Terminale S
Informatique et Sciences du Numériques

Chapitre 4
Introduction à Python

Ce cours est une introduction très succinte à Python3. Il est conseillé de lire ce cours avec une console Python sous la main.

Ce cours a été basé suite à la lecture :

- du livre *Apprendre à programmer avec Python*, de Gérard Swinnen, publié chez O'Reilly. Il peut être librement téléchargé ;
- du livre *Programmation en Python pour les mathématiques* de *Guillaume Conan, Pascal Chauvin et Alexandre Casamayou-Boucau* ;
- des excellents documents d'*Henri Garreta*, de *Patrick Fuchs* et du site *mégamath.*

Document sous licence Creative Commons BY-NC-SA (Paternité[1] - Pas d'utilisation commerciale - Partage des conditions initiales à l'identique[2])

Aymar de Saint-Seine
Année scolaire 2016 – 2017

1. l'œuvre peut être librement utilisée, à la condition de l'attribuer à l'auteur en citant son nom.
2. le titulaire des droits peut autoriser à l'avance les modifications ; peut se superposer l'obligation (SA) pour les œuvres dites dérivées d'être proposées au public avec les mêmes libertés (sous les mêmes options Creative Commons) que l'œuvre originale.

1 Mise en place

1.1 Obtenir Python

Pour obtenir et installer Python3, il suffit de télécharger la version 3 qui correspond à votre système d'exploitation à l'adresse http://www.python.org/ [3]

La documentation officielle et très copieuse peut être parcourue ou téléchargée à partir de ce site.

1.2 Utiliser Python

Que ce soit sur Windows ou sur Linux, après une installation réussie, vous avez surtout deux manières d'utiliser Python :

- Si vous n'avez jamais programmé, le plus simple pour exécuter des instructions Python est d'utiliser l'environnement spécialisé Idle. Sous Windows, le programme **Idle** s'installe en même temps que Python. Il suffit de lancer Idle depuis le menu *démarrer*. Sous Linux, il suffit de taper "python3" dans une console pour le lancer. Cet environnement se compose d'une fenêtre appelée indifféremment **console, shell** ou **terminal Python**.
  Comme on le voit ci-dessous, l'**invite de commande** se compose de trois chevrons (>>>) ; il suffit de saisir à la suite une **instruction** puis d'appuyer sur la touche "Entrée" de votre clavier pour l'executer.

  Nous ne dérogerons pas à la tradition informatique qui consiste à commencer l'apprentissage d'un langage par l'affichage de la salutation anglophone :

```
>>> print("Hello World !")
Hello World !
```

  Cet exemple permet de présenter la fonction print [4] qui permet d'afficher du texte ou le contenu d'une variable à l'écran.

- Si vous souhaitez faire des programmes un peu élaboré, le mieux est d'écrire le **code du programme** dans un **fichier** (appelé aussi un **script**) et ensuite d'exécuter ce script. Par exemple, on peut mettre le code ci-dessous dans un simple fichier texte que l'on nomme *presentation.py*. Remarquez l'extension du fichier.

```
nom = input(" Quel est ton nom ?")
print('Bonjour', nom)
print('Comment ça va ?')
```

3. En septembre 2016, les versions officielles de Python sont les versions 3.5.1 et 2.7.11. Attention, il s'agit de deux branches incompatibles : la version 3 est certainement "meilleure", mais certaines bibliothèques spécialisées ne fonctionnent qu'avec la version 2.

4. La fonction print() comporte de nombreuses options qui permettent de personnaliser la présentation des données. Pour plus d'informations, voir la section **Les primitives** **(page 20)**.

Pour exécuter le script, il suffit de taper dans un terminal *python3 presentation.py*.

```
c:> python3 presentation.py

Quel est ton nom ?Monty
Bonjour Monty
```

À noter que Idle permet l'ouverture et la gestion de fichier.
Dans le cas de projet encore plus élaboré (mélant différents modules et programmes qui s'appellent entre eux), l'utilisation d'un logiciel dédié à la programmation devient incontournable. Il en existe plusieurs. Nous utiliserons **Pyzo**[5] qui a l'avantage d'être un logiciel libre, portable et multiplateforme.

5. http://www.pyzo.org/index.html

2 Variables

2.1 Qu'est ce qu'une variable ?

Une variable est une zone de la mémoire dans laquelle une valeur est stockée. Aux yeux du programmeur, cette variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une adresse (i.e. une zone particulière de la mémoire).
En Python, la déclaration d'une variable et son initialisation (c.-à-d. la première valeur que l'on va stocker dedans) peut se faire en même temps. Pour vous en convaincre, regardez puis testez les instructions suivantes après avoir lancé l'interpréteur :

```
>>> x = 2
>>> id(x)
138405136
>>> x = 4
>>> id(x)
138405168
>>> x = 2
>>> id(x)
138405136
```

Lors de la première ligne, on a **déclaré la variable** x et on l'a **initialisé** à la valeur 2.
Lors de la troisième ligne, grâce à la id() que l'on a appelée juste avant, on sait que cette valeur est à la zone mémoire 138405136.
Lors de la quatrième ligne, on a modifié la valeur de x et la sixième ligne nous indique que la valeur 4 est à l'adresse 138405168.
Si on ré-affecte la valeur 2 à x, on s'apercoit que la zone mémoire sur laquelle pointe x est à nouveau 138405136. Cet exemple permet de montrer que c'est la valeur qui est fixée en mémoire et non la variable.

2.2 Nommer une variable

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que *altitude*, *altit* ou *alt* conviennent mieux que x pour exprimer une altitude.

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

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits, à l'exception du caractère "_" (underscore).
- La casse est significative (les caractères majuscules et minuscules sont distingués).
  Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. De même, éviter les accents et le caractère "_" qui peuvent poser des problèmes d'encodage.

N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatieres*.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 **mots réservés** ci-dessous (ils sont utilisés par le langage lui-même) :

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

2.3 Type d'une variable

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

Dans certains langages (C, Java ...), le type d'une variable doit être déclaré avant son affectation et il ne peut pas changer, on parle de **typage statique**.
Python est un langage à **typage dynamique**, c'est-à-dire que le type d'une variable peut changer après réaffectation. Cependant c'est un typage fort, par exemple on ne peut additionner une variable de type float et une variable de type string. En Python, comme dans la plupart des langages, les principaux types de base sont :

- le type entier **int**, réservé à une variable dont la valeur est un entier relatif stocké en valeur exacte. Il n'y a pas de limite de taille.
- le type flottant **float** pour un nombre à virgule, comme 3,14, stocké en valeur approchée sous la forme d'un triplet (s, m, e) sur 64 bits : 1 bit de signe s, 11 bits d'exposant e, 52 bits de mantisse m (voir chapitre 2 pour le codage numérique des floattants).
- Le type complexe **complex**, cité pour mémoire, correspond à une variable dont la valeur est un nombre complexe, stocké comme un couple de float (donc en valeur approchée). Le nombre complexe 3 + 4i est noté 3+4J , le nombre i est noté 1J (on peut remplacer J par j) ;
- le type booleen **bool**, pour des variables ou des expressions logiques comme $1 < 2$ qui peuvent prendre deux valeurs : vrai (True) ou faux (False). Pour plus de précisions sur les tests, voir la section 3.1
- le type **string** ou **str** pour les chaînes de caractères.

```
>>> x = 2 ; type(x)  #des nombres
<class 'int'>
>>> y = 3.14; type(y)
<class 'float'>
>>> type(1+2J)
<class 'complex'>
>>> a = 'bonjour' ; b = "c'est"; c = '''le "boss"''' #du texte
>>> print(a,b,c);type(a);type(b);type(c);
bonjour c'est le "boss"
<class 'str'>
<class 'str'>
<class 'str'>
>>> 1>2 ;  type(1>2); 2 == 5 #des booléens
False
```

```
<class 'bool'>
True
```

Remarques :

- L'exemple montre que l'on peut mettre plusieurs instructions sur la même ligne en utilisant le point-virgule (;) entre chacune. Il montre aussi l'utilisation du caractère # qui permet de mettre des commentaires dans le code. Tout ce qui suit ce caractère n'est pas interprété par Python.
- On a utilisé la fonction type() qui retourne le type de la variable (ou donnée) passée en paramètre.
- Python reconnaît certains types de variable automatiquement (entier, réel). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets simples, doubles voire triples (utile si le texte lui-même comprte des apostrophes ou des guillemets) afin d'indiquer à Python le début et la fin de cette chaîne.

On dit, excepté pour les chaines de caractères, que ce sont des **variables de type simple**.

Il existe d'autres types (par exemple, les **listes**, les **dictionnaires**, les **tuples**, les **ensembles** (set) ...) qui sont des **variables de type composé**. Nous en parlerons plus loin dans ce cours.

```
>>> a = [1,2,3];type(a)
<class 'list'>
>>> a = {'note1' : 15 , 'note2' : 17};type(a)
<class 'dict'>
```

On peut **forcer le type d'une variable** en utilisant les **primitives**[6] int(), float(), bool(), str(). De plus, Python convertit automatiquement les variables lors de certaines opérations (par exemple, lors d'une division comme illustré ci-dessous).

```
>>> x = 8; y = 4
>>> x/y; type(x/y)
2.0
<class 'float'> #car résultat d'une division
>>> z = int(x/y)
>>> type(z)
<class 'int'>
```

6. Une primitive est une fonction définie par défaut. Elle fait partie des fonctions utilisables directement, sans nécessité de charger un module spécial (voir section 5.1.1). Très souvent, on fait l'abus de language en utilisant le mot **fonction** au lieu de **primitive**.

2.4 Lire une entrée utilisateur

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc…). Dans un script en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la primitive input(). Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec **<Enter>**. Après l'appui sur la touche, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a entré. Cette chaîne peut alors être assignée à une variable quelconque, convertie, etc.

On peut utiliser la primitive input() en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur.

```
>>> prenom = input("Entrez votre prénom : ")
Entrez votre prénom :
```

Attention!!! la primitive input() renvoie toujours une chaîne de caractères.[7] Cela est source de nombreux bugs, comme dans l'exemple ci-dessous :

```
 >>> age = input("Donnez votre age : ")
Donnez votre age : 5
>>> age+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Python indique qu'il a eu un problème et vous donne le maximum d'indices dans son message d'erreur. En effet, la variable *age* est une chaine de caractère (type 'str') alors que la valeur '1' est automatiquement typé en tant qu'entier (type 'int'). Les variables n'étant pas de même type, Python ne peut les additionner (à moins de convertir l'une des deux ce qu'il ne veut pas faire implicitement d'après son message).
Afin de ne pas avoir de problème, il faut convertir la variable *age* du type chaine de caractère au type entier (ou float) en utilisant la fonction int().

```
>>> age = int(input("Donnez votre age : "))
Donnez votre age : 5
>>> age+1
6
```

7. Dans les versions de Python antérieures à la version 3.0, la valeur renvoyée par input() était de type variable, suivant ce que l'utilisateur avait entré. Le comportement actuel est en fait celui de l'ancienne primitive raw_input(), que lui préféraient la plupart des programmeurs.

2.5 Opérations

2.5.1 Sur les nombres

Les opérations de base se font de manière simple sur les types numériques (nombres entiers et réels) :

```
>>> x = 45;x+2
47
>>> y = 2
>>> x - y
45
>>> (x * 10) / y #résultat est de type flottant alors que les
    entrées sont de type entier
225
>>>x**2 # puissance d'un nombre
50625
>>> 34//5; 34%5 # quotient et reste de la division de 34 par 5
6
4
>>>x+=5;y-=2 #remplace x par x+5 et y par y-2
>>>x,y=y,x #échange le contenu des variables x et y
```

2.5.2 Sur les chaines de caractère

Pour les chaînes de caractères, deux opérations sont possibles : la **concaténation** et la **duplication**.

```
>>> a="Hello "; b='World !'
>>> a+b
'Hello World !'
>>> 3*a
'Hello Hello Hello '
```

L'opérateur de concaténation permet d'assembler deux chaînes de caractères et l'opérateur de duplication permet de répéter plusieurs fois une chaîne.
Attention à ne pas faire d'opération illicite car vous obtiendriez un message d'erreur :

```
>>> 'toto' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

On retrouve l'erreur déjà mentionnée plus haut.

On verra un peu plus loin dans le cours la notion de **liste** et de **slicing** (section 4.2). Pour Python, une chaine de caractère est une liste particulière[8]. L'idée principale est que l'on peut saucissonner une chaine de caractère en tranche numérotée de 0 à (*longeur de la chaine*)−1'. Si l'on compte depuis la fin de la chaine, on utilise un nombre négatif.

Ci-dessous quelques exemples pour éviter de long discours. Tout ceci est expliqué plus en détail à la section 4.2 (page 14).

```
 >>> s = 'Bonjour'
>>> s[0] # La "première" tranche
'B'
>>> s[-1] # La "première" tranche en partant de la fin
'r'
>>> s[3 :5] # avoir ce qui est entre la tranche 3 et la tranche 5
'jo'
>>> s[3 :] # avoir ce qui est après la tranche 3
'jour'
>>> s[ :5] #avoir ce qui est avant la tranche 5
'Bonjo'
```

Les chaînes de caractères sont immuables : une fois créées, il ne peut rien leur arriver. Pour modifier les caractères d'une chaîne, on doit construire une nouvelle chaîne qui, si cela est utile, peut remplacer la précédente. Par exemple, proposons-nous de changer en 'J' le 'j' (d'indice 3) de la chaîne précédente :

```
>>> s = 'Bonjour'
>>> s[3] = 'J'
  File "<stdin>", line 1
    s[3] = 'J'
            ^
SyntaxError: invalid character in identifier
>>> s = s[ :3] + 'J' + s[4 :]
>>> print(s)
BonJour
```

8. Un Tuple pour être plus rigoureux.

3 Structures de contrôle

Pour l'instant, toutes nos instructions sont exécutées les unes après les autres. Le programme commence par exécuter la première instruction, puis la deuxième et s'arrête lorsqu'il a fini la dernière instruction. Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Une **structure de contrôle** est une commande qui contrôle l'ordre dans lequel les différentes instructions d'un algorithme ou d'un programme informatique sont exécutées.
On peut globalement classer en deux catégories les structures de contrôle :

- les **structures alternatives** : exécuter un bloc d'instructions si une ou certaines condition(s) sont réunies.
- les **structures répétitives** : exécuter un bloc d'instructions à plusieurs reprises.

3.1 Syntaxe générale

Sous Python, les structures de contrôle ont toujours la même syntaxe :

- une ligne d'en-tête contenant une instruction formulée par un mot spécifique (`if`, `elif`, `else`, `for`, `while`, `def`, etc.) et se terminant par un double point.
- suivie d'une ou de plusieurs instructions (on parle de **blocs d'instructions**) indentées sous cette ligne d'entête exactement de la même manière (c'est-à-dire décalées vers la droite d'un même nombre d'espaces). Cette indention est très importante puisqu'elle délimite les blocs d'instructions.

Pour obtenir une disjonction de cas, on utilise en général un ou plusieurs opérateurs de comparaison. Ces opérateurs sont au nombre de huit :

x == y	x est égal à y
x != y	x est différent de y
x > y	x est strictement supérieur à y
x < y	x est strictement inférieur à y
x <= y	x est inférieur ou égal à y
x >= y	x est supérieur ou égal à y
x is y	id(x) == id(y)
x in y	x appartient à y (voir le type list)

Attention !! Bien distinguer l'instruction d'affectation "=" du symbole de comparaison "==". Ci-dessous, la première instruction est une déclaration et affectation, la seconde est une expression booléenne.

```
>>> x = 2; print(x)
2
>>> x == 3
False
```

Pour exprimer des conditions complexes, on peut combiner des variables booléennes en utilisant les trois opérateurs booléens (par ordre de priorité croissante) : `or` (ou), `and` (et), et `not` (non). Notez que Python, contrairement à beaucoup de langages, offre aussi quelques raccourcis syntaxiques agréables comme : (x<y<z) qui est un raccourci pour (x<y) and (y<z).

```
>>> x = -1
>>> (x > -2) and (x**2 < 5)
True
>>> (x <= -2) or (x**2 > 5)
False
>>> not(x >= -2)
False
>>> test = -2 < x <= 0 #utilisation d'une variable booléenne
>>> test
True
```

3.2 Structure alternative : If

En Python, l'entête d'une structure de choix commence par le mot clef if suivi d'une condition à valeur booleenne (**True** ou **False**) et se termine par le symbole " :".

Le bloc d'instructions qui suit s'exécute si et seulement si la condition de l'instruction de contrôle a pour valeur True. Si la valeur de la condition est False, on peut proposer l'exécution d'un bloc d'instruction alternatif après le mot clef else suivi de " :".

Pour les tests multiples, on peut enchaîner les if en cascade grâce à l'instruction elif, contraction de *else if*. Sur un exemple concret cela donne :

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

3.3 Structures itératives : For et While

Pour les structures de répétitions, il existe deux structures. La première est le for qui prend ses valeurs dans une liste : for variable in listeValeurs. À chaque itération, la variable prendra pour valeur un élément de la liste. Voici un exemple :

```
>>> for fruit in ['pomme','poire','kiwi']:
...     print(fruit)
...
pomme
poire
kiwi
```

Pour créer une boucle équivalente aux boucles traditionnelles "pour i variant de n à m, faire ... ", nous utiliserons la primitive range [9] qui construit une liste d'entiers sur demande. L'instruction fonctionne sur le modèle range([début,] fin[, pas]). Les arguments entre crochets sont optionnels.

```
>>> range(10) #liste de 10 entiers de 0 à 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(15,20) #liste des entiers de 15 inclus à 20 exclus
[15, 16, 17, 18, 19]
>>> range(0,1000,200) #liste des entiers de 0 inclus à 1000 exclus
    par pas de 200
[0, 200, 400, 600, 800]
>>> range(2,-2,-1)
[2, 1, 0, -1]
```

Ainsi, pour afficher les termes de la table de multiplication du 7 :

```
>>> for i in range(1,11) :
...     print(7*i,end=';')#pour avoir ';' en fin d'affichage
   plutôt que le retour à la ligne
7;14;21;28;35;42;49;56;63;70;
```

9. La primitive range construit effectivement en mémoire la liste demandée ; on peut regretter l'encombrement qui en résulte, alors qu'il nous suffirait de disposer successivement de chaque élément de la liste. La primitive xrange fait cela : vis-à-vis de la boucle *for* elle se comporte de la même manière que *range*, mais elle ne construit pas la séquence.

La seconde structure de boucle est le tant que : tant qu'une condition est vraie, on recommence l'exécution du bloc d'instruction. Contrairement à une structure *for*[10], on ne connait pas à l'avance le nombre de répétitions du bloc d'instruction.
Sur un exemple concret cela donne :

```
>>> nombreEntre = 0;
>>> while (nombreEntre != 47) :
...     nombreEntre = int(input("Tapez le nombre 47 ! "))
...
Tapez le nombre 47 ! 5
Tapez le nombre 47 ! 9
Tapez le nombre 47 ! 47
```

J'ai volontairement entré deux mauvaises valeurs avant de réussir brillamment le test !

Attention ! ! ! Si la condition mentionnée ne devient jamais fausse, le bloc d'instructions est répété indéfiniment ; on dit que le **programme boucle**. Quand on écrit une boucle, il faut impérativement s'assurer qu'elle va réellement s'arrêter (dans les conditions normales d'utilisation)[11]. Aussi on doit vérifier que durant l'exécution du bloc d'instruction, la valeur de la condition doit changer pour pouvoir sortir de la boucle. Dans l'exemple ci-dessous, la boucle en infinie.

```
i = 0
while i < 10:
   i = 1
```

10. Toute structure *for* peut être remplacée par une structure *while*. La réciproque n'est pas vraie.
11. Pour interrompre un programme (mal conçu) qui boucle, on utilise la combinaison de touches Ctrl-C.

4 Les listes

4.1 Définition

Une liste est une **donnée de type composé**. Elle contient une série de valeurs. Ces valeurs n'étant pas forcément du même type, ceci confère une grande flexibilité à ces listes. Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets. En voici quelques exemples :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ['girafe', 5, 'souris', 0.15]
>>> print(animaux)
['girafe', 'tigre', 'singe', 'souris']
>>> print(tailles)
[5, 2.5, 1.75, 0.15]
>>> print(mixte)
['girafe', 5, 'souris', 0.15]
```

Lorsque l'on affiche une liste, Python la restitue dans l'ordre et telle qu'elle a été saisie.

4.2 Slicing d'une liste

Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé **indice** (ou **index**) de la liste. Le principe et la syntaxe sont similaires à ceux utilisés pour les chaines de caractère.

liste :	['girafe'	, 'tigre'	, 'singe'	, 'souris']
indice :	0	1	2	3

Soyez très attentifs au fait que les indices d'une liste de n éléments commence à 0 et se termine à $n-1$. Voyez l'exemple suivant :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> animaux[0]
'girafe'
>>> animaux[1]
'tigre'
>>> animaux[3]
'souris'
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
>>> animaux[4]
Traceback (innermost last):
File "<stdin>", line 1, in ?
IndexError: list index out of range
```

N'oubliez pas ceci ou vous risqueriez d'obtenir des bugs inattendus !

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

liste :	['girafe'	, 'tigre'	, 'singe'	, 'souris']
indice positif :	0	1	2	3
indice négatif :	-4	-3	-2	-1

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez appeler le dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de la liste.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> animaux[-4]
'girafe'
>>> animaux[-2]
'singe'
>>> animaux[-1]
'souris'
```

Un autre avantage des listes est la possibilité de sélectionner une partie en utilisant un indiçage construit sur le modèle [m :n+1] pour récupérer tous les éléments, du m-ième au n-ième (de l'élément m inclus à l'élément $n+1$ exclus). On dit alors qu'on récupère une tranche de la liste, par exemple :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
>>> animaux[0:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[1:]
['tigre', 'singe', 'souris']
>>> animaux[1:-1]
['tigre', 'singe']
```

Remarquez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole " :", Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement. Dans les versions récentes de Python, on peut aussi préciser le pas en ajoutant un " :" supplémentaire et en indiquant le pas par un entier.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:3:2]
['girafe', 'singe']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
```

Finalement, on voit que l'accès au contenu d'une liste avec des crochets fonctionne sur le modèle liste[début :fin :pas].

4.3 Opérations sur les listes

Tout comme les chaînes de caractères, les listes supportent l'opérateur + de **concaténation**, ainsi que l'opérateur * pour la **duplication** :

```
>>> ani1 =['girafe','tigre']
>>> ani2 =['singe','souris']
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1*3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

L'exemple ci-dessous montre quelques **méthodes**[12] très utiles à connaitre lorsque l'on manipule des listes :

```
>>>l = [0, 2, 3, 4] ; len(l) # nombre d'éléments de la liste
4
>>> l.append(5) ; l # ajoute un nombre à une liste
[0, 2, 3, 4, 5]
>>> l.insert(1,9) ; l #ajoute un nombre à un emplacement donné
    dans une liste
[0, 9, 2, 3, 4, 5]
>>> l.extend([1,2]); l #concatenation de la liste [1,2] à la liste
     l
[0, 9, 2, 3, 4, 5, 1, 2]
>>> l.remove(3) ; l #supprime la première occurrence de la valeur
    passée en paramètre
[0, 9, 2, 4, 5, 1, 2]
>>> l.pop(3) ; l # supprime l'élément situé à l'indice indiqué en
    argument
[0, 9, 2, 5, 1, 2]
>>> l.index(5)#indice de la première occurrence d'un élément
3
>>> 2 in l ; 3 in l #teste si une valeur est dans la liste
True
False
```

Pour compléter, sachez que Python implémente un mécanisme appelé **compréhension de listes**, permettant d'utiliser une fonction qui sera appliquée sur chacun des éléments d'une liste. Voici un exemple :

```
>>> l=[0,1,2,3,4,5]
>>> carre=[x**2 for x in l]
>>> carre
[0, 1, 4, 9, 16, 25]
```

À l'aide de l'instruction *for x in l*, on récupère un à un les éléments contenus dans la liste ***l*** et on les place dans la variable ***x***. On élève ensuite chacune des valeurs au carré (***x**2***) dans un contexte de liste, ce qui produit une nouvelle liste. Ce mécanisme peut produire des résultats plus complexes.

12. Une méthode peut être vue comme une fonction qui ne s'applique qu'à des variables d'un certain type. Nous n'aborderons pas la programmation objet dans ce cours.

4.4 Listes de listes

Pour finir, sachez qu'il est tout-à-fait possible de construire des listes de listes. Cette fonctionnalité peut être parfois très pratique. Par exemple :

```
>>> enclos1 = ['girafe', 4]
>>> enclos2 = ['tigre', 2]
>>> enclos3 = ['singe', 5]
>>> zoo = [enclos1, enclos2, enclos3]
>>> zoo
[['girafe', 4], ['tigre', 2], ['singe', 5]]
```

Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie. Pour accéder à un élément de la sous-liste, on utilise un double indiçage.

```
>>> zoo[1]
['tigre', 2]
>>> zoo[1][0]
'tigre'
>>> zoo[1][1]
2
```

On verra un peu plus loin qu'il existe en Python les dictionnaires qui sont très pratiques pour faire ce genre de choses (voir section 7.1).

5 Les fonctions

L'un des concepts les plus importants en programmation est celui de **fonction**. Les fonctions permettent en effet

- de décomposer un programme complexe en une série de sous-programmes plus simples (lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite). Cela permet une compréhension et une programmation plus facile du programme.
- D'éviter de ré-écrire plusieurs fois les mêmes lignes de code. Si nous avons à faire plusieurs fois les mêmes opérations mais sur des variable différentes, il suffit de faire une fonction et de l'appeler pour chaque variable.
- de modifier rapidement un programme.

Avant d'entrer dans des explications plus poussées, un petit exemple des avantages des fonctions. On souhaite avoir l'affichage écran suivant :

```

----------------------------------------

Le vol en direction de Tokyo décollera à 9h00

----------------------------------------

Le vol en direction de Sydney décollera à 9h30

----------------------------------------

Le vol en direction de Toulouse décollera à 9h45

----------------------------------------
```

On peut obtenir ceci avec ce code fonctionnel mais peu pratique :

```python
print()
print("----------------------------------------")
print()
print("Le vol en direction de Tokyo décollera à 9h00")
print()
print("----------------------------------------")
print()
print("Le vol en direction de Sydney décollera à 9h30")
print()
print("----------------------------------------")
print()
print("Le vol en direction de Toulouse décollera à 9h45")
print()
print("----------------------------------------")
print()
```

ou écrire le code ci-dessous dans lequel on définit deux fonctions puis on les utilise

```python
#definition des fonctions
def tirerUnTrait() :
      print()
      print("-----------------------------------------")
      print()

def annoncerUnVol(destination,horaire):
    print("Le vol en direction de ",destination, " décollera à ",
       horaire)


#corps du programme
tirerUnTrait()
annoncerUnVol("Tokyo","9h00")
tirerUnTrait()
annoncerUnVol("Sydney","9h30")
tirerUnTrait()
annoncerUnVol("Toulouse","9h45")
tirerUnTrait()
```

Les avantages :

- Gain de temps à l'écriture du programme ;
- Gain de lisibilité du programme qui est moins long et mieux structuré ;
- Facilité de modification. Si on veut remplacer le trait par une suite d'étoile, il suffit de faire une seule modification dans la fonction plutôt que de rechercher et modifier dans tout le programme chaque endroit ou les traits sont indiqués.

5.1 Fonctions prédéfinies

5.1.1 Les primitives

Python offre des fonctions intégrées au module de base. Ces fonctions de base sont appelées des **primitives** (ou "built-in functions " en anglais). Nous avons déjà rencontré les primitives print() , id() , type() , input() , etc.

La primitive print() comporte de nombreuses options qui permettent de personnaliser la présentation des données. On retiendra parmi les plus pratiques :

- le paramètre sep=” qui permet d'indiquer quel caractère sépare les différents élements affichés par la fonction print() ;
- le paramètre end=' ' qui permet d'indiquer ce qu'il faut faire en fin de ligne (par défaut, l'instruction est \n qui indique de faire un retour à la ligne) ; Très utile pour faire des affichages en lignes.
- la méthode .format() qui permet de créer des chaînes de caractères en remplaçant certains champs (entre accolades) par des valeurs (passées en argument de la méthode format) après conversion de celles-ci.

Ci-dessous quelques exemples concrets d'utilisation :

```
>>> x = 3 ; y = 1000000000 ; z = 3.1416
>>> print(x,y,z) #affichage par défaut
3 1000000000 3.1416
>>> print(x, y, z, sep='; ') # séparateur est ";"
3; 1000000000; 3.1416
>>> print(x, y, z, sep='') # pas de séparateur
310000000003.1416
>>> print(x, y, z, sep='\n') # séparateur est "retour à la ligne"
3
1000000000
3.1416
>>> print('Bonjour');print('Marc') #affichage par défaut
Bonjour
Marc
>>> print('Bonjour',end=" ");print('Marc') #pour avoir affichage
    en ligne
Bonjour Marc
>>> x = 32
>>> nom = 'John'
>>> print("{0} a {1} ans".format(nom,x))
John a 32 ans
```

La méthode format offre la possibilité de préciser à l'intérieur de chaque accolade un code de conversion, ainsi que le gabarit d'affichage. Ci-dessous quelques exemples.

```
>>> x = 1037.123456789
>>> print('x vaut {:,}'.format(x)) # pour séparer les milliers par
    une virgule
x vaut 1,037.123456789
>>> print('x vaut {:.3f}'.format(x)) # pour fixer le nombre de
    décimales
x vaut 1037.123
>>> print('x et -x valent {0:+.3f} et {1:+.3f}'.format(x, -x)) #
    affiche toujours le signe
x et -x valent +1037.123 et -1037.123
>>> print('x vaut {0:20.3f}'.format(x)) # précise la longueur de
    la chaîne
x vaut             1037.123
>>> print('x vaut {0:>20.3f}'.format(x)) # justifié à droite
x vaut             1037.123
>>> print('x vaut {0:<20.3f}'.format(x)) # justifié à gauche
x vaut 1037.123
>>> print('x vaut {0:^20.3f}'.format(x)) # centré
x vaut     1037.123
>>> print('{0:*^16}'.format('mot')) # avec justification et
    remplissage
******mot*******
```

```
>>> print('{0:*<16}'.format('mot')) # avec justification et
    remplissage
mot*************
>>> print('{0:*>16}'.format('mot')) # avec justification et
    remplissage
*************mot
```

La liste complète et description des primitives est fournie par Python à la page :
http://docs.python.org/py3k/library/functions.html

5.1.2 Les modules

En complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des modules [13].
Ces modules couvrent des domaines très divers : mathématiques (fonctions mathématiques usuelles, calculs sur les flottants), administration système, programmation réseau, manipulation de fichiers, interface graphique, etc.
L'instruction `from ... import ...,...,...` permet de charger une ou plusieurs fonction(s) précise(s) d'un module.

```
>>> from math import sqrt, pi, sin, cos
>>> print(pi)
3.14159265359
>>> print(sqrt(5)) # racine carrée de 5
2.2360679775
>>> print(sin(pi/6)) # sinus d'un angle de 30 degrés
0.5
```

Nous pouvons même importer toutes les fonctions d'un module comme montré ci-dessous :

```
 >>> from random import *
>>> randint(1,6) #choisit un entier aléatoirement entre 1 et 6
6
>>> uniform(1,6) # choisit un nombre aléatoirement entre 1 et 6
5.704992681958675
>>> seq=[2,5,7]
>>> choice(seq) # choisit aléatoirement un élément dans la liste
    seq
5
```

13. La liste complète des modules de la bibliothèque standard est accessible à l'adresse : https://docs.python.org/3/library/index.html.
À cela s'ajoute les 82588 modules développés par la communauté Python : http://pypi.python.org/pypi

5.2 Créer ses propres fonctions

5.2.1 Syntaxe générale

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(argument1,argument2, ...):
    ...
    bloc d'instructions
    ...
```

Remarques :

- Comme les structures de controle *if*, *for* et *while* que vous connaissez déjà, l'instruction **def** est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
- Dans les parenthèses suivant le nom de la fonction, on indique les **arguments** nécessaires à la fonction. Leurs rôles est détaillés dans les paragraphes suivants. Les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.

Sur un exemple concret cela donne :

```
>>> def table7() :
...     for i in range(1,11) :
...         print(i*7,end=" ") #Utilisation de end=" " pour avoir
    un affichage en ligne
...
>>> table7()
7 14 21 28 35 42 49 56 63 70
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses, le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le **corps de la fonction** proprement dit).

5.2.2 Le passage d'arguments

Dans le précédent exemple, nous avons défini et utilisé une fonction qui affiche les termes de la table de multiplication par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Plutôt que de réécrire une nouvelle fonction, nous allons utiliser **le passage de paramètres** qui permettra de calculer n'importe quelle table de multiplication.

```
>>> def table(base) :
...     for i in range(1,11) :
...         print(i*base,end=" ")
...
>>> table(9)
9 18 27 36 45 54 63 72 81 90
>>> table(13)
13 26 39 52 65 78 91 104 117 130
```

À noter que l'argument que nous utilisons dans l'appel d'une fonction peut être une variable lui aussi. Analysez l'exemple ci-dessous. Que fait-il ?

```
>>> for k in range (1,11) :
...     table(k)
...     print("*****")
```

Il est possible de définir des **paramètres ayant une valeur par défaut**. Si ces paramètres ne sont pas spécifiés lors de l'appel à la fonction, ce seront les valeurs par défaut qui seront utilisées. La seule contrainte est que les paramètres ayant une valeur par défaut doivent être positionnés a la fin de la liste des paramètres :

```
def ma_fonction(a, b, c=2, d=3):
```

Cette fonction pourra être appelée de trois façons différentes :

```
>>> ma_fonction(0,1) #dans ce cas, on aura a=0, b=1, c=2, d=3
>>> ma_fonction(0,1,4) #dans ce cas, on aura a=0, b=1, c=4, d=3
>>> ma_fonction(0,1,4,5) #dans ce cas, on aura a=0, b=1, c=4, d=5
```

5.2.3 La récupération de valeurs

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des **procédures** [14]. Une "vraie " fonction [15] (au sens strict) doit en effet renvoyer une valeur lorsqu'elle se termine.

```
>>> def cube(w):
...     return w*w*w
...
>>> cube(4)
64
```

À titre d'exemple un peu plus élaboré, nous allons maintenant modifier quelque peu la fonction *table()* afin qu'elle renvoie elle aussi une valeur. Cette valeur sera en l'occurrence une liste (la liste des dix premiers termes de la table de multiplication choisie).

```
def table(base):
...     resultat = [] # # initialisation d'une liste vide
...     for i in range(1,11) :
...         b = i*base
...         resultat.append(b) # ajout d'un terme à la liste
...     return resultat
...
>>> table9=table(9)
>>> print(table9)
```

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

15. On remarquera l'analogie avec les mathématiques où une fonction est un objet qui à une variable (ou argument) renvoie une valeur.

```
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> table9[2]
27
```

Quel est l'intérêt ? Celui de désormais avoir à disposition la table de 9 dans une variable. La première fonction que nous avions fait ne faisait qu'afficher la table demandée. Sitôt cet affichage terminé, Python n'avait plus aucune mémoire de cette table. Désormais, on peut avoir accès à la table du 9 car elle a été sauvegardée dans la variable *table9*.

5.2.4 Portée des variables : Variables locales/globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales** à la fonction. Ces variables sont inaccessibles depuis l'extérieur de la fonction.
Dans l'exemple ci-dessous, la variable 'p' existe bien dans le corps de la fonction et peut-être affichée mais elle n'existe pas dans le programme principal. Sitôt la fonction terminée, la variable est effacée d'où le message d'erreur.

```
>>> def mask():
...     p = 20
...     print(p)
...
>>> mask()
20
>>> p
Traceback (most recent call last):
File "<stdin>", line 7, in <module>
NameError: name 'p' is not defined
```

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

```
>>> def mask():
...     p = 20
...     print(p)
...
>>> p= 15
>>> mask()
20
>>> print(p)
15
```

Analysons attentivement cet exemple : Une fois la définition de la fonction terminée, nous revenons au niveau principal pour y définir la variable p à laquelle nous attribuons la valeur 15. Cette variable définie au niveau principal est donc une variable globale.
Ainsi le même nom de variable p a été utilisé ici à deux reprises, pour définir deux variables différentes : l'une est globale et l'autre est locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction (où elles pourraient entrer en compétition),

ce sont les variables définies localement qui ont la priorité. On constate en effet que lorsque la fonction mask() est lancée, c'est la valeur attribuée localement de p qui est affichée. On pourrait croire d'abord que la fonction mask() a simplement modifié le contenu de la variable globale p (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction mask(), la variable globale p a conservé sa valeur initiale.

Tout ceci peut vous paraître compliqué au premier abord. Vous comprendrez cependant très vite combien il est utile que des variables soient ainsi définies comme étant locales, c'est-à-dire en quelque sorte confinées à l'intérieur d'une fonction. Cela signifie en effet que vous pourrez toujours utiliser une infinité de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs.

Cet état de choses peut toutefois être modifié si vous le souhaitez. Il peut se faire par exemple que vous ayez à définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction global. Cette instruction permet d'indiquer – à l'intérieur de la définition d'une fonction – quelles sont les variables à traiter globalement. Dans l'exemple ci-dessous, la variable a utilisée à l'intérieur de la fonction monter() est non seulement accessible, mais également modifiable, parce qu'elle est signalée explicitement comme étant une variable qu'il faut traiter globalement.

```
>>> def monter():
...     global a
...     a = a+1
...     print(a)
>>> a = 15
>>> monter()
16
>>> monter()
17
```

Par comparaison, essayez le même exercice en supprimant l'instruction global : la variable a n'est plus incrémentée à chaque appel de la fonction. Même si on appelle plusieurs fois la fonction, l'affichage sera toujours de 16.

5.3 La récursivité

On appelle **fonction récursive** une fonction qui comporte un appel à elle-même.
L'exemple le plus connu (mais pas le plus malin [16]) concernant la récursivité est celui de la fonction factorielle. Par définition, on a

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

Par exemple, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$. De la définition , on tire que $n! = n \times (n-1)!$.
Ci-dessous un exemple de programmation par récursivite de la fonction factorielle.

16. En effet, on connait parfaitement les points de départ et d'arrivée. Pourquoi redescendre jusqu'à 1, puis tout remonter, plutôt que de partir directement de 1 ? La bonne manière de programmer la factorielle de n, c'est une simple boucle for.

```
>>> def factorielle(n) :
...     if n==1 :
...         return(1)
...     else :
...         return(n*factorielle(n-1))
...
>>> factorielle(6)
720
```

Ci-dessous l'exemple un peu modifié. Regardez attentivement pour bien comprendre ce qui se passe :

```
>>> def factorielle(n) :
...     if n==1 :
...         return(1)
...     else :
...         print("calcul de factorielle(",n-1,")")
...         y=n*factorielle(n-1)
...         print("retour de factorielle(",n-1,")")
...         return(y)
...
>>> factorielle(4)
calcul de factorielle( 3 )
calcul de factorielle( 2 )
calcul de factorielle( 1 )
retour de factorielle( 1 )
retour de factorielle( 2 )
retour de factorielle( 3 )
24
```

La notion de récursivité n'est pas facile à comprendre. La vidéo

https://www.youtube.com/watch?v=oKpYx0_PcBE

permet de comprendre un peu mieux.
Une fonction récursive doit respecter ce qu'on pourrait appeler les **trois lois de la récursivité** :

1. Une fonction récursive doit s'appeler elle-même.
2. Une fonction récursive contient un cas de base.
3. Une fonction récursive doit au cours de ses appels successifs se se ramener à un moment donné au cas de base.

6 Gestions de Fichiers

6.1 Pourquoi utiliser des fichiers ?

Imaginons par exemple que nous voulions écrire un petit programme exerciseur qui fasse apparaître à l'écran des questions choisies au hasard avec traitement automatique des réponses de l'utilisateur. Comment allons-nous mémoriser le texte des questions elles-mêmes ?

- **Idée 1 :** L'idée la plus simple consiste à placer chacune de ces questions dans une variable, en début de programme, et d'afficher au hasard une des questions.
  Cette idée est malheureusement beaucoup trop simpliste. Cela va induire une très longue suite d'instructions if ... elif ... elif .. bien pénible à écrire comment on peut l'imaginer avec l'exemple ci-dessous.

```
import random

a = "Quelle est la capitale du Guatemala ?"
b = "Qui à succédé à Henri IV ?"
c = "Combien font 26 x 43 ?"


choix = random.randint(1,3)
if choix == 1:
    print(a)
elif choix == 2:
    print(b)
elif choix == 3:
    print(c)
```

- **Idée 2 :** Utiliser une liste.

```
import random

questions = ["Quelle est la capitale du
   Guatemala ?",
             "Qui à succédé à Henri IV ?",
             "Combien font 26 x 43 ?"]

choix = random.randint(0,2)
print(questions[choix])
```

  Bien que le code soit plus court, on peut encore noter quelques désavantages :
  - La lisibilité du programme va se détériorer très vite lorsque le nombre de questions deviendra important. En corollaire, nous accroîtrons la probabilité d'insérer une erreur de syntaxe dans la définition de cette longue liste. De telles erreurs seront bien difficiles à débusquer.
  - L'ajout de nouvelles questions, ou la modification de certaines d'entre elles, imposeront

à chaque fois de rouvrir le code source du programme.
- ◇ L'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.
- ◇ Dernier désavantage (non illustré par l'exemple) : Les variables étant supprimées à l'arrêt du programme, on ne peut pas mettre en place un tableau des meilleurs scores ou un historique des parties effectuées.

Ces remarques nous suggèrent la direction à prendre : il est temps que nous apprenions à séparer dans des fichiers différents les données et les programmes qui les traitent.

6.2 La pratique

Analogie du livre : L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver (à l'aide de son titre), puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux en même temps. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

En Python, l'accès à un fichier est assuré par l'intermédiaire d'un objet particulier appelé **objet-fichier** et créé à l'aide de la fonction `open(nom, mode)`. Cette fonction attend deux arguments, qui doivent tous deux être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture qui peut prendre plusieurs valeurs :
- r (de Read) : ouverture pour lecture seule ;
- w (de Write) : ouverture pour écriture : si le fichier n'existe pas, il est créé, sinon son contenu est écrasé ;
- a (de Append) : ouverture pour ajout : si le fichier n'existe pas, il est créé, sinon l'écriture s'effectue à la suite du contenu déjà existant ;
- r+ : ouverture pour lecture et écriture depuis le début du fichier avec écrasement de l'existant.
- w+ : ouverture pour lecture et écriture. Le contenu existant est supprimé.
- a+ : ouverture pour lecture et ajout en fin de fichier.

Une fois la lecture ou l'écriture dans le fichier terminée, on doit refermer le fichier à l'aide de la méthode `close()`.

Une fois l'objet fichier créé, on opère sur celui-ci par l'intermédiaire de différentes méthodes. Ci-dessous une description des principales méthodes.
- **méthodes de lecture :**
  - ◇ la méthode `read()` lit les données présentes dans le fichier et les transfère dans une variable de type chaîne de caractères. L'argument indique combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier. Si on utilise cette méthode sans argument, la totalité du fichier est transférée.
  - ◇ La méthode `readline()` ne lit qu'une seule ligne à la fois (en incluant le caractère de fin de ligne). La méthode readline() renvoie une chaîne de caractères.
  - ◇ La méthode `readlines()` permet de lire l'intégralité d'un fichier en une instruction seule-

ment. La méthode readlines() renvoie une liste de chaine de caractère. Chaque élément de cette liste correspond à un eligne du fichier.

- **méthodes d'écriture :**
  - ◇ La méthode write() réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument et de type chaine de caractère. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier à accès séquentiel). Chaque nouvel appel de write() continue l'écriture à la suite de ce qui est déjà enregistré.
  - ◇ La méthode writelines() écrit une liste de chaînes de caractères en les concaténant.
- **méthodes de déplacement :**
  Les méthodes seek() et tell() permettent respectivement de se déplacer au n^e^ caractère (plus exactement au n^e^ octet) d'un fichier et d'afficher où en est la lecture du fichier, c'est-à-dire quel caractère (ou octet) est en train d'être lu.

Exemples :

```
# Création d'un nouveau fichier et écriture
nom = 'mon_fichier.txt'
fichier = open(nom, 'w')
for i in range(1, 4):
  fichier.write('Ceci est la ligne ')
  fichier.write(str(i))
  fichier.write('\n') # caractère de fin de ligne
fichier.close()

# Lecture
fichier = open(nom, 'r')
liste = fichier.read()
print(liste)
fichier.close()
```

```
Ceci est la ligne 1
Ceci est la ligne 2
Ceci est la ligne 3
```

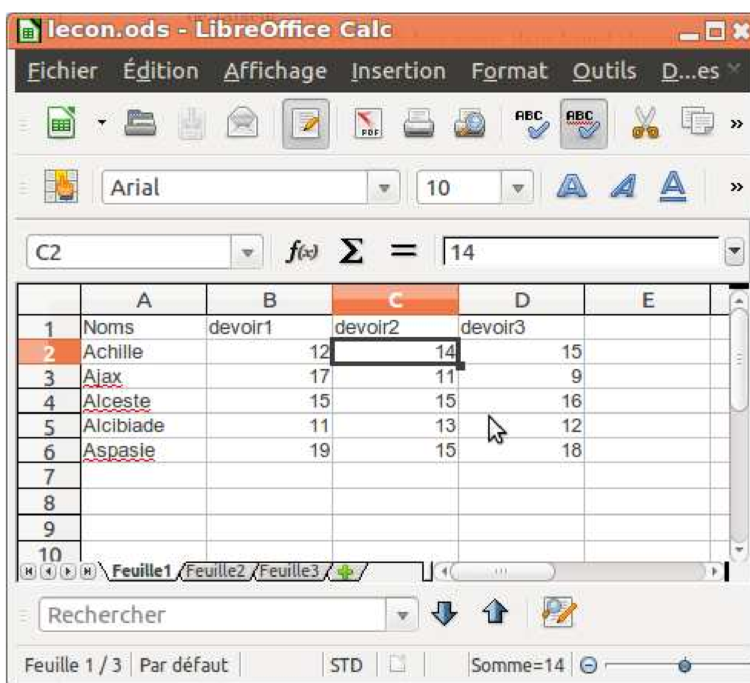
```
# Ajout en fin de fichier
fichier = open(nom, 'a')
for i in range(4, 6):
  fichier.write('Ceci est la ligne ')
  fichier.write(str(i))
  fichier.write('\n')
fichier.close()

# Variante pour la lecture
fichier = open(nom, 'r')
liste = fichier.readlines()
print(liste)
fichier.close()
```

```
['Ceci est la ligne 1\n', 'Ceci est la ligne 2\n', 'Ceci est la
  ligne 3\n', 'Ceci est la ligne 4\n', 'Ceci est la ligne 5\n']
```

6.3 Le format CSV

Le **format Comma-separated values** (CSV) est un format informatique ouvert représentant des données tabulaires sous forme de valeurs séparées par des virgules ; c'est le format le plus couramment utilisé pour importer ou exporter des données d'une feuille de calcul de tableur.
Un fichier CSV est un fichier texte, dans lequel chaque ligne correspond à une rangée du tableau et les cellules d'une même rangée sont séparées par une virgule. Dans un tableur, il suffit d'enregistrer une feuille de calcul en précisant le format CSV pour créer un tel fichier.
La figure suivante montre un exemple de feuille de calcul ouverte dans LibreOffice Calc :



Le fichier csv correspondant sera :
"Noms","devoir1","devoir2","devoir3"
"Achille",12,14,15
"Ajax",17,11,9
"Alceste",15,15,16
"Alcibiade",11,13,12
"Aspasie",19,15,18

Python comporte un module *csv*. Comme pour la gestion d'un fichier, il faut d'abord créer un objet-fichier avec la fonction open(nom). On dispose ensuite des méthodes suivantes :

- csv.reader() qui retourne une liste dont chaque élément est une ligne du fichier csv.
- csv.writer() pour écrire un flux csv.

7 Extra : Les dictionnaires et Tuples

7.1 Les dictionnaires

Un **dictionnaire** est une liste où l'accès aux éléments se fait à l'aide d'une clé (alphanumérique ou purement numérique). Il s'agit d'une association clé/valeur sous la forme clé :valeur. Les dictionnaires sont définis a l'aide des accolades. Dans d'autres langages les dictionnaires sont aussi appelés **tableaux associatifs** ou **tables de hachage**. Voici un exemple d'utilisation d'un dictionnaire :

```
>>> t ={'nom':'Valjean', 'prenom':'Jean'}
>>> t['nom']
'Valjean'
```

la fonction de del() permet de supprimer un élément et le mot clé , in permet de vérifier l'existence d'une clé :

```
>>> d={'cle1':1, 'cle2':2, 'cle3':3}
>>> del d['cle2']
>>> d
{'cle3': 3, 'cle1': 1}
>>> 'cle1' in d
True
>>> 'cle2' in d
False
```

Vous aurez pu remarquer lors de l'affichage du dictionnaire d que les couples clé/valeur n'étaient pas affichés dans l'ordre de création. C'est tout à fait normal car les dictionnaires ne sont pas ordonnés !
Plusieurs méthodes permettent de récupérer des listes de clés, de valeurs et de couples clé/valeur :

```
>>> courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
>>> courses.keys()
dict_keys(['poires', 'kiwis', 'pommes'])
>>> courses.values()
dict_values([5, 7, 3])
>>> courses.items()
dict_items([('poires', 5), ('kiwis', 7), ('pommes', 3)])
```

Ces méthodes renvoient des objets itérables : ce ne sont pas des listes et ils ne consomment donc pas autant de place mémoire que pour stocker une liste, on ne stocke qu'un pointeur vers l'élément courant. Avec ces structures vous pourrez toujours utiliser les boucles *for* classiques, par contre vous n'aurez plus un accès direct aux valeurs en spécifiant leur indice.

Dans un dictionnaire, pour obtenir la valeur correspondant à une clé on peut bien entendu utiliser la notation classique dictionnaire[clé], mais on peut aussi utiliser la méthode get() qui renvoie la valeur associée a la clé passée en premier paramètre ou, si elle n'existe pas, la valeur passée en second paramétré :

```
>>> courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
>>> courses.get('pommes', 'stock epuise')
3
>>> courses.get('salades', 'stock epuise')
'stock epuise'
```

On peut fusionner deux dictionnaires avec la méthode update() : Notez que si une clé existe déjà, la valeur stockée sera écrasée :

```
>>> courses ={ 'pommes':3, 'poires':5 , 'kiwis':7}
>>> courses2={'kiwis':3, 'salades':2}
>>> courses.update(courses2)
>>> courses
{'poires': 5, 'salades': 2, 'kiwis': 3, 'pommes': 3}
```

7.2 Les Tuples

Les tuples sont des listes non modifiables (les chaînes de caractères sont donc en fait des tuples particuliers). Pour les définir, on utilise des parenthèses et les éléments sont séparés par des virgules. Les éléments sont accessibles de la même manière que pour les listes. Notez que pour lever toute ambiguïté, un tuple ne contenant qu'un seul élément sera noté *(élément,)*. Si vous omettez la virgule, Python pensera qu'il s'agit d'un parenthésage superflu et vous n'aurez donc pas créé un tuple.

```
>>> t=(1,2,3)
>>> t[0]
1
>>> t[0]=0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

La primitive tuple() permet de convertir une liste en tuple.

```
  >>> l=[1,2,3]
>>> t=tuple(l)
>>> t
(1, 2, 3)
>>> l
[1, 2, 3]
>>> t=tuple('1234')
>>> t
('1', '2', '3', '4')
```

Tout comme avec les chaînes, il sera possible de concaténer des tuples et avoir la sensation d'avoir modifié un tuple alors que nous aurons simplement réaffecté une variable :

```
>>> t=(1,2,3)
>>> t=t+(4,5,6)
>>> t
(1, 2, 3, 4, 5, 6)
```

Quel peut être l'intérêt d'utiliser ce type plutôt qu'une liste ?
Tout d'abord, les données ne peuvent pas être modifiées par erreur mais surtout, leur implémentation en machine fait qu'elles occupent moins d'espace mémoire et que leur traitement par l'interpréteur est plus rapide que pour des valeurs identiques mais stockées sous forme de listes. De plus, de par leur aspect non modifiable, les valeurs contenues dans un tuple peuvent être utilisées en tant que clé pour accéder à une valeur dans un dictionnaire (alors que c'est beaucoup plus dangereux avec les valeurs contenues dans une liste, voir exemple ci-dessous) :

```
>>> l=['cle1','cle2','cle3']
>>> d={'cle1':1,'cle2':2,'cle3':3}
>>> d[l[0]]
1
>>> l[0]='cle_modifiee'
>>> d[l[0]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cle_modifiee'
```

8 S'aider soi-même

Un bon programmeur veille à ce que ses lignes d'instructions soient faciles à lire.

Pour ce faire, il est bon de prendre tout de suite l'habitude de bien présenter son code en Python selon les conventions suivantes :

- Bien commenter le code en indiquant régulièrement le role des instructions;
- Choisir des noms de variables explicites;
- Bien aérer le code. En particulier :
  - ◇ Avoir une taille fixe pour les indentations (4 espaces sont préconisées);
  - ◇ avoir des lignes pas trop longues (79 caractères maximum);
  - ◇ toujours placer une espace de chaque côté d'un opérateur (ex a = 3 plutot que a=3);
  - ◇ toujours placer une espace après une virgule, un point-virgule ou deux-points mais jamais avant;
  - ◇ ne pas placer d'espace entre le nom d'une fonction et sa liste d'arguments.

exemple :

```
def f(x) :
       return x**2
x=1 ; y=2;z = 3
x,y ,z
f  (z)
```

Listing 1 – Déconseillé

```
def f(x):
    return x**2
x = 1; y = 2; z = 3
x, y, z
f(z)
```

Listing 2 – Conseillé

Table des matières

1 Mise en place **2**
1.1 Obtenir Python . . . 2
1.2 Utiliser Python . . . 2

2 Variables **4**
2.1 Qu'est ce qu'une variable ? . . . 4
2.2 Nommer une variable . . . 4
2.3 Type d'une variable . . . 5
2.4 Lire une entrée utilisateur . . . 7
2.5 Opérations . . . 8
2.5.1 Sur les nombres . . . 8
2.5.2 Sur les chaines de caractère . . . 8

3 Structures de contrôle **10**
3.1 Syntaxe générale . . . 10
3.2 Structure alternative : If . . . 11
3.3 Structures itératives : For et While . . . 12

4 Les listes **14**
4.1 Définition . . . 14
4.2 Slicing d'une liste . . . 14
4.3 Opérations sur les listes . . . 16
4.4 Listes de listes . . . 18

5 Les fonctions **19**
5.1 Fonctions prédéfinies . . . 20
5.1.1 Les primitives . . . 20
5.1.2 Les modules . . . 22
5.2 Créer ses propres fonctions . . . 23
5.2.1 Syntaxe générale . . . 23
5.2.2 Le passage d'arguments . . . 23
5.2.3 La récupération de valeurs . . . 24
5.2.4 Portée des variables : Variables locales/globales . . . 25
5.3 La récursivité . . . 26

6 Gestions de Fichiers **28**
6.1 Pourquoi utiliser des fichiers ? . . . 28
6.2 La pratique . . . 29
6.3 Le format CSV . . . 31

7 Extra : Les dictionnaires et Tuples **32**
7.1 Les dictionnaires . . . 32
7.2 Les Tuples . . . 33

8 S'aider soi-même **35**