelles, etc... est faux. Python sait bien

se dissimuler et faire comme si **c** = 2 + 3j ou **ch** = "Bonjour" étaient de simples variables ne contenant que 2 + 3j ou une simple chaîne "Bonjour". On peut d'ailleurs complètement s'illusionner et taper du code Python pendant des mois ou des années en pensant qu'on a affaire à des variables simples.

Le fait que Python soit un langage orienté objet ne lui retire rien par rapport à çà. Cela ne fait que le rendre plus riche en possibilités. C'est ce que vous allez voir dans les sections suivantes.

5.3 Comment accéder à un attribut ?

Vous pouvez accéder l'attribut d'un objet grâce à la syntaxe :

nom_objet**.**nom_attribut

Prenons un exemple avec l'interpréteur :

In [1] : c1 = 4 + 7j
In [2] : c1**.**real
Out [2] : 4.0
In [3] : c1**.**imag
Out [3] : 7.0

Cette notation avec un point " **.** " séparant le nom de l'objet du nom de son attribut est typique des langages orientés objets. Continuons avec des exemples :

In [4] : AutreC = -2 + 3j
In [5] : AutreC**.**real
Out [5] : -2.0

Vous pouvez alors utiliser ces attributs pour faire des calculs de façon usuelle, comme avec les variables classiques :

In [6] : a = c1**.**real + 5
In [7] : a
Out [7] : 9.0

En revanche vous ne pouvez pas affecter une valeur à un attribut comme avec une variable classique. Si vous tapez :

In [7] : c1**.**imag = 15

AttributeError
<ipython-input-50-413e956a26ee> in <module>()
—-> 1 c1.imag = 15

AttributeError : readonly attribute

Python n'accepte pas et il vous dit pourquoi : readonly attribute. L'attribut n'est accessible qu'en lecture !

> Un point important à retenir, qui concerne la philosophie des langages orientés objet : il faut éviter au maximum de manipuler directement les attributs d'un objet !

Je vous ai cité les attributs **real** et **imag** d'un objet de classe **complex** parce qu'ils sont intéressants et qu'on peut les utiliser en lecture...Si maintenant vous voulez les modifier, il vaut mieux écrire :

In [8] : c1 = 4 + 15j
In [9] : c1**.**imag
Out [9] : 15.0

5.4 À quoi sert une méthode ?

Une **méthode** d'un objet sert à *réaliser une action* sur les attributs de cet objet. Il s'agit d'une véritable **fonction** informatique comme vous en avez étudié dans la section 4.

Comment utilise-t-on une méthode dans un programme ? C'est simple, si un objet nommé **ch** possède une méthode nommée **replace**, vous appelez cette fonction en écrivant :

ch.replace() ou **ch.replace**(*param1*, *param2*, ...)

- La notation est toujours : nom_objet**.**nom_methode, avec le point séparant le nom de l'objet de celui de la méthode. Rappelez-vous que c'est une notation classique dans un langage objet.

- Une méthode étant une **fonction**, il faut à minima l'écrire avec des parenthèses ouvrante et fermante. Si elle nécessite des paramètres, il faut les mettre entre les parenthèses : **ch.replace**() méthode sans paramètre ou **ch.replace**(*param1*, *param2*) méthode avec deux paramètres.

- Comment savoir quels sont les paramètres de la méthode et ce qu'elle fait exactement ? Il faut aller voir dans l'aide de Python ou sur internet. Il y a plein de sites qui sont très documentés sur toutes les méthodes de tous les objets de Python.

Encore une chose avant de revenir à du concret. Sachez qu'il y a deux types de méthodes :

1. Celles qui agissent uniquement sur les attributs de l'objet et **qui ne retournent rien**. Dans ce cas, il n'y a pas d'instruction **return** à la fin de la méthode.
2. Celles **qui retournent un autre objet** avec des attributs dont le contenu est rempli grâce aux instructions de la méthode.

Regardons cela sur un exemple, toujours avec un objet de classe **complex**. On commence par créer l'objet nommé **c** :

In [1] : c = 1 + 2j

Ensuite, nous savons que cet objet possède la méthode **conjugate**. Pour avoir plus de précisions sur cette méthode, on peut toujour taper :

In [2] : **help**(c**.**conjugate)
Help on built-in function conjugate :
conjugate(...) method of builtins.complex instance
  complex.conjugate() -> complex
    Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j.

Les points intéressants ici sont les deux dernières lignes. Vous trouvez : complex.conjugate() -> complex ce qui vous donne l'information que cette méthode ne prend pas de paramètres et qu'elle retourne (flèche ->) un nouvel objet de classe **complex**. Cette méthode est donc de type 2.

Cela est confirmé par la dernière ligne qui vous dit en plus que le complexe retourné est le nombre complexe conjugué. Bon, çà on s'en doutait...

Vous pouvez donc récupérer ce nouvel objet, en le nommant **c1** par exemple :

In [3] : c1 = c**.**conjugate()
In [4] : c1
Out [4] : (1 - 2j)
In [5] : c1**.**imag
Out [5] : -2.0
In [6] : c
Out [6] : (1 + 2j)
In [7] : c**.**imag
Out [7] : 2.0

Vous voyez que l'objet **c** *n'a pas été modifié*. Il y a eu création d'un second objet qu'on a nommé **c1** et qui est le complexe conjugué de **c**.

Nous allons voir dans la section suivante des méthodes de type 1 qui modifient les attributs de l'objet et ne retournent rien

5.5 Les objets de la classe str

5.5.1 Introduction aux objets str

Souvenez-vous qu'en Python, tout est objet.

Lorsque dans un programme vous écrivez : **ch** = "Bonjour", vous créez un **objet** nommé **ch** et qui, en l'occurence, est de la classe **str**, c'est à dire la classe des objets "chaîne de caractères". Comme tout objet, **ch** possède :

- des **attributs**. Un de ces attributs contient la chaîne de caractères "Bonjour" qui nous a servi à créer l'objet.
- Des **méthodes**.

Si vous voulez connaître les noms de tous les attributs et méthodes de **ch**, vous tapez **dir**(ch) dans l'interpréteur. Vous verrez apparaître toute une liste, avec des **attributs et méthodes spéciaux**, du genre '___nom___', qu'on laisse donc de côté, mais aussi, en fin de liste des attributs et méthodes "ordinaires", sans tirets de soulignement.

Prenons une de ces méthodes : **upper** par exemple. Pour savoir ce qu'elle fait, tapons **help**(ch.upper) dans l'interpréteur :

In [1] : ch = "Bonjour"
In [2] : **help**(ch.upper)
Help on built-in function upper :

upper(...) method of builtins.str instance

S.upper() -> str

Return a copy of S converted to uppercase.

Vous voyez tout de suite que la méthode ne prend aucun paramètre et qu'elle renvoie un nouvel objet de la classe **str** qui est formé des caractères de **ch** convertis en majuscules.

Essayez ! (Attention il faut récupérer le nouvel objet) :

In [3] : ch1 = ch**.**upper()
In [4] : ch1
Out [4] : 'BONJOUR'

Cà marche !

Il y a beaucoup de méthodes avec les objets "chaînes de caractères" et le mieux est que vous les exploriez par vous même en vous aidant de l'aide dans l'interpréteur ou sur internet. Je vais simplement en décrire 4 assez utilisées :

5.5.2 La méthode count

L'aide de Python vous indique que la syntaxe est S.count(sub [, start [, end]]) et qu'elle retourne un objet de classe **int** (Eh oui, même les entiers sont des objets en Python !). Pour la notation, il faut savoir que tous les paramètres entre crochets sont facultatifs, donc, ici *start* et *end* peuvent être omis.

Cette méthode renvoie le nombre de fois qu'une sous-chaîne **sub** apparaît dans la chaîne **S**[**start** : **end**]

Par exemple, si ch = "Près de cette ville passait un grand fleuve qui coulait d'ouest en est", alors :

- i = **ch.count**('s') : **i** contient le nombre de fois qu'apparaît le caractère 's' dans **ch**.
- i = **ch.count**('ou', 8) : **i** contient le nombre de fois qu'apparaît 'ou' dans la sous-chaîne de **ch** qui commence au caractère de **rang** 8 (c'est à dire le 'c' de cette..) et va jusqu'à la fin.

- i = **ch.count**('fleuve', 8, 51) : i contient le nombre de fois qu'apparaît 'fleuve' dans la sous-chaîne de **ch** qui commence au caractère de **rang** 8 et qui s'achève au caractère de **rang** 50 (c'est à dire le 'u' de coulait) (souvenez-vous du parcours de chaîne de la section 1.3).

In 5] : ch = "Près de cette ville passait un grand fleuve qui coulait d'ouest en est"
In [6] : i = ch.count('s')
In [7] : i
Out [7] : 5
In [8] : i = ch.count('ou', 8)
In [9] : i
Out [9] : 2
In [10] : i = ch.count('fleuve', 8, 51)
In [11] : i
Out [11] : 1
In [12] : i = ch.count('fleuve', 8, 15)
In [13] : i
Out [13] : 0

5.5.3 La méthode lower

Si **S** est un objet de classe **str**, **S.lower**() renvoie un objet de type **str** dont les caractères sont ceux de **S** mais tous en minuscules.

In 14] : ch1 = "HeLlO WoRlD"
In [15] : petite_ch = ch1.lower()
In [16] : petite_ch
Out [16] : 'hello world'
In [17] : ch1
Out [17] : "HeLlO WoRlD"

Remarquez que **ch1** n'a pas été modifiée.

5.5.4 La méthode replace

Je vous en donne la version simple (voir l'aide pour une version plus sophistiquée avec plus de paramètres). **nom_chaine.replace**(*old*, *new*) renvoie un objet de type **str** où la sous-chaîne *old* est remplacée par la sous-chaîne *new*.

Par exemple, **S** étant une chaîne, **S.replace**('a', 'ou') renverra une chaîne où tous les caractères 'a' de **S** seront remplacés par des 'ou'.

In 18] : ch = "Papa"
In [19] : ch1 = ch.replace('a', 'ou')
In [20] : ch1
Out [20] : 'Poupou'
In [21] : ch
Out [21] : 'Papa'

Ici encore, **ch1** contient la nouvelle chaîne de caractères 'Poupou', mais **ch** n'a pas été modifiée.

5.5.5 La méthode split

Encore une jolie méthode de chaîne de caractères. Lorsque **ch** est une chaîne, **ch.split**(*separateur*) renvoie une liste dont les éléments sont les sous-chaînes coupées dans **ch** selon le *separateur* (qui est un des caractères de **ch**). Un exemple tout de suite pour faire concret :

In 22] : ch = "Papa est parti en vacances"

In [23] : Liste = ch.split(" ")
In [24] : Liste
Out [24] : ['Papa', 'est', 'parti', 'en', 'vacances']

Ici, le séparateur est le caractère "Espace" : " ". La méthode découpe **ch** en sous-chaines qui, dans **ch**, étaient séparées par un " " et elle renvoie toute ces chaînes comme éléments d'une liste. Prenons un autre exemple !

In [25] : ch = "Une jolie maison, spacieuse, ancienne, typique"
In [26] : Liste = ch.split(",")
In [27] : Liste
Out [27] : ['Une jolie maison', ' spacieuse', ' ancienne', ' typique']

Dans ce cas, la chaîne **ch** a été découpée selon le séparateur ",".

5.6 Les objet de la classe list

Eh oui, nos belles listes sont aussi des **objets**, de classe **list**, avec leurs propres méthodes. En voici 4 qui sont intéressantes :

5.6.1 La méthode append

Cette méthode ajoute un élément en fin de liste : on en avait bien besoin car on ne savait pas comment faire jusqu'à présent. C'est une méthode qui ne renvoie rien (donc du type 1) ! Elle modifie la liste elle-même (enfin, disons plutôt ses attributs). Sa syntaxe est :

L.append(*element*)

et cela rajoute *element* à la liste **L**, en fin de liste

In 1] : l1 = [1, "pomme"]
In [2] : l1.append(3.2)
In [3] : l1
Out [3] : [1, 'pomme', 3.2]
In [4] : l1.append("poire")
In [5] : l1
Out [5] : [1, 'pomme', 3.2, 'poire']

5.6.2 La méthode insert

Encore une méthode qui ne renvoie rien ! **L.insert**(n, *element*) insère *element* au **rang** n dans la liste **L**.

In 6] : MaListe = ['c', 2, 3.7]
In [7] : MaListe.insert(1,"poire")
In [8] : MaListe
Out [8] : ['c', 'poire', 2, 3.7]

Cette méthode a donc inséré 'poire' au **rang** 1, c'est à dire après l'élément 'c' qui est de **rang** 0. Les autres éléments, de **rangs** > 1 sont décalés pour faire de la place.

5.6.3 La méthode pop

Là c'est une méthode qui renvoie quelque chose (type 2) : **L.pop**(n) supprime l'élément de **rang** n dans la liste L et renvoie cet élément (ou disons plutôt un objet de la classe de cet élément pour faire précis).

Attention car si la liste est vide où si n est un rang qui n'existe pas dans la liste, Python génère une erreur.

Si le paramètre n est omis : **L.pop**() supprime le *dernier élément* de la liste **L** et renvoie cet élément.

In 9] : MaListe = ['chien', 22, 'cheval']
In [10] : a = MaListe.pop(1)
In [11] : a

Out [11] **:** 22
In [12] **:** MaListe
Out [11] **:** ['chien', 'cheval']
In [12] **:** b = MaListe.pop()
In [13] **:** b
Out [13] **:** 'cheval'
In [14] **:** MaListe
Out [14] **:** ['chien']
In [15] **:** ch = b.upper()
In [16] **:** ch
Out [16] **:** 'CHEVAL'

Ligne **12**, MaListe.pop() supprime le dernier élément de **MaListe**, c'est à dire 'cheval' et le renvoie dans **b**. Cependant comme la valeur renvoyée est un objet (tout est objet en python...), **b** est donc un objet de type **str**. On exploite cela à la ligne **15** en appelant la méthode **upper** de l'objet **b** et l'objet renvoyé est appelé **ch**, de type **str** et contenant 'CHEVAL'.

5.6.4 La méthode remove

La méthode **pop** vue précédemment supprimait un élément d'une liste en y accédant par son **rang**. Vous pouvez aussi avoir besoin de supprimer un élément en y accédant par sa *valeur* : pas de problème, la méthode **remove** est là pour çà !

Si vous avez une liste nommée **L**, alors **L.remove**(*valeur*) supprime l'élément égal à *valeur* de cette liste. Si plusieurs éléments sont égaux à *valeur*, c'est le premier d'entre eux qui est supprimé. Cette méthode ne retourne rien ! Elle supprime simplement un élément dans **L**.

Attention car si **L** ne contient aucune élément égal à *valeur*, le programme s'arrête et une erreur est générée par Python. En voici trois exemples :

In [17] **:** MaListe = ['chien', 22, 'cheval', 43, "chat"]
In [18] **:** MaListe.remove(43)
In [19] **:** MaListe
Out [19] **:** ['chien', 22, 'cheval', 'chat']
In [20] **:** Liste2 = ["crayon", "gomme", "stylo", "gomme", 12]
In [21] **:** Liste2.remove("règle")

ValueError
<ipython-input-8-76a8c537bc11> in <module>()
—-> 1 Liste2.remove("règle")

ValueError : list.remove(x) : x not in list

Python n'aime pas du tout et il arrête tout !

Il existe beaucoup d'autres méthodes de listes. Je vous invite à aller les découvrir avec les fonctions **dir** et **help** ou encore par une recherche sur internet.

6 La bibliothèque numpy

6.1 Qu'est-ce que numpy ?

numpy est un **module** utilisé dans presque tous les projets de calcul numérique sous Python. Il fournit des structures de données performantes pour la manipulation de vecteurs et de matrices. **numpy** est écrit avec du code optimisé d'où ses performances élevées dans les les calculs sur les vecteurs et les matrices.

Pour être utilisé, **numpy** doit être importé, selon une des procédure décrites dans le section 4.5 *Utilisation des bibliothèques*. Je vous conseille d'utiliser la forme "alias", c'est à dire d'écrire par exemple :

import numpy **as** np

Les objets de **numpy** sont de la classe **ndarray**, ce qui signifie *numpy data array*, c'est à dire *tableau* (...de données numpy) si on traduit en français. Par la suite, je désignerai ces objets sous le nom : **tableaux**. Concrètement, vous pouvez regarder un tableau comme une liste mais *dont tous les éléments sont de même type* : que des réels, que des entiers ou encore que des complexes.

Impossible ici d'avoir des tableaux hétérogènes avec certains éléments entiers, d'autres réels et d'autres encore du type "chaîne de caractères" par exemple. Cela est dû au fait que les tableaux sont utilisés pour faire du calcul numérique : un calcul optimisé pour des complexes ne marchera pas forcément pour des entiers. Quant à utiliser des éléments de type non numérique, genre chaînes de caractères : n'en parlons même pas !

Qu'est-ce qui distingue alors un tableau d'une liste numérique, de la forme L = [1, 4, 5, 9, 2] ? Eh bien tout simplement le fait que **numpy** fournit des opérateurs et des méthodes de calculs sur les tableaux qui n'existent pas pour des simples listes numériques.

6.2 Création de tableaux (ndarrays) numpy

Il y a deux façons de créer un tableau numpy :

- À partir d'une liste numérique ;
- En utilisant des fonctions du module numpy telles que **arange**, **linspace**, etc...

6.2.1 À partir d'une liste numérique

Il s'agit d'une véritable **conversion** qui transforme un objet de classe **list** en un objet de classe **ndarray**. Voyons cela en écrivant dans un fichier :

```
1   #Création d'un array à partir d'une liste
2   import numpy as np
3   a = np.array( [ 1, 2, 3, 4 ] )   # Transtypage.
4   print(a)
5   print( type(a) )
```

En voici le résultat :

```
In [1] : (executing lines 1 to 5 of "Prog.py")
[1 2 3 4]
<class 'numpy.ndarray'>
```

Faites attention à l'écriture de la Ligne **3** : **np.array** est une fonction du **module** numpy qu'on a rebaptisé np. Pour l'utiliser, il faut donc écrire **np.array** et pas array tout seul (revoir la section 4.5 sur les bibliothèques si pas clair). Cette fonction convertit une liste en un tableau.

a est désormais un tableau numpy : remarquez que les virgules "," qui séparaient les éléments de la liste ont été remplacées par des espaces " ". L'affichage du type de **a** ne laisse aucun doute : c'est un objet de la classe **ndarray**.

Bien entendu, vous pouvez commencer par mettre votre liste dans un variable nommée **L** par exemple, avant de la convertir :

```
1   #Création d'un tableau à partir d'une liste dans une variable
2   import numpy as np
3   L = [1, 2.34, 1e-5, 6]
4   a = np.array( L )   # Conversion
5   print(a)
```

Résultat :

```
In [1] : (executing lines 1 to 5 of "Prog.py")
```

[1.00000000e+00 2.34000000e+00 1.00000000e-05 6.00000000e+00]

Remarquez qu'ici, Python a automatiquement converti tous les éléments de **L** en réels (type **float**), ce qui est normal : tous les éléments d'un **array** sont du même type : si des conversions doivent être faites, elles le sont !

Vous pouvez demander vous-même le type d'éléments que vous souhaitez dans votre tableau : il suffit d'ajouter le paramètre **dtype** (qui vient de *data type*) dans la fonction **np.array** :

```
1 #Création d'un tableau avec éléments complexes
2 import numpy as np
3 L = [1, 2, 1, 6]
4 a = np.array(L, dtype=complex)
5 print(a)
```

et voilà :

```
In [1] : (executing lines 1 to 5 of "Prog.py")
[ 1.+0.j 2.+0.j 1.+0.j 6.+0.j]
```

Les possibilités sont :

dtype = int ; dtype = float ou dtype = complex

Bon, pour le moment, nous n'avons créé qu'un tableau à une dimension, ce qu'on peut encore interpréter soit comme un *vecteur ligne*, soit comme une *vecteur colonne* dans le langage de l'algèbre linéaire. Voyons maintenant comment on peut définir un tableau à 2 dimensions, c'est à dire une *matrice*.

Rappelez-vous encore une fois qu'en Python, une matrice de type (n, p), c'est à dire possédant n lignes et p colonnes, peut être définie comme une *liste de listes* : **MatriceL** = [L1, L2, ..., Ln] où les Li sont des listes qui représentent *les lignes* de la matrice **MatriceL**, chaque liste Li contenant p éléments.

Transformons cette liste de listes en un tableau numpy à deux dimensions. Prenons par exemple la matrice (2,5) :

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

Voici la procédure :

```
1 #Création d'un tableau à deux dimensions
2 import numpy as np
3 MListe = [ [1, 2, 3, 4, 5], [6, 7, 8, 9, 10] ]
4 M = np.array(MListe)    #Conversion
5 print(M)
6 print( type(M) )
```

Voici ce que cela donne :

```
In [1] : (executing lines 1 to 6 of "Prog.py")
 [ [ 1 2 3 4  5 ]
   [ 6 7 8 9 10 ] ]

<class 'numpy.ndarray'>
```

Pour le moment, rien de bien excitant si ce n'est qu'on a bien créé un tableau à deux dimensions, qui représente la matrice. Notez que les lignes de cette matrice sont bien rangées l'une sous l'autre et qu'il n'y a pas de virgule " , " entre deux lignes.

6.2.2 À partir de fonctions de numpy

Il y a cinq fonctions intéressantes :

La fonction arange :

Elle joue le même rôle que la fonction **range** qui générait des listes, mais c'est elle qu'il faut utiliser pour les tableaux de numpy. La syntaxe est (si on a fait **import** numpy **as** np avant) :

np.arange(*debut*, *fin*, *pas*)

Cela génère un tableau à une dimension (pouvant être interprété comme un vecteur ligne ou un vecteur colonne selon les cas, voir section 6.5 *Opérations sur les tableaux*) formé d'entiers ou de réels, commençant à *debut*, et finissant à *fin* (valeur exclue), avec un *pas*. Si vous n'indiquez pas la valeur de *pas*, Python considère implicitement que *pas* = 1. Prenons quelques exemples :

```
1  #Création d'un tableau avec arange
2  import numpy as np
3  a = np.arange(1, 7)   #Tableau formé des entiers de 1 à 6
4  b = np.arange(1, 2, 0.1)  # Tableau formé de réels
5  print(a)
6  print(b)
```

qui donne [6] :

```
In [1] : (executing lines 1 to 6 of "Prog.py")
[1 2 3 4 5 6]
[ 1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9]
```

Dans le premier cas, **a** est un tableau d'entiers. Dans le second cas, on a indiqué un pas de 0.1 dans **arange**, ce qui génère un tableau **b** de réels (**float**) compris entre 1.0 et 2.0 exclus, par *pas* de 0.1. Contrairement à ceux de la fonction **range**, les paramètres *debut*, *fin* et *pas* peuvent ici être des **nombres réels**.

Vous pouvez aussi avoir des valeurs négatives :

```
1  #Création d'un tableau avec arange. Valeurs négatives
2  import numpy as np
3  a = np.arange(-3.0, 4.0, 0.5)   #Tableau formé de réels
4  print(a)
```

ce qui donne :

```
In [1] : (executing lines 1 to 4 of "Prog.py")
[-3. -2.5 -2. -1.5 -1. -0.5 0. 0.5 1. 1.5 2. 2.5 3. 3.5]
```

On peut aussi obtenir un tableau dans l'ordre décroissant : il suffit que *debut* > *fin* et de mettre un *pas* négatif :

```
1  #Création d'un tableau avec ordre décroissant
2  import numpy as np
3  a = np.arange(5, -1, -0.5)   #Tableau formé de réels
4  print(a)
```

qui donne :

```
In [1] : (executing lines 1 to 4 of "Prog.py")
[ 5. 4.5 4. 3.5 3. 2.5 2. 1.5 1. 0.5 0. -0.5]
```

La fonction linspace :

C'est une fonction très utilisée car très pratique. Elle s'écrit **np.linspace**(*debut*, *fin*, *N*) et crée un tableau à une dimension formée de *N* nombres réels régulièrement répartis entre *debut* et *fin* inclue

6. Remarquez dans le deuxième exemple que 1. représente le nombre réel 1 pour le différentier de l'entier 1. Python préfère noter 1. plutôt que 1.0

(attention, c'est ici une exception à la convention générale de Python qui exclut d'habitude la dernière valeur. Ce n'est pas le cas ici et *fin* fait partie du tableau créé).

```
1 #Utilisation de linspace
2 import numpy as np
3 a = np.linspace(1, 5, 10)    #Tableau de 10 réels entre 1 et 5
4 print(a)
```

En voici le résultat :

```
In [1] : (executing lines 1 to 4 of "Prog.py")
[ 1. 1.44444444 1.88888889 2.33333333 2.77777778 3.22222222
3.66666667 4.11111111 4.55555556 5. ]
```

La fonction zeros :

np.zeros((n, p)) (attention aux parenthèses !) crée un tableau à deux dimensions (une matrice) de n lignes et p colonnes, dont les éléments sont 0. Vous pouvez indiquer *dtype* en paramètre pour préciser si vous voulez des valeurs entières, réelles ou complexes.

```
1 #Utilisation de np.zeros
2 import numpy as np
3 a = np.zeros( (3,3), dtype = int)    #Matrice du type (3, 3).
4 print(a)
```

ce qui donne :

```
In [1] : (executing lines 1 to 4 of "Prog.py")
[ [0 0 0]
  [0 0 0]
  [0 0 0] ]
```

La fonction ones :

Elle fonctionne de façon identique à **zeros** mais les éléments de la matrice générée sont tous égaux à 1

```
1 #Utilisation de np.ones
2 import numpy as np
3 a = np.ones( (3,4), dtype = float)    #Matrice du type (3, 4)
4 print(a)
```

ce qui donne :

```
In [1] : (executing lines 1 to 4 of "Prog.py")
[ [1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.] ]
```

La fonction diag :

Examinons-la sur un exemple :

```
1 #Utilisation de np.diag
2 import numpy as np
3 a = np.diag( [2, -1, 5] )    #Matrice diagonale
4 print(a)
```

ce qui donne :

```
In [1] : (executing lines 1 to 4 of "Prog.py")
[ [2  0 0]
  [0 -1 0]
  [0  0 5] ]
```

Cette fonction crée une matrice diagonale, dont les éléments de la diagonale sont ceux de la liste passée en paramètre ([2, -1, 5] dans notre exemple).

6.3 Les attributs size et shape

Une fois créé, un tableau est un objet de la classe **ndarray**. Comme tout objet, celui-ci possède des attributs (voir section 5) que l'on peut consulter (au moins en lecture). Deux de ces attributs sont intéressants : l'attribut **size** et l'attribut **shape**.

L'attribut size :

Si **M** est un objet de la classe **ndarray**, alors **M.size** contient la taille de ce tableau, c'est à dire son nombre d'éléments.

```
1  #Manipulation de l'attribut size
2  import numpy as np
3  M = np.array( [3, -2, 5] )
4  print(M)
5  print(M.size)    #Afficher la taille de M
6
7  M1 = np.ones( (3,3), dtype = int)    #Matrice (3, 3)
8  print(M1)
9  print(M1.size) #Afficher la taille de M1
```

```
In [1] : (executing lines 1 to 9 of "Prog.py")
[ 3 -2 5]
3
 [   [1   1   1]
     [1   1   1]
     [1   1   1]   ]
9
```

L'attribut shape :

De même, **M** étant un objet de la classe **ndarray**, **M.shape** est un attribut qui contient une donnée de type **tuple**, de la forme (n, p) où n est le nombre de lignes de **M** et p son nombre de colonnes.

```
1  #Manipulation de l'attribut shape
2  import numpy as np
3  M = np.ones( (3,2), dtype = int)    #Matrice (3, 2)
4  print(M)
5  print(M.shape) #Afficher le type de matrice
```

Voici ce que donne ce programme :

```
In [1] : (executing lines 1 to 5 of "Prog.py")
 [   [1   1]
     [1   1]
     [1   1]   ]
(3, 2)
```

La matrice **M** possède bien 3 lignes et 2 colonnes.

Une petite particularité à signaler pour cet attribut : si le tableau est à une dimension, alors **M.shape** contient le nombre d'éléments de ce tableau, sous la forme (*nombre_elements*,).

```
1  #Attribut shape pour tableau unidimensionnel
2  import numpy as np
3  A = np.array( [2, -3, 5] )
4  print(A)
5  print(A.shape) #Afficher l'attribut shape
```

Résultat :

```
In [1] : (executing lines 1 to 5 of "Prog.py")
[2 -3 5]
(3, )
```

6.4 La méthode reshape

Les tableaux étant des objets de la classe **ndarray**, ils possèdent aussi des **méthodes**. Nous en verrons plusieurs dans la section 6.6 *Opérations sur les tableaux*. Pour le moment nous allons voir une de ces méthodes, particulièrement intéressante lorsqu'on veut construire un tableau : la méthode **reshape**.

Prenez un tableau unidimensionnel, nommé **A** par exemple, contenant N éléments, du genre : **A** = [1.2 3.4 -2.1 6.7]. Lorsque $N = n \times p$, vous pouvez facilement le transformer en une matrice **M** de type (n, p) possédant n lignes et p colonnes grâce à la méthode :

A.reshape(n, p)

Cette méthode renvoie un nouvel objet tableau avec les bonnes dimensions. Vérifions-le en écrivant le programme suivant :

```
1 #Méthode reshape
2 import numpy as np
3 A = np.array( [1, 2, 3, 4] )
4 M = A.reshape( 2,2 )
5 print(A)
6 print(M)
```

qui donne :

```
In [1] : (executing lines 1 to 6 of "Prog.py")
[ 1 2 3 4 ]

[ [1 2]
  [3 4] ]
```

Vous avez transformé un tableau unidimensionnel de 4 éléments en une matrice de type (2,2). Sympa non ? En voici un autre exemple :

```
1 #Autre exemple de redimensionnement
2 import numpy as np
3 A = np.arrange(1, 11)
4 M = A.reshape(2,5) #Matrice 2 lignes, 5 colonnes
5 print(A)
6 print(M)
```

qui donne :

```
In [1] : (executing lines 1 to 6 of "Prog.py")
[ 1 2 3 4 5 6 7 8 9 10 ]

[ [1 2 3 4 5]
  [6 7 8 9 10] ]
```

6.5 Parcours de tableau

6.5.1 Cas d'un tableau unidimensionnel

Chaque élément dans un tableau unidimensionnel de N éléments possède un **rang**, qui va de 0 pour le premier à $N-1$ pour le dernier (comme pour les chaînes de caractères et les listes). On peut accéder en lecture et en écriture à l'élément de **rang** i du tableau **A** grâce à la syntaxe : **A**[i].

Par exemple :

```
1 #Accès à un élément d'un tableau
2 import numpy as np
3 A = np.arrange(1, 11)
4 print(A)
5 elem = A[0] #Premier élément
6 print(elem)
7 A[5] = -1 #On modifie l'élément de rang 5
8 print(A)
```

qui donne :

```
In [1] : (executing lines 1 to 8 of "Prog.py")
[ 1 2 3 4 5 6 7 8 9 10]
1
[ 1 2 3 4 5 -1 7 8 9 10]
```

Comme pour les chaînes de caractères et les listes, vous pouvez extraire un sous-tableau de **A** en écrivant :

$$\mathbf{A}[debut : fin]$$

qui créé un nouveau tableau composé des éléments allant du **rang** *debut* jusqu'au **rang** *fin - 1*. Attention donc : l'élément de **rang** *fin* ne fait pas partie du sous-tableau.

Si vous omettez *debut* ou *fin*, cela donne :

- **A**[: *fin*] est le sous-tableau composé des éléments allant du **rang** 0 à *fin - 1*. Cette syntaxe est identique à **A**[0 : *fin*].
- **A**[*debut* :] est le sous-tableau composé des éléments allant du **rang** *debut* jusqu'au dernier élément de **A**. Cette syntaxe est identique à **A**[*debut* : *N*] si **A** possède *N* éléments.

Illustration de suite :

```
1  #Parcours d'un tableau
2  import numpy as np
3  A = np.arrange(1, 11)
4  print(A)
5  B = A[2 : 5]
6  C = A[ : 7]
7  D = A[5 : ]
8  print(B)
9  print(C)
10 print(D)
```

Résultat :

```
In [1] : (executing lines 1 to 10 of "Prog.py")
[ 1 2 3 4 5 6 7 8 9 10 ]
[ 3 4 5 ]
[ 1 2 3 4 5 6 7 ]
[ 6 7 8 9 10 ]
```

6.5.2 Cas d'un tableau bi-dimensionnel (matrice)

Dans une matrice numpy de type (n, p), nommée **M** par exemple, chaque élément est repéré par son **rang de ligne** i et son **rang de colonne** j, avec i variant de 0 à $n-1$ et j variant de 0 à $p-1$. Vous pouvez accéder à un élément en particulier en tapant : **M**[i, j].

```
1  #Eléments d'une matrice
2  import numpy as np
3  M = np.array( [ [2.3, 1.4], [4.2, 1.8] ] )
4  print(M)
5  elem = M[1,1]
6  print(elem)
7  M[0,1] = -5.0
8  print(M)
```

En voici le résultat :

```
In [1] : (executing lines 1 to 8 of "Prog.py")

[  [2.3   1.4]
   [4.2   1.8]   ]

1.8
```

```
[ [2.3 −5.0]
  [4.2  1.8] ]
```

Faites toujours très attention : la première ligne ou première colonne porte le *numéro* 0.

Vous pouvez aussi extraire des sous-matrices à partir d'une matrice **M**, grâce à la syntaxe :

M[*deb_ligne* **:** *fin_ligne* **,** *deb_col* **:** *fin_col*]

(attention au placement des deux points " : " et de la virgule " , ").

Si vous omettez le paramètre de *debut*, çà commence à la première ligne (ou à la première colonne). De même, si vous omettez le paramètre de *fin*, çà va jusqu'à la dernière ligne (ou la dernière colonne).

```
1 #Extraction de sous-matrice
2 import numpy as np
3 A = np.arange(25) #Tableau de 25 éléments
4 M = A.reshape(5, 5) #Matrice (5, 5)
5 M1 = M[1 : 4 ,1 : 4]
6 M2 = M[ : 3 , 3 : ] #Lignes 0..2 Col 3-4
7 print(M)
8 print(M1)
9 print(M2)
```

ce qui donne :

```
In [1] : (executing lines 1 to 9 of "Prog.py")

[  [0  1  2  3  4]
   [5  6  7  8  9] ]
  [10 11 12 13 14] ]
  [15 16 17 18 19] ]
  [20 21 22 23 24] ]

[  [6  7  8]
  [11 12 13] ]
  [16 17 18] ]

[  [3  4]
   [8  9] ]
  [13 14] ]
```

6.6 Opérations sur les tableaux

Dans la suite, nous supposons que **A** est un tableau numpy. À partir de **A**, vous pouvez construire les tableaux $\mathbf{B} = x*\mathbf{A}$ ou $\mathbf{C} = \mathbf{A} + x$ où $x \in \mathbb{R}$ ou $\mathbb{N}$. Dans **B**, tous les éléments de **A** ont été multipliés par x et pour obtenir **C**, x a été ajouté à chaque élément de **A**.

```
1 #Deux opérations simples sur les tableaux
2 import numpy as np
3 Liste = [ [1, 2] , [3,4] ]
4 A = np.array( Liste )
5 B = 2*A  #Multiplions chaque élement par 2
6 C = A + 1  #Ajoutons 1 à chaque élément
7 print("A =", A)
8 print("B =", B)
9 print("C =", C)
```

En voici le résultat :

```
In [1] : (executing lines 1 to 9 of "Prog.py")
A = [ [1 2]
      [3 4] ]

B = [ [2 4]
      [6 8] ]

C = [ [2 3]
```

[4 5]]

Il faut faire attention à l'opération multiplication " * " : si **A** et **B** sont deux tableaux de même dimension (même nombre de lignes et de colonnes), alors **A*B** est le tableau dont les éléments sont les produits des éléments de **A** et **B**. Il s'agit donc d'une *multiplication élément par élément.*

```
1  #Multiplication de A et B
2  import numpy as np
3  L1 = [ [1, 2] , [3,4] ]
4  A = np.array( L1 )
5  L2 = [ [1, 2] , [1,2] ]
6  B = np.array( L2 )
7  C = A*B
8  print("A =", A)
9  print("B =", B)
10  print("C =", C)
```

et voilà :

```
In [1] : (executing lines 1 to 10 of "Prog.py")
A = [ [1 2]
      [3 4] ]

B = [ [1 2]
      [1 2] ]

C = [ [1 4]
      [3 8] ]
```

Si maintenant vous voulez faire le *produit matriciel* de **A** et **B**, noté **A.B**, il faut utiliser la **méthode dot** de **A**, qui prend la seconde matrice du produit (B ici) en paramètre :

$$\textbf{A.dot}(\mathrm{B}) \iff \mathrm{A.B} \quad \text{et} \quad \textbf{B.dot}(\mathrm{A}) \iff \mathrm{B.A}$$

Par exemple, **A** et **B** étant les mêmes matrices (2,2) que dans l'exemple précédent, c'est à dire :

$$\mathrm{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{et} \quad \mathrm{B} = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$$

vous pouvez directement taper dans l'interpréteur (normalement **A** et **B** sont déjà dans la mémoire de l'ordinateur) :

```
In [2] : C = A.dot(B)
In [3] : print(C)
Out [3] : [ [ 3   6 ]
            [ 7  14 ] ]
```

Çà marche ! **C** est bien le produit matriciel de **A** et **B**. Bien entendu, il est nécessaire que les dimensions des deux matrices soient compatibles avec cette multiplication matricielle.

Parlons un peu de l'action d'une matrice sur un vecteur ainsi que du produit entre un vecteur et une matrice. Pour rester concret, nous envisageons ici les deux types d'opérations suivantes :

$$\begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix} \underbrace{\begin{pmatrix} -1 \\ 3 \end{pmatrix}}_{\mathrm{V}} = \begin{pmatrix} 12 \\ -1 \end{pmatrix} \quad \text{(Action matrice sur vecteur)}$$

et

$$\underbrace{\begin{pmatrix} -1 & 3 \end{pmatrix}}_{\mathrm{V}} \begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix} = \begin{pmatrix} 18 & 1 \end{pmatrix} \quad \text{(Produit vecteur matrice)}$$

Dans le premier cas, le vecteur V est une vecteur colonne, c'est à dire une matrice (2, 1) et dans le second cas, le vecteur V est un vecteur ligne, c'est à dire une matrice (1, 2). Par la suite, nous noterons M la matrice :

$$M = \begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix}$$

Dans le module **numpy**, un vecteur est représenté par un tableau unidimensionnel, et **numpy** ne fait aucune différence entre un vecteur ligne ou un vecteur colonne. Les deux seront définis de la même façon en écrivant pour notre exemple :

V = **np.array**([−1, 3])

Ensuite, **numpy** s'adapte pour donner du sens aux deux opérations précédentes. La première s'écrira : **M.dot**(V) et la seconde **V.dot**(M)

Essayons !

```
1   #Action d'une matrice sur un vecteur
2   import numpy as np
3   L = [ [3, 5] , [7,2] ]
4   M = np.array( L )
5   V = np.array( [-1, 3] )
6   W = M.dot(V)
7   print(W)
8   print("——————————")
9
10  #Produit vecteur par une matrice
11  P = V.dot(M)
12  print(P)
```

Voici le résultat de ce programme :

```
In [1] : (executing lines 1 to 12 of "Prog.py")
[12 -1]
——————-
[18 1]
```

Tout va bien !

Vous pouvez alors vous douter que le produit scalaire de 2 vecteurs V et W pouvant s'écrire par exemple :

$$\underbrace{\begin{pmatrix} -1 & 3 \end{pmatrix}}_{V}\underbrace{\begin{pmatrix} 1 \\ 2 \end{pmatrix}}_{W} = 5$$

on peut le programmer de la façon suivante :

```
1   #Produit scalaire entre 2 vecteurs
2   import numpy as np
3   V = np.array( [-1, 3] )
4   W = np.array( [1, 2] )
5   ps = V.dot(W)
6   print("produit scalaire =", ps)
```

ce qui donne :

```
In [1] : (executing lines 1 to 6 of "Prog.py")
produit scalaire = 5
```

Vous auriez d'ailleurs aussi pu le programmer avec W.dot(V). Essayez pour voir !

Une dernière opération est la *transposition*. Si **M** est une matrice, il faut utiliser l'attribut **M.T** de l'objet **M** qui contient la matrice transposée. En reprenant la matrice **M** programmée plus haut et normalement encore dans la mémoire de l'ordinateur, nous obtenons :

In [4] **:** P = M.T
In [5] **: print**(P)
Out [5] **:** [[3 7]
[5 2]]

6.7 Fonctions agissant sur un tableau

Il y a des tas de fonctions de la bibliothèque numpy qui agissent directement sur tous les éléments d'un tableau numpy. Si vous avez importé numpy en le renommant np vous pouvez utiliser :

np.cos , **np.sin**, **np.exp**, **np.log**, etc...

Vous pouvez connaître toutes les fonctions disponibles en tapant **np.** dans l'interpréteur. Un menu contextuel apparaît à l'écran et affiche toutes les fonctions de np. Vous pouvez aussi taper **help**(np), ce qui affichera la même liste.

Prenons l'exemple de **np.sin** :

```
1  #Fonctionnement de np.sin
2  import numpy as np
3  from math import pi
4  X = np.linspace(0, pi, 10)
5  Y = np.sin(X)
6  print(Y)
```

ce qui donne :

```
In [1] : (executing lines 1 to 6 of "Prog.py")
[ 0.00000000e+00 3.42020143e-01 6.42787610e-01
 8.66025404e-01 9.84807753e-01 9.84807753e-01
 8.66025404e-01 6.42787610e-01 3.42020143e-01 1.22464680e-16 ]
```

Ligne **3**, on importe la constante pi du module math pour l'utiliser ligne **4** dans la construction d'un tableau unidimensionnel **X** de 10 nombres réels régulièrement espacés entre 0 et pi. À la ligne **6**, on crée le tableau **Y** dont les éléments sont les sinus des éléments de **X**.

Je vous invite à aller tester d'autre fonctions de numpy.

7 Lire et écrire un fichier

Nous allons nous intéresser dans cette section à l'écriture et à la lecture des données contenues dans un fichier placé sur un support de mémoire de masse, à partir d'un programme Python. Cela peut être très utile si vous souhaitez sauvegarder des données générées par votre programme.

De façon générale, un **support de mémoire de masse** est un appareil qui permet de stocker des données de façon durable : ces données persistent même après l'extinction de l'ordinateur. Vous connaissez bien entendu plusieurs exemples de support de mémoire de masse, comme les disques durs, les clés USB, les CD ou encore les DVD.

Commençons par quelques notions sur ce qu'on appelle *l'arborescence* des fichiers.

7.1 Arbre des fichiers

Sur un support de mémoire de masse, il y a deux types de structures : les **fichiers** et les **répertoires** (encore appelés **dossiers**) :

- Un **fichier** est comme une feuille de papier sur laquelle vous écrivez et vous lisez des données. Ces données peuvent être des chaînes de caractères, des données numériques, des objets, etc... Naturellement, cette feuille peut être très longue si le fichier est volumineux. Un fichier est caractérisé par son **nom** et par son **extension** : d'une façon générale, il s'appelle : *nom_fichier.extension*

Les extensions renseignent sur la nature du fichier et surtout sur le logiciel capable de lire les données de ce fichier. Vous connaissez bien sûr un certain nombre d'extensions : .doc (ou .docx) pour les fichiers Word, .jpg pour un fichier image ou encore .py pour un fichier python.

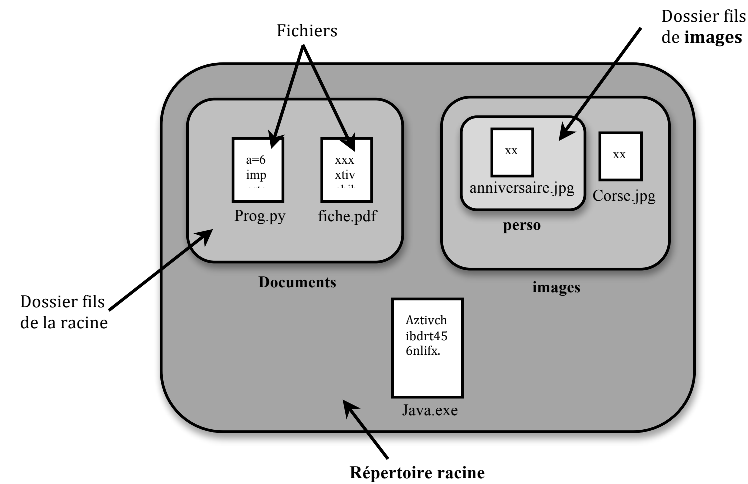
- Un **répertoire** aussi appelé **dossier** permet de regrouper plusieurs fichiers : c'est donc comme une grande pochette dans laquelle on dépose les fichiers. Un **répertoire** peut aussi contenir d'autres **répertoires** qui contiennent eux-même des fichiers et encore d'autres répertoires...

  Lorsqu'un dossier A contient un dossier B : on dit que B est un **dossier fils** de A (on dit aussi que c'est un sous-répertoire de A) et que A est le **dossier parent** de B.

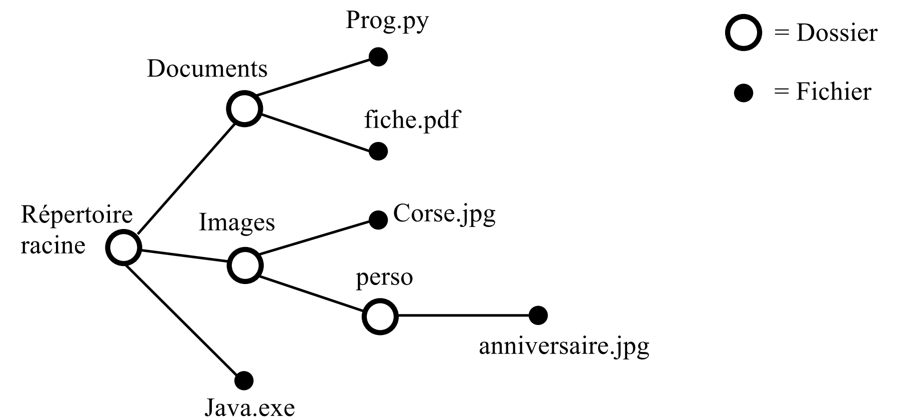
Dans un support de mémoire de masse, il y a toujours un répertoire qui contient tous les autres répertoires et les fichiers présents sur le disque : on l'appelle le **répertoire racine** (ou, plus simplement, **la racine**)

Par exemple, dans le système d'exploitation *Windows*, ce répertoire racine a pour nom[7] C : ou D : ou encore E : , F : , etc... (par exemple, le répertoire racine d'une clé USB insérée dans l'ordinateur va couramment s'appeler E : , F : ou G :).

Ainsi, l'organisation des dossiers et fichier sur une support de mémoire de masse a l'allure ci-dessous :



Une façon plus efficace de représenter cette organisation des fichiers est de la voir comme un **arbre**. Voici ce à quoi ressemble la structure décrite ci-dessus :



7. Windows, comme tous les systèmes d'exploitation, permet de partitionner le disque dur, c'est à dire de le diviser en plusieurs parties et de faire comme si il y avait plusieurs disques durs, chacun avec son répertoire racine. Dans ce cas les différents répertoires racines sont nommés : C : , D :, etc... Par défaut, lorsqu'il n'y a qu'un seul répertoire racine, celui-ci est appelé C :

- Tout part du **répertoire racine**.
- De chaque *répertoire* sont issues des **branches** qui aboutissent à des *sous-répertoires* ou à des *fichiers*.
- Aucune branche ne part d'un fichier. C'est ce qu'on appelle une **feuille** de l'arbre.

7.2 Chemin relatif et chemin absolu

7.2.1 Le chemin absolu

Si vous voulez accéder à un fichier, vous pouvez indiquer comment y arriver *à partir du répertoire racine* : vous indiquez alors la suite des dossiers menant au fichier. C'est ce qu'on appelle le **chemin absolu**.

Par exemple, sous Windows, si vous voulez accéder à un fichier nommé **fic.txt** contenu dans le dossier **essais**, lui même présent dans le répertoire racine **C :** . Le chemin absolu menant à ce fichier s'écrira [8] :

C :\essais\fic.txt

Le symbole " \" (appelé anti-slash) qui sépare les noms des répertoires et fichiers est typique du système d'exploitation Windows. Les systèmes d'exploitation UNIX, comme les ordinateurs Mac par exemple, utilisent le slash " / " comme symbole séparateur.

7.2.2 Le chemin relatif

Lorsque votre ordinateur est allumé, il y a toujours ce qu'on appelle le **répertoire de travail courant**. Ce répertoire est mémorisé par le système d'exploitation et, en général, il s'agit du répertoire dans lequel est placé le logiciel avec lequel vous travaillez actuellement.

Supposez par exemple que le logiciel avec lequel vous écrivez ou lisez un fichier soit dans le répertoire **essais** : il y a toutes les chances que ce soit le **répertoire de travail courant** [9].

Imaginez que ce dossier particulier contienne, en plus du fichier fic.txt, un sous-répertoire nommé **documents**, qui contient lui-même un fichier **fic2.py**. Vous avez donc affaire à une arborescence du style :

- C :
  - essais
    - fic.txt
    - documents
      - fic2.py

Si le fichier est déjà dans le répertoire de travail courant, le **chemin relatif** est simplement le nom du fichier. S'il est dans un sous répertoire de ce rép. de trav. courant , alors le **chemin relatif** s'écrit à partir du *répertoire de travail courant* (mais sans écrire son nom !) et vous indiquez la suites des répertoires menant au fichier cible.

Dans la convention Windows, qui utilise des " \", vous écrivez [10] :

- Chemin relatif de fic.txt : fic.txt
- Chemin relatif de fic2.py : \documents\fic2.py

8. Sous les systèmes Unix, le répertoire racine ne porte pas de nom (donc on laisse un vide à la place du nom de la racine) et il faut utiliser des "slash" / à la place des anti-slash \. Le chemin absolu s'écrira alors : /essais/fic.txt

9. À chaque moment, il n'y a qu'un seul répertoire de travail courant dans votre machine

10. Sous UNIX, il faudra bien sûr écrire : /documents/fic2.py

7.2.3 Comment connaître et changer le répertoire de travail courant ?

Vous n'êtes pas obligé de vous faire imposer le répertoire de travail courant ! Ben voyons, il ne manquerait plus que çà !
Il y a un moyen de connaître et de changer le répertoire de travail courant avec du code Python. Le module **os** est fait pour cela (os = operating system c'est à dire système d'exploitation).

Prenons un exemple : supposez que le répertoire de travail courant soit C :\Users\MP1 et que vous avez créé un dossier nommé **travail** sur le **Bureau** de votre ordinateur, lui même situé dans le dossier **MP1**. Très simple : vous pouvez écrire dans l'interpréteur ou dans votre fichier de programme les lignes suivantes :

In [1] **: import** os
In [2] **:** os.getcwd()
Out [2] **:** C :\\Users\\MP1
In [3] **:** os.chdir("C :/ Users/MP1/Desktop/travail")
In [4] **:** os.getcwd()
Out [4] **:** C :\\Users\\MP1\\Desktop \\travail

La fonction[11] **getcwd**() du module **os** renvoie le chemin absolu du répertoire de travail courant et la fonction[12] **chdir**(*nouveau_repertoire*) change le répertoire de travail courant : il faut lui indiquer en paramètre le chemin absolu du nouveau répertoire de travail courant que vous souhaitez, sous a forme d'une **chaîne de caractères** (**str**), c'est à dire "C :/Users/MP1/Desktop/travail" dans notre cas.

11. cwd : current work directory
12. chdir = change directory

Plusieurs remarques et mises en garde :

1. Windows ne faisant rien comme tout le monde, il utilise des anti-slash " \" comme caractère séparateur. Si vous voulez utiliser la fonction **os.chdir**, **il faut absolument mettre** des slash dans le chemin absolu et donc bien écrire "C :/Users/MP1/Desktop/...etc", sinon Python ne comprend pas et génère une erreur de syntaxe.
2. Lorsque la fonction **os.getcwd** renvoie le chemin du répertoire de travail courant, elle utilise " \\" comme symbole séparateur : bon ben c'est comme çà.
3. **Desktop** est le nom du bureau dans le langage de la machine (eh oui, les systèmes d'exploitation sont fabriqués par des anglo-saxons).

   Le chemin absolu pour aller sur le Bureau de votre ordinateur personnel dépend de votre configuration. Pour le connaître, le mieux est de cliquer avec le bouton droit de votre souris sur un fichier placé sur votre bureau : un menu contextuel apparait avec quelque part un truc du style **propriétés** ou **Lire les informations**. Vous cliquez dessus et vous allez voir s'affichez toutes les informations à propos de votre fichier et, en particulier, son chemin absolu.

7.3 Lire et écrire du texte dans un fichier

Nous allons commencer par nous intéresser à l'écriture et à la lecture d'un fichier **texte**. Ce type de fichier ne contient qu'une suite de caractères, regroupés dans une seule chaîne de caractères qui représente tout le texte.

Pour étudier cela , vous allez commencer par créer un dossier **test** que vous allez placer sur le Bureau.

Nous allons ensuite faire de **test** le répertoire de travail courant, en tapant dans l'interpréteur :

In [**1**] **: import** os
In [**2**] **:** os.chdir("C :/Users/.../Desktop/test")

Et voilà ! Vous allez maintenant écrire et lire des fichiers texte qui seront directement déposés dans ce dossier. Naturellement, le /.../ dépend de votre configuration personnelle : renseignez vous sur le chemin absolu vers le Bureau.

7.3.1 Ouvrir un fichier

Pour commencer à lire ou à écrire dans un fichier, *il faut d'abord l'ouvrir* : cela se fait avec la fonction **open** qui prend *trois paramètres* :

1. Le premier paramètre est le *chemin d'accès au fichier*, relatif ou absolu, *sous la forme d'une chaine de caractères*, avec le nom du fichier.

   Par exemple : "monTexte.txt" (chemin relatif) ou "C :/Users/.../Desktop/test/monTexte.txt" (chemin absolu) va ouvrir le fichier **monTexte.txt** situé dans le répertoire de travail courant **test**. Bien entendu, vous pouvez mettre le nom et l'extension que vous voulez.

2. Le second paramètre de la fonction **open** est le **mode** d'ouverture du fichier. Il s'agit de l'un des trois caractères : 'w', 'r' ou 'a' :

   – 'w' (ou "w" c'est la même chose) indique que le fichier est ouvert en écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, son ancien contenu est **effacé** et vous aller pouvoir réécrire dans le fichier comme s'il était neuf... (donc attention !).

   – 'r' indique que le fichier est ouvert en lecture. Vous ne pourrez que lire le contenu du fichier. Si celui ci n'existe pas, le programme s'arrête et une erreur de type **FileNotFoundError** est générée.

   – 'a' indique que le fichier est ouvert en mode **ajout** (append). Si le fichier n'existe pas, il est créé. S'il existe déjà son contenu n'est pas effacé et les données que vous y écrivez sont ajoutées en fin de fichier, à la suite de celles qui y sont déjà.

3. Le troisième paramètre est le **mode** d'encodage des caractères. Il faut que vous sachiez que, dans la machine, les caractères, par exemple "a", "§", "3", etc... , sont codés sous formes de *nombres entiers*, selon une table : à chaque caractère correspond un nombre unique qui permet de l'identifier. Le problème est qu'il y a plusieurs tables de codage, comme l'ASCII, l'ASCII-US ou encore l'utf-8. Certaines tables ont été développées par les américains comme l'ASCII-US et ne permettent pas de coder des caractères accentués comme les "é", "è" ou "à", présents dans la langue française mais inconnus des anglo-saxons. Je vous invite à faire des recherches sur internet pour voir comment ces tables de codage fonctionnent.

   Le troisième paramètre de **open** est donc le type de codage des caractères et, pour ne pas avoir de soucis, je vous conseille vivement d'utiliser la table utf-8 qui est universelle. Vous indiquerez donc à la fonction : encoding = "utf-8".

La syntaxe de la fonction **open** sera donc, par exemple pour ouvrir en écriture un fichier appelé **monTexte.txt** dans le répertoire de travail courant, avec un encodage utf-8 :

open("monTexte.txt", "w", encoding = "utf-8")

Que fait exactement la fonction **open** ? Eh bien elle renvoie un objet de classe **TextIOWrapper** qui gère les entrées et sorties de texte vers le fichier. Pour lire et écrire dans le fichier, on utilise alors les **méthodes** de cet objet. Voyons tout cela sur un exemple concret.

7.3.2 Écrire dans un fichier

Pour commencer, vous aller créer un fichier qui contient la chaîne de caractères : "Je viens d'écrire dans un fichier" (vous pouvez aussi prendre un exemple plus intelligent mais cette chaîne a le mérite de contenir le caractère accentué "é" qui n'est pas disponible dans toutes les tables d'encodage). Voici l'exemple :

```
1  #Création et écriture d'un fichier
2  mon_fichier = open( "monTexte.txt", 'w', encoding = "utf-8")
3  ch = "Je viens d'écrire dans un fichier"
4  mon_fichier.write(ch)
5  mon_fichier.close()
```

En voici le résultat :

In [1] : (executing lines 1 to 5 of "<programme.py>")
In [2] :

Python n'a rien dit de spécial : cela montre simplement qu'il a compris ce qu'on lui a dit de faire.

À la ligne **2** la fonction **open** crée un objet de la classe **TextIO-Wrapper** que nous récupérons dans la variable **mon_fichier**. À la ligne **4** nous utilisons la méthode **write** de mon_fichier pour écrire la chaîne **ch** dans le fichier. On finit le programme en fermant le fichier par la **méthode close** c'est à dire : **mon_fichier.close**().

Vous pouvez aller voir dans le dossier **test** et vous constaterez que le fichier **monTexte.txt** y a bien été créé !

Il est important de fermer un fichier à l'aide de la méthode **close** à la fin de son utilisation. Si vous ne le fermez pas, il sera impossible à ouvrir en lecture un peu plus loin dans le programme.

7.3.3 Lecture d'un fichier

Nous allons maintenant lire le fichier que nous venons de créer. Pour cela, il faut commencer par l'ouvrir en lecture grâce au paramètre "r", sans oublier d'indiquer à Python le type d'encodage pour qu'il puisse s'y retrouver. Voici ce que cela donne :

```
1  #Lecture du fichier
2  mon_fichier = open( "monTexte.txt", 'r', encoding = "utf-8")
3  contenu = mon_fichier.read()
4  print(contenu)
5  mon_fichier.close()    #Toujours refermer un fichier
```

ce qui donne :

In [2] : (executing lines 1 to 5 of "<programme.py>")
Je viens d'écrire dans un fichier

Cà marche ! Nous avons utilisé ici la méthode **read** de l'objet **mon_fichier** : cette méthode renvoie tout le contenu du fichier sous la forme d'une seule chaîne de caractères que nous stockons dans **ch**. Nous demandons ensuite d'afficher **ch** à l'aide de la fonction **print**.

Enfin, le programme se termine proprement par la fermeture du fichier au moyen de la méthode **close**.

Il faut que vous sachiez que, lorsque vous lisez un fichier au moyen de la méthode **read**, il y a un **curseur** qui se déplace dans le fichier et se positionne après le dernier caractère lu. Un nouvel appel de **read**

démarre la lecture à partir de cette position. Si la position du curseur est à la fin du fichier (plus aucun caractère à lire), **read** renvoie une *chaîne vide*.

De même on peut ne lire que les N premiers caractères d'un fichier texte en passant N en paramètre à **read**. Cela donne la syntaxe suivante :

mon_fichier.read(N)

Développons deux exemples :

```
1 #Lecture du fichier, suite
2 mon_fichier = open( "monTexte.txt", 'r', encoding = "utf-8")
3 contenu = mon_fichier.read()
4 print(contenu)
5 contenu = mon_fichier.read() #Deuxième lecture du fichier
6 print(contenu)
7 print("Fin du programme")
8 mon_fichier.close()    #On ferme la boutique!
```

qui donne :

In [3] : (executing lines 1 to 8 of "<programme.py>")
Je viens d'écrire dans un fichier

Fin du programme

Remarquez la ligne vide avant "Fin du programme". La deuxième lecture du fichier a renvoyé une chaîne vide dans **contenu** puisqu'on était déjà arrivé à la fin du fichier.

Recommençons encore une fois :

```
1 #Lecture du fichier, encore...
2 mon_fichier = open( "monTexte.txt", 'r', encoding = "utf-8")
3 contenu = mon_fichier.read(8) #Lecture des 8 premiers caractères
4 print(contenu)
5 contenu = mon_fichier.read(9)    #On lit les 9 caractères suivants
6 print(contenu)
7 contenu = mon_fichier.read(70) #Oups...beaucoup de caractères
8 print(contenu)
9 mon_fichier.close()
```

dont voici le résultat :

In [4] : (executing lines 1 to 9 of "<programme.py>")
Je viens
 d'écrire
 dans un fichier

La première lecture renvoie les 8 premiers caractères. La seconde lecture renvoie les 9 caractère suivants (le curseur s'est déplacé). La dernière lecture demande les 70 caractères suivants ce qui va au delà de la fin du fichier : **read** renvoie alors uniquement les caractères jusqu'à la fin du fichier (il n'y a donc pas d'erreur générée).

7.3.4 Écrire plusieurs lignes de texte dans un fichier

Continuons sur cette lancée et écrivons une seconde phrase dans le fichier **monTexte.txt**. Afin de ne pas l'effacer, nous allons l'ouvrir dans le **mode** 'a' (append) et y écrire : "J'y imprime la seconde phrase".

```
1 #Écriture d'une seconde phrase
2 mon_fichier = open( "monTexte.txt", 'a', encoding = "utf-8")
3 ch2 = "J'imprime une seconde phrase"
```

```
4  mon_fichier.write(ch2)
5  mon_fichier.close()
6
7  #On ouvre maintenant le fichier en lecture
8  mon_fichier = open( "monTexte.txt", 'r', encoding = "utf-8")
9  contenu = mon_fichier.read()
10  print(contenu)
11  mon_fichier.close()
```

En voici le résultat :

In [5] : (executing lines 1 to 11 of "<programme.py>")
Je viens d'écrire dans un fichierJ'y imprime la seconde phrase

Tout cela a bien marché...mais argh ! ! Les deux phrases ont été collées bout à bout, sans espace et sans saut de ligne !

Quand vous écrivez des instructions **write** les unes à la suite des autres, Python va venir coller les chaînes de caractères bout à bout dans le fichier. Une instruction **read** renverra toute la chaîne obtenue sans aucune séparation, ni aucun saut de ligne.

Si vous voulez quand même des séparations ou des sauts de lignes, il faudra les insérer vous-même !

- Pour une séparation, vous tapez un espace " " en fin ou en début de chaîne ou encore un **caractère de tabulation** "\t".

- Pour imposer un saut de ligne, il faut taper le caractère **saut de ligne** "\n" à la fin de la chaîne. Ce caractère est interprété par Python comme un saut de ligne.

Remarquez que nous avons introduit un nouveau type de caractère : '"\t" (qui se lit échappement t) ou "\n" (échappement n). Ce sont des exemples de *caractères non imprimables*. Dans une chaîne de caractères, il y a donc :

- Les *caractères imprimables* : "a', "é", "6", etc ... ils sont affichés tels quels à l'écran lors d'un appel de la fonction **print**.

- Les *caractère non imprimables* du genre "\n" qui sont interprétés comme des commandes du style "fais ceci ou fais cela...". Par exemple, si la fonction **print** rencontre "\n" dans une chaîne, elle saute une ligne avant d'afficher la suite. Si elle rencontre "\t", elle va créer un espacement avant d'afficher le caractère suivant.

Essayons !

```
1  #Écrasons le fichier et recommençons !
2  mon_fichier = open( "monTexte.txt", 'w', encoding = "utf-8")
3  ch = "Première phrase\nSeconde phrase\tpuis fin"
4  mon_fichier.write(ch)
5  mon_fichier.close()   #On ferme !
6
7  #On ouvre maintenant le fichier en lecture
8  mon_fichier = open( "monTexte.txt", 'r', encoding = "utf-8")
9  contenu = mon_fichier.read()
10  print(contenu)
11  mon_fichier.close()   #Je referme soigneusement
```

ce qui donne :

In [6] : (executing lines 1 to 11 of "<programme.py>")
Première phrase
Seconde phrase   puis fin

Tout va bien ! Le retour à la ligne a bien eu lieu et il y a eu un espacement entre "seconde phrase" et "puis fin".

Sachez que ces caractères non imprimables ne donnent des ordres de mise en forme de l'affichage qu'à la fonction **print**. Allez donc voir dans l'interpréteur en tapant directement **contenu** après l'exécution de votre programme (la variable **contenu** a été créée et chargée dans la mémoire de l'ordinateur : vous pouvez donc y accéder) :

In [7] : contenu
Out[7] : 'Première phrase\nSeconde phrase\tet fin'

On y retrouve tout ce qui y a été mis, y compris les caractères non imprimables !

7.3.5 La méthode readlines

Lorsque dans le fichier, les différentes lignes sont séparées par le caractère "\n", la méthode **readlines** renvoie une liste constituée des chaînes de caractères qui composent chaque ligne. Regardez cela sur l'exemple ci-dessous : on commence par effacer le contenu de l'ancien fichier et on y écrit une chaîne de caractères contenant 3 phrases sur 3 lignes :

```
1  #Lecture des lignes du fichier
2  mon_fichier = open( "monTexte.txt", 'w', encoding = "utf-8")
3  ch = "Première phrase\nSeconde phrase\nTroisième phrase"
4  mon_fichier.write(ch)
5  mon_fichier.close()   #Bien fermer le fichier
6
7  #On ouvre maintenant le fichier en lecture
8  mon_fichier = open( "monTexte.txt", 'r', encoding = "utf-8")
9  Liste_contenu = mon_fichier.readlines()
10  print(Liste_contenu)
11  mon_fichier.close()
```

ce qui donne :

In [8] : (executing lines 1 to 11 of "<programme.py>")
['Première phrase\n', 'Seconde phrase\n', 'Troisièmephrase']

En voilà une belle liste, avec nos trois phrases comme éléments !

7.4 Écrire autre chose que du texte

Bon, écrire ou lire du texte c'est bien, mais vous aurez sûrement besoin de sauvegarder autre chose que cela, par exemple des entiers, des nombres réels ou complexes, ou même des objets plus complexes comme des listes, des tableaux numpy, vos propres objets, etc ...

7.4.1 Enregistrer un objet dans un fichier

Il faut commencer par importer le **module pickle** en écrivant dans votre programme :

```
1  import pickle
```

Vous ouvrez ensuite votre fichier en **mode** *écriture binaire*. Pour cela il faut passer "wb" en second paramètre à la fonction **open** et il n'y a plus besoin d'écrire encoding = "utf-8" puisque ce ne sera plus un fichier texte :

```
2  fichier2 = open("Donnees.dat", 'wb')
```

Là j'ai mis une extension .dat mais vous pouvez mettre l'extension que vous voulez ou pas d'extension du tout. Comme le fichier n'existait pas avant, il va être créé dans le dossier **test** (sinon le contenu de l'ancien fichier serait effacé !).

Il faut ensuite créer un objet de la classe **Pickler** en lui passant en paramètre l'objet fichier nommé **fichier2** que nous venons d'ouvrir en écriture binaire :

```
3  obj_pic = pickle.Pickler(fichier2)
```

Dans la ligne de programme ci-dessus, la fonction **pickle.Pickler**(...) est la fonction **Pickler**(...) du **module pickle** (tout comme math.sin(...) est une fonction du module math) : elle renvoie un objet de la classe **Pickler** que l'on récupère dans la variable **obj_pic** (mais vous pouvez l'appeler comme vous voulez !).

Et maintenant, nous allons enregistrer deux objets listes dans le fichier **Donnees.dat** en utilisant la méthode **dump** de **obj_pic**. Voici le programme complet :

```
1  import pickle
2  fichier2 = open("Donnees.dat", 'wb')
3  obj_pic = pickle.Pickler(fichier2)
4
5  #On crée deux listes
6  Liste1 = ["arbre", 12, 1+2j, "truc"]
7  Liste2 = [0, 1, 2, 3, 4]
8
9  #Enregistrement des deux listes
10  obj_pic.dump(Liste1)
11  obj_pic.dump(Liste2)
12  fichier2.close()   #On n'oublie pas de fermer
```

À la fin de ce programme, vous devez avoir un nouveau fichier nommé **Donnees.dat** dans votre répertoire de travail courant **test**. Bon, ce fichier est illisible avec un éditeur de texte mais c'est normal : il s'agit d'un fichier binaire. Néanmoins, les deux objets ont bien été enregistrés dans ce fichier, l'un à la suite de l'autre.

Sachez que vous pouvez écrire des *objets de classes différentes* dans un même fichier avec cette méthode **dump** : vous pouvez par exemple y écrire une liste, suivie d'une chaîne de caractères, elle-même suivie d'un entier. Essayez pour voir !

Une dernière chose avant de passer à la suite. Si vous voulez ajouter un nouvel objet à la fin de votre fichier sans tout effacer, il faut l'ouvrir en **mode** *ajout binaire* en écrivant 'ab' :

fichier2 = **open**("Donnees.dat, 'ab')

7.4.2 Récupérer nos objets enregistrés

Nous allons maintenant récupérer nos deux listes. Pour cela, il faut utiliser un objet de la classe **Unpickler** définie dans le module **pickle**. En supposant que ce module a déjà été importé, nous écrivons :

```
1  #Récupération des objets
2  fichier2 = open("Donnees.dat", 'rb')
3  obj_unpic = pickle.Unpickler(fichier2)
```

Ligne **2**, le fichier est ouvert en **mode** *lecture binaire* 'rb' et on récupère l'objet dans **fichier2**...que l'on passe en paramètre à la fonction **Unpickler** du module **pickle**... pour créer l'objet **obj_unpic** ! Simple non ? !

Afin de récupérer nos deux objets, il faut utiliser la méthode **load** de notre objet **obj_unpic**. Cette méthode doit être appelée autant de fois qu'il y a d'objets à récupérer :

```
...
4  recup1 = obj_unpic.load()
5  recup2 = obj_unpic.load()
...
```

et voilà ! Liste1 est récupérée dans la variable recup1 et Liste2 est récupérée dans la variable recup2.

Voici le programme complet :

```
1  #Récupération des deux listes
2  import pickle    #Comme çà on est sûr qu'il est là
3  fichier2 = open("Donnees.dat", 'rb')
4  obj_unpic = pickle.Unpickler(fichier2)
5
6  #Début de la récupération
7  recup1 = obj_unpic.load()
8  recup2 = obj_unpic.load()
9
10  print(recup1)
11  print(recup2)
12  fichier2.close()    #On n'oublie pas de fermer
```

En voici le résultat :

```
In [7] : (executing lines 1 to 12 of "<programme.py>")
['arbre', 12, (1+2j), 'truc']
[0, 1, 2, 3, 4]
```

Tout s'est bien passé.

Faites cependant attention avec cette méthode **load**. Si vous l'appelez plus de fois qu'il n'y a d'objets à récupérer dans votre fichier, Python va générer une erreur **EOFError**, où EOF signifie fin de fichier [13]. Le programme s'arrête alors brutalement.

Il y a bien sûr d'autres méthodes dans ces modules de lecture et d'écriture. Si vous êtes intéressés, je vous invite à consulter l'aide de Pyzo ou à faire une recherche sur internet.

13. EOF = End Of File

8 Annexe : représentation d'un nombre réel en machine

8.1 Notation décimale d'un réel

Rappelons qu'un nombre réel r qui s'écrit par exemple, en notation décimale, $r = 34.213$ (on utilise la notation anglo-saxonne qui met un point à la place de la virgule) se calcule de la façon suivante :

$$r = 3 \times 10^1 + 4 \times 10^0 + 2 \times 10^{-1} + 1 \times 10^{-2} + 3 \times 10^{-3}$$

Plus généralement, la notation décimale "code" le réel sous la forme :

$$r = d_n...d_3d_2d_1d_0.d_{-1}d_{-2}d_{-3}...d_{-m}$$

où les d_i sont des **chiffres** pouvant prendre une valeur 0, 1, 2, ..., 8, 9.

d_0 est le chiffre des unités, d_1 celui des dizaines, d_2 celui des centaines, etc... Après le point, d_{-1} est le chiffre des dizièmes, d_{-2} celui des centièmes, etc...De façon générale, r se calcule par la relation :

$$r = \sum_{i=-m}^{n} d_i \times 10^i$$

8.2 Notation d'un réel en base deux

Il s'agit de la même logique de notation. Simplement, les chiffres d_i de la notation décimale sont remplacés par des chiffres b_i (dans ce cas, on parle plutôt de **bits**) ne pouvant prendre que les valeurs 0 ou 1 et, à la place d'avoir des puissances de 10, on utilisera les puissances de 2. Par exemple :

$$r = 101.1011$$

signifie :

$$r = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$$

La représentation du réel s'écrira de façon générale sous la forme :

$$r = b_n ... b_2 b_1 b_0 . b_{-1} b_{-2} ... b_{-m}$$

ce qui signifie que la formule permettant de calculer r est :

$$r = \sum_{i=-m}^{n} b_i \times 2^i$$

8.3 Codage binaire des nombres réels

Tout d'abord, l'ordinateur utilise la représentation en base deux. Il y a cependant plusieurs représentations possibles d'un même nombre réel r dans cette base. Par exemple :

$$0.110 \times 2^5 \quad \text{ou} \quad 110 \times 2^2 \text{ ou} \quad 0.0110 \times 2^6$$

sont trois représentations différentes d'un même nombre réel. Il est donc convenu de se ramener à la représentation suivante :

$$r = 1.b_{-1} b_{-2} .... b_{-m} \times 2^{\pm e}$$

On choisit donc **l'exposant** $\pm e$ de sorte que le **bit** des unités soit toujours $b_0 = 1$ et que les b_1, b_2, etc... b_i avec $i > 0$ soient tous nuls. Il ne reste alors que les bits des chiffres après la virgule et l'exposant.

Le nombre réel pouvant aussi être négatif, on partira sur une esquisse de codage de la forme :

$$r = \underbrace{\pm}_{\text{signe}} 1. \underbrace{b_{-1} b_{-2} .... b_{-m}}_{\text{mantisse}} \times 2^{\pm e}$$

Par exemple : -110.1001 sera transformé en -1.101001×2^2

La **mantisse** est l'ensemble des m bits $b_{-1} b_{-2} .... b_{-m}$ et l'**exposant** est le nombre $\pm e$ que l'on va aussi coder en base deux. Comme le 1 du b_0 est toujours présent, on l'omet car il est implicite ! Le codage se dirige alors vers la forme suivante :

Signe	Exposant	Mantisse

Dans cette notation :

- Le signe est codé par un seul bit : 0 pour un réel positif et 1 pour un réel négatif.
- L'exposant $\pm e$ est codé sur k bits.
- La mantisse est codée sur m bits

Comme le nombre de bits de codage de l'exposant est fini (k), on ne pourra pas représenter des nombres réels aussi grands ou aussi petits que l'on veut.

De même, le nombre fini de bits de la mantisse (m) entraîne une erreur d'arrondi.

8.4 Codage de l'exposant

L'exposant $\pm e$ est un entier positif ou négatif. On pourrait donc penser qu'on le code comme les entiers, en complément à deux s'il est négatif... mais ce n'est pas la solution qui a été retenue !

Un codage sur k bits offre 2^k possibilités : de $\underbrace{000...000}_{k \text{ bits}}$ à $\underbrace{111...111}_{k \text{ bits}}$.

Le type de codage retenu est le suivant :

- On introduit un **décalage** déc $= 2^k - 1$.
- La vraie valeur de l'exposant se calcule à partir de sa représentation sur k bits par la formule :

$$\text{valeur exposant} \;=\; \text{codage } k \text{ bits} - \text{ déc}$$

Prenons un exemple avec un codage sur 8 bits, ce qui donne : déc $= 2^7 - 1 = 127$.

- Un codage de l'exposant sous la forme 0000 0111 $= 2^2 + 2^1 + 2^0 = 7$ représentera une vraie valeur de l'exposant égale à :

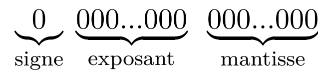
$$\text{valeur exposant } = 7 - 127 = -120$$

- Si maintenant le codage est 1100 0011 $= 2^7+2^6+2^1+2^0 = 195$, la vraie valeur de l'exposant sera donc :

$$\text{valeur exposant } = 195 - 127 = 68$$

Cependant, les deux valeurs extrêmes du codage de l'exposant, c'est à dire 0000...0000 (que des 0) et 1111...1111 (que des 1) sont réservées à des usages spéciaux :

- Le réel $r = 0$ pose un problème car en omettant le bit b_0 supposé être égal à 1, on ne fait plus la différence entre $r = 0.0$ et $r = 1.0$ On convient donc coder $r = 0$ avec uniquement des 0, aussi bien dans le signe que pour sa mantisse et son exposant. On aura donc :



L'exposant 000...000 est donc **réservé à ce nombre nul** (et à d'autres nombres appelés non normalisés dont nous ne parlerons pas ici).

- le codage maximal 111...111 (que des 1) de l'exposant est réservé au "nombre" ∞.

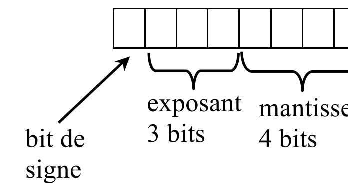
Prenons maintenant un exemple concret de codage d'exposant avec $k = 4$ bits. Le décalage est alors : déc $= 2^3 - 1 = 7$. On aura donc, valeur exposant = codage exposant - 7 :

Codage expos.	Valeur expos.	Codage expos.	Valeur expos.
0000	réservé à $r = 0$	1000	1
0001	-6	1001	2
0010	-5	1010	3
0011	-4	1011	4
0100	-3	1100	5
0101	-2	1101	6
0110	-1	1110	7
0111	0	1111	infini

Les valeurs -7 et 8 de l'exposant sont donc réservées à $r = 0$ et à ∞.

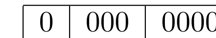
8.5 Exemples

Prenons un cas très simple, comme ci-dessous :

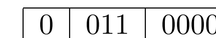


Le décalage vaut : déc $= 2^2 - 1 = 3$.

- Codage du réel $r = +0.0$. Par convention :



- Codage du réel $r = +1.0 = +1.0 \times 2^0$. Les codage de l'exposant est : $0 + 3 = 3$, c'est à dire 011. La mantisse est 0000 et le bit de signe vaut 0, d'où :



- Codage du réel $r = -3.125$ en base 10. Dans la base deux, r peut s'écrire : $r = -11.001 = -1.1001 \times 2^1$. Le codage de l'exposant est : $1 + 3 = 4$, c'est à dire 100. La mantisse est égale à 1001 et le bit de signe vaut 1. Il vient :

1	100	1001

8.6 Codage d'un réel en précision simple

Dans ce cas :

- Le nombre réel est codé sur 32 bits
- La mantisse occupe 23 bits
- L'exposant est codé sur 8 bits. Le décalage est donc : déc $= 2^7 - 1 = 127$.

Exemple : $r = 1\ 10000010\ 00110000000000000000000$

bit de signe = 1 $\Longrightarrow$ réel négatif. Exposant = $2^7 + 2^1$ - 127 = 130 - 127 = 3. Mantisse = $2^{-3} + 2^{-4} = 0.125 + 0.0625 = 0.1875$ à laquelle il faut ajouter 1 pour tenir compte du b_0 omis. Il vient donc :

$$r = -1.1875 \times 2^3 = -9.5$$

8.7 Codage d'un réel en double précision

C'est le type de codage utilisé par Python ! On a ici :

- Le nombre réel est codé sur 64 bits
- La mantisse occupe 52 bits
- L'exposant est codé sur 11 bits. Le décalage est donc : déc $= 2^{10} - 1 = 1023$.

Dans ce codage :

- Les plus petits nombres positifs et négatifs non nuls sont $\pm$ 2,2250738585072014 $\times 10^{-308}$.
- Les plus grands nombres positifs et négatifs sont $\pm$ 1,7976931348623157 $\times 10^{308}$.