INF101 - INF131 - INF104
Algorithmique et programmation en Python
Cahier de cours

Responsable d'UE : Carole Adam
carole.adam@imag.fr

Septembre 2021

Polycopié conçu à partir du cours INF204 2016 créé par :
Lydie du Bousquet, Aurélie Lagoutte, Julie Peyre, Florent Bouchez Tichadou, Amir Charif, Grégoire Cattan, Emna Mhiri

Table des matières

1 Résumé de cours **5**
1.1 Quelques définitions . . . 5
1.2 Variables, expressions, instructions . . . 7
1.3 Outils utiles . . . 11
1.4 Entrées / Sorties . . . 12
1.5 Les expressions booléennes . . . 14
1.6 Instructions conditionnelles 'if' . . . 16
1.7 La boucle conditionnelle 'while' . . . 19
1.8 Les fonctions . . . 25
1.9 Chaînes de caractères . . . 32
1.10 Listes . . . 36
1.11 Boucle inconditionnelle : for . . . 42
1.12 Listes avancées . . . 44
1.13 Fonctions : compléments . . . 45
1.14 Dictionnaires . . . 48
1.15 Introduction aux fichiers . . . 51

2 Mémo (fourni aux examens) **53**

Chapitre 1

Résumé de cours

1.1 Quelques définitions

1.1.1 Programmes, compilateur et interprète

Les ordinateurs permettent d'automatiser des tâches. Mais ce sont des machines sans capacité d'initiative. Un ordinateur fait ce qu'on lui dit de faire. Pour cela, il faut lui parler dans un langage qu'il "comprend". On appelle ça, un **programme**.

De façon générale et naïve, un programme est une suite d'instructions (de commandes) respectant une syntaxe précise, qui sont exécutées de façon séquentielle (c'est-à-dire les unes après les autres). Ces instructions sont élémentaires : des affectations, des instructions conditionnelles, des répétitions (itérations).

Il existe de nombreux langages dans lesquels on peut exprimer des programmes : C, Java... Dans la suite du cours, on utilisera un langage appelé Python.

Une fois que l'on a écrit un programme (c'est un texte écrit dans une syntaxe très précise), on utilise un autre programme qui transforme le texte de ce programme (lisible par des humains) en un texte lisible par la machine (suite d'octets). Ce programme peut être un compilateur ou un interpréteur, en fonction du langage de programmation choisi.

- Le **compilateur** traduit une bonne fois pour toutes un code source en un fichier indépendant exécutable (donc utilisant du code machine ou du code d'assemblage). Par exemple C est un langage compilé.

- L'**interpréteur** est nécessaire à chaque lancement du programme interprété, pour traduire au fur et à mesure le code source en code machine. Python est un langage interprété.

1.1.2 Algorithme vs Programme

Comme dit précédemment, un programme est créé pour demander à l'ordinateur de faire une suite d'actions, en général pour résoudre un problème. Le programme (=le texte) va être lu et traduit par un compilateur ou un interpréteur, qui sont eux-mêmes des programmes. Compilateurs et interpréteurs attendent en entrée un texte sans fautes "d'orthographe" (*syntax error*) et qui résout le problème attendu. C'est parfois compliqué d'écrire directement un texte juste : qui résolve le problème et qui soit juste syntaxiquement (= sans faute d'orthographe). Rappelez-vous vos sujets de dissertations de philo, par exemple.

L'astuce quand on a un problème un peu compliqué est de travailler en 2 temps.

1. D'abord, on essaye de résoudre le problème (en s'autorisant des fautes d'orthographe, des approximations, des abréviations, en s'aidant de schémas), comme quand on rédige le plan de la dissertation avant de rédiger le texte final. C'est ce qu'on appelle un **algorithme**, et c'est indépendant du langage de programmation choisi.

2. Ensuite, on s'occupe de rédiger la solution finale au problème, en respectant la syntaxe exacte d'un langage. La même solution pourra ainsi être écrite dans différents langages de programmation. C'est ce qu'on appelle un **programme**, il n'est compréhensible que par le compilateur ou l'interpréteur du langage de programmation choisi, et il permet de faire faire ce qui est demandé à l'ordinateur.

On trouve des algorithmes partout dans la vie courante. Par exemple, en cuisine, un algorithme s'appelle une recette. En chimie ou en biologie, un algorithme s'appelle un protocole d'expérimentation. On peut aussi donner l'exemple qui consiste à indiquer le chemin à quelqu'un, qui peut se faire en plusieurs langues différentes, et selon plusieurs itinéraires distincts.

Exemple : Donner l'algorithme pour faire une omelette (un seul oeuf). Préciser : les ingrédients nécessaires, les actions à mener (dans l'ordre).

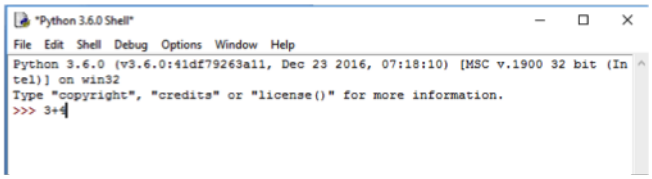
```
Algo Omelette_Elémentaire
Début
   {ingrédients}
      1 oeuf, sel poivre, beurre
   {ustensiles}
      1 saladier, une fourchette, une poêle, une spatule

   {procédure}
   Casser l'oeuf dans un saladier
   Saler et poivrer
   Battre l'oeuf à la fourchette
   Dans une poêle, faire chauffer le beurre,
   Verser l'oeuf battu dans la poêle,
   Cuire doucement jusqu'à l'obtention de la texture souhaitée
     {baveuse à bien cuite}
   Servir
Fin
```

Dans une recette de cuisine, on a généralement les ingrédients et les ustensiles nécessaires, la liste des actions à mener et parfois des commentaires. On retrouve les mêmes éléments dans les programmes et les algorithmes. On parle de déclarations (pour indiquer ce dont on va avoir besoin), d'instructions (les commandes) et de commentaires (informations pour l'humain qui va lire ou relire le programme).

1.1.3 En Python

Python est un langage interprété. On disposera de deux modes d'exécution d'un code Python :

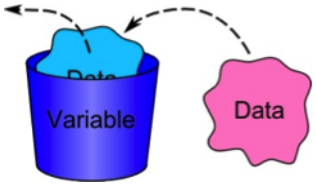
- soit on écrit un ensemble d'instructions dans un fichier puis on l'exécute
- soit on utilise l'interpréteur Python, instruction par instruction, un peu à la façon d'une calculatrice.



1.2 Variables, expressions, instructions

1.2.1 Variables

Une **variable** est un conteneur d'information qui est identifié par son nom, c'est un endroit pour ranger une valeur.



1.2.2 Identificateur

Pour nommer une variable, on utilise un **identificateur** : c'est une suite non vide de caractères respectant un certain nombre de contraintes :

- Doit commencer par un lettre ou le caractère _
- Doit contenir seulement des lettres, des chiffres et/ou le caractère _
- Ne doit pas être un mot-clé réservé de Python

Attention : les identificateurs sont sensibles à la casse, c'est-à-dire que les caractères majuscules et minuscules sont considérés comme différents (ex : ma_var $\neq$ Ma_Var).

Exemples :

- d'identificateurs valides : toto, prochaine_val, max1, MA_VALEUR, r2d2, bb8, _mavar
- d'identificateurs non valides : 2be, C-3PO (le tiret est considéré comme un opérateur 'moins'), ma var (l'espace sépare 2 identificateurs distincts)

Par convention, pour les variables en Python, on utilise des minuscules. On s'interdira également d'utiliser des accents. On essayera de choisir des noms parlants pour faciliter la lisibilité du code.
Standard : le standard `PEP8` [1] liste d'autres conventions et bonnes pratiques stylistiques en Python (cohérence de nommage des variables et fonctions, longueur des lignes, style des commentaires, etc).

1.2.3 Affectation, initialisation

Affectation. Pour mémoriser une valeur dans une variable, on fait une **affectation**, en utilisant le signe = (qui est ici très différent de celui utilisé en maths). Dans une affectation le membre de gauche (identificateur) reçoit la valeur qui est à droite (qui doit **d'abord** être évaluée). C'est-à-dire que cette valeur (une fois calculée si nécessaire) est stockée dans cette variable.

```
n = 33
a = 42 + 2 * 5
ch = "bonjour"
euro = 6.55957
```

Initialisation. La première affectation d'une variable donnée s'appelle l'**initialisation** de la variable. Comme son nom l'indique, la valeur d'une **variable** peut varier ensuite au cours de l'exécution d'un programme. La valeur antérieure est perdue, remplacée par la nouvelle valeur.

```
>>> a = 3 * 7   # l'expression est évaluée, et sa valeur affectée dans a
>>> a
21
>>> b = 7.3     # le séparateur décimal est un point . et non une virgule ,
>>> b
7.3
>>> a = b + 5
>>> a
12.3
>>> a = a * 3   # la nouvelle valeur peut dépendre de l'ancienne valeur
>>> a
36.9
```

1. Voir ici : `https://www.python.org/dev/peps/pep-0008/#naming-conventions`

Affectation vs comparaison. **Attention** : le signe "=" en Python sert **seulement** à faire une affectation. Si on veut savoir si deux nombres sont égaux, on utilise l'opérateur de comparaison "==".

```
>>> a = 6       # initialisation de a
>>> a
6
>>> b = 9       # initialisation de b
>>> a==b        # comparaison, renvoie un booléen
False
>>> a=b         # affectation, a reçoit la valeur de b
>>> a           # a vaut maintenant 9
9
```

1.2.4 Types

Les exemples ci-dessus montrent que les valeurs des variables peuvent être de plusieurs natures (ici entier, réel, chaîne de caractères). En informatique, on parle de **type**. Dans de nombreux langages de programmation, on doit déclarer le type des variables (typage statique).

Ce n'est pas le cas en Python. En Python, le typage est **dynamique**, c'est-à-dire que l'interpréteur déterminera automatiquement le type en fonction de la valeur qui a été affectée à une variable (et il change quand la valeur de la variable change). On peut connaître ce type en écrivant : `type(ma_var)`. Les types de base sont : entier (`int`), réel (`float`), chaîne de caractères (`str`) et booléen (`bool`). On en apprendra plus sur les booléens au chapitre 1.5, et sur les chaînes de caractères au chapitre 1.9.

```
>>> a = 17
>>> type(a)
<class 'int'>

>>> a = "salut"
>>> type(a)
<class 'str'>

>>> a = 3.14
>>> type(a)
<class 'float'>

>>> a = (21==7*3)
>>> a
True
>>> type(a)
<class 'bool'>
```

1.2.5 Expressions et opérateurs

Dans les exemples, nous avons montré qu'il était possible de faire des opérations, par exemple en écrivant `42 + 2 * 5`. Cela s'appelle une **expression**, c'est une "formule" qui peut être évaluée (calculée). Une variable ou une constante est aussi une expression, mais dont on n'a pas besoin de calculer la valeur (elle est déjà connue).

```
3
x
3*2.0 - 5
"bonjour"
20 / 3
x > 7
```

Dans ces expressions, on a des **opérandes**, et des **opérateurs**. Les opérateurs que l'on peut utiliser ne sont pas les mêmes selon le type des valeurs qu'on manipule. Quelques opérateurs :

- arithmétiques (sur des nombres, produisent des nombres) : addition +, soustraction -, multiplication *, puissance **, division réelle /, modulo %, division entière //
- de comparaison (sur nombres ou chaînes de caractères, produisent un résultat de type booléen) : == ,!=, < , >, <=, >=
- logiques (entre des booléens, résultat booléen) : `or` (disjonction), `and` (conjonction), `not` (négation). (Voir le chapitre sur les booléens)

Exemples d'opérateurs :

```
>>> 2 + 3       # addition
5
>>> 2 * 3       # multiplication
6
>>> 2 ** 3      # puissance
8
>>> 20 / 3      # division réelle
6.666666666666667
>>> 20 // 3     # division entière
6
>>> 20 % 3      # reste de la division entière (modulo)
2
>>> 2 > 8       # comparaison de 2 entiers, expression booléenne
False
>>> 'a' < 'z'   # comparaison de 2 caractères, expression booléenne
True
>>> 'b' < 'a'   # la comparaison respecte l'ordre alphabétique
False
>>> 2 <= 8 < 15 # comparaison de 3 entiers, expression booléenne
True
>>> (2 <= 8) and (8 < 15) # conjonction, expression booléenne équivalente à la précédente
True
>>> (x % 2 == 0) or (x >= 0) # disjonction, expression booleenne dont la valeur dépend de la
                             # valeur de la variable x (True si x pair ou positif)
>>> "a"+"b"     # concaténation de chaînes de caractères
"ab"
>>> "to"*3      # répétition de chaîne de caractères
"tototo"
```

Abréviations Quelques opérateurs permettent d'abréger les notations des affectations d'une valeur dans une variable qui dépend de son ancienne valeur. Par exemple :

```
>>> a = 3       # affectation
>>> a
3
>>> a += 1      # abréviation de a = a + 1
>>> a
4
>>> a *= 3      # abréviation de a = a * 3
>>> a
12
>>> a -= 5      # abréviation de a = a - 5
>>> a
7
```

Priorité des opérateurs Lorsqu'une expression comporte plusieurs opérateurs, afin de savoir dans quel ordre elle est évaluée, on considère la **priorité** des opérateurs. Par exemple en arithmétique vous avez appris que la multiplication est plus prioritaire que l'addition, ainsi $2*3+4$ est évalué en $(2*3)+4 = 6+4 = 10$ et non pas en $2*(3+4) = 2*7 = 14$. Le tableau 1.1 indique les priorités des opérateurs principaux de Python, en ordre décroissant. Quand des opérateurs sont de même priorité, l'expression est évaluée de gauche à droite. En cas de doute et pour améliorer la lisibilité, il est conseillé de parenthéser les expressions pour s'assurer qu'elles soient évaluées dans l'ordre souhaité.

```
>>> 8 // 4 // 2            # même opérateur, même priorité, évaluation de gauche à droite
1
>>> 8 // (4 // 2)          # les parenthèses forcent l'évaluation de droite à gauche
4
>>> 1 < 2 < 3 == 7%2 != 0  # les opérateurs de comparaisons sont aussi prioritaires
                           # évaluation de gauche à droite: True == 1 != 0
False
>>> (1<2<3) == (7%2!=0)    # comparaison des résultats booléens des 2 tests
True
```

Symbole	Nom
**	Puissance
+ unaire	Positif
- unaire	Négatif
*	Multiplication
/	Division
//	Division entière
%	Modulo
+ binaire	Addition
- binaire	Soustraction
<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal
!=	Différent
in	Appartenance
not	Négation booléenne
and	Conjonction booléenne (et)
or	Disjonction booléenne (ou)

TABLE 1.1 – Tableau de priorité des principaux opérateurs Python

1.2.6 Instructions

Une instruction est une **action** à exécuter par le programme. Un programme informatique est constitué d'une suite d'instructions exécutées dans l'ordre pour atteindre un résultat. Par exemple, une affectation de valeur dans une variable est une instruction. Dans la suite de ce cours nous allons apprendre :

- des types de données plus ou moins complexes (booléens, listes, dictionnaires, fichiers)
- d'autres **instructions**, comme les instructions d'entrées-sorties permettant l'interaction avec l'utilisateur du programme (lecture d'une valeur tapée au clavier, affichage d'une valeur à l'écran : cf chapitre suivant) ;
- des **structures de contrôle** qui permettent d'écrire des programmes plus complexes que de simples séquences d'instructions : exécution sélective / conditionnelle (if), répétition en boucle (for, while).

1.2.7 Les commentaires

Les commentaires sont des annotations du programme, qui ne sont pas analysées par l'interpréteur, mais qui servent à mieux comprendre le programme (pour vous, votre binôme, votre professeur...). En Python les commentaires s'écrivent en commençant une ligne par # pour dire à l'interpréteur de l'ignorer.

1.3 Outils utiles

C'est en programmant qu'on s'améliore ! Pratiquez autant que vous le pouvez, grâce aux outils ci-dessous ou à d'autres que vous pourrez trouver en ligne. Dans cette UE nous utiliserons un certain nombre d'outils :

- CaseIne `https://caseine.org/enrol/index.php?id=86`. Sur cette plateforme vous trouverez l'ensemble des documents du cours ainsi que beaucoup d'exercices d'entraînement auto-évalués. Il faut choisir "Connexion Grenoble" puis vous connecter avec vos identifiants Agalan. Vous devrez d'abord vous inscrire dans votre groupe de TD pour accéder à la suite.
- En TP, nous programmerons sous Idle, que vous pouvez aussi installer chez vous : `https://python.developpez.com/telecharger/detail/id/1925/IDLE`
- PythonTutor permet de visualiser l'exécution d'un programme Python pas à pas (penser à sélectionner Python 3.6 dans le menu déroulant) : `http://pythontutor.com/visualize.html#mode=edit`
- PLM (Programmer's Learning Machine) : `https://plm.telecomnancy.univ-lorraine.fr/#/`. Plateforme d'exercices en ligne. Changer Java en Python en haut à droite. Choisir les thèmes d'exercices en haut à gauche.

D'autres outils utiles :

- PyCharm, une IDE avec un bon débugueur, qui permet de voir la pile d'appels, et interfacée avec GIT (gestionnaire de versions) : `https://www.jetbrains.com/pycharm/` (version Community gratuite)
- Anaconda, une distribution de Python pour des applications scientifiques : `https://www.anaconda.com/distribution/`
- GIT est un gestionnaire de versions (utile par exemple pour un projet en groupe). Voir par exemple ce tutoriel des commandes basiques `https://rubygarage.org/blog/most-basic-git-commands-with-examples`, et cette interface graphique pour GIT `https://www.sourcetreeapp.com/` (outil gratuit pour Windows / Mac).

Des sites pour s'entraîner :

- Le Monde de Reeborg `http://reeborg.ca/reeborg.html` : une initiation graphique interactive à Python, en promenant un robot dans un monde simple.
- CodinGame `https://www.codingame.com/learn` fournit de nombreux exercices (bien sélectionner le langage Python).

1.4 Entrées / Sorties

Dans la plupart des cas on a besoin de pouvoir interagir avec un programme :

- Pour lui fournir les données à traiter, en général au clavier -> **entrées**
- Pour pouvoir connaître le résultat d'exécution d'un programme, ou pour que le programme puisse écrire ce qu'il attend de l'utilisateur, en général texte écrit à l'écran -> **sorties**

1.4.1 Les entrées

Pour gérer les entrées au clavier, on utilise la fonction `input()`. Quand l'ordinateur exécute la fonction `input()`, il interrompt l'exécution du programme, affiche éventuellement un message à l'écran (si demandé), et attend que l'utilisateur entre une donnée au clavier et la valide par un appui sur la touche `Entrée`.

Saisie textuelle et conversion de type. La fonction `input()` effectue une saisie **en mode texte** : la valeur saisie est considérée comme une chaîne de caractères. On peut ensuite changer son type, pour le convertir en nombre par exemple (attention cela ne fonctionne que si l'utilisateur a bien saisi un nombre, sinon on déclenche une erreur). Cette conversion est indispensable si on veut ensuite manipuler la valeur numérique.

```
>>> texte = input()
123                     # on suppose que l'utilisateur saisit 123 en réponse
>>> texte
'123'                   # la variable texte contient la chaîne de caractères '123'
>>> texte + 1           # erreur, addition chaîne + entier interdite
>>> val = int(texte)    # conversion de la chaîne en entier
>>> val + 1             # pas d'erreur, val est bien un entier
124
>>> y = int(input()) +1         # en une seule instruction
```

Message d'instructions. La fonction `input()` peut recevoir un paramètre optionnel indiquant le message à afficher. Il est toujours préférable de préciser un message, afin que l'utilisateur comprenne pourquoi le programme s'arrête et ce qui est attendu de lui.

```
>>> x = float(input("Entrez un nombre : "))
Entrez un nombre :              # le programme affiche le message et attend
12.3                            # on suppose que l'utilisateur répond 12.3
>>> x + 2
14.3
```

1.4.2 Les sorties

En mode "calculatrice", Python lit-évalue-affiche (comme fait dans les exemples précédents) mais quand on veut demander un affichage au sein d'un programme écrit dans un fichier (script), on utilise la fonction `print()`. Elle se charge d'afficher la représentation textuelle de n'importe quel nombre de valeurs fournies entre les parenthèses et séparées par des virgules (arguments). Par défaut, à l'affichage, ces valeurs sont séparées par un espace et l'ensemble se termine par un retour à la ligne.

```
>>> a = 20
>>> b = 13
>>> print("La somme de", a, "et", b, "vaut", a+b,".")
La somme de 20 et 13 vaut 33.
```

Modification de l'affichage Cependant on peut modifier ce comportement par défaut (séparation par des espaces et retour à la ligne final) en spécifiant les paramètres optionnels `sep` et/ou `end`. **Attention**, ces paramètres fonctionnent uniquement pour la fonction `print` (pas avec `input`). On peut aussi insérer manuellement des sauts de ligne en utilisant `"\n"` et des tabulations avec `"\t"` ; ce sont des "caractères spéciaux".

```
>>> print(a,b,sep=";")              # séparateur ; mais on garde le retour à la ligne final
20;13
>>> print("a=",a,"b=",b, sep="\n") # retour a la ligne entre chaque argument et final
a=
20
```

```
b=
13
>>> print(a,end="!")                # pas de retour a la ligne final
20!>>>
>>> c = 7
>>> print(a,b,c,sep=";",end=" !\n") # séparateur ; au lieu d'espace, et on rajoute un ! avant
                                    # le retour chariot final (qu'il faut spécifier avec \n)
20;13;7!
>>>
```

Alignement de l'affichage Il peut parfois être utile d'aligner les chaînes affichées, notamment quand elles sont de tailles différentes. Par exemple imaginons qu'on veuille afficher une liste d'entiers (colonne de gauche), leur carré (colonne du milieu), et leur cube (colonne de droite). L'affichage standard avec l'instruction `print(x,x**2,x**3)` produira un tableau peu lisible (à gauche ci-dessous).

La méthode `rjust()` permet de justifier l'affichage d'une chaîne de caractères à droite. Elle reçoit un argument qui indique la taille de la colonne dans laquelle on justifie à droite (et donc permet de déduire combien d'espaces il faut insérer à gauche). Ainsi l'instruction `print(str(x).rjust(2), str(x*x).rjust(3),str(x*x*x).rjust(4))` permet de justifier la première colonne (entiers à 1 ou 2 chiffres) sur 2 caractères, la 2e sur 3 caractères, et la 3e sur 4 caractères (taille maximale de 4 chiffres). Elle produit un tableau mieux aligné et plus lisible (à droite ci-dessous).

```
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Formatage Une autre manière de formater l'affichage consiste à utiliser l'opérateur % comme dans les exemples ci-dessous.

```
for i in range(7,11):                   # répétition, cf chapitre 10
    print('%2d' % i,'%3d' % i**2)       # affichage de l'entier i sur 2 chiffres
                                        # et de son carré sur 3 chiffres
 7  49
 8  64
 9  81
10 100
# affichage du nombre pi (fourni par le module math)
>>> print('%1.7f' % math.pi)    # avec 1 chiffre avant la virgule et 7 chiffres après
3.1415927
>>> print('%1.17f' % math.pi)   # avec 17 chiffres après la virgule
3.14159265358979312
```

Plus d'informations. Pour plus de détails sur l'affichage et le formatage des données, voir par exemple : `https://docs.python.org/fr/3/tutorial/inputoutput.html`

1.5 Les expressions booléennes

1.5.1 Définition : expression booléenne

Une expression qui ne peut prendre que les valeurs `True` (vrai) ou `False` (fausse) est appelée **expression booléenne**. En Python, il s'agit du type `bool` (booléen), vu au chapitre précédent.

```
>>> 1 < 2 < 3       # les entiers 1,2,3 sont-ils en ordre croissant strict
True
>>> 7%2 == 0        # 7 est-il multiple de 2 (reste de la division est nul)
False
```

1.5.2 Opérateurs booléens

OU logique : or. `expr1 or expr2` vaut vrai si et seulement si au moins une des deux expressions expr1 et expr2 est vraie, éventuellement les deux.

ET logique : and. `expr1 and expr2` vaut vrai si et seulement si les deux expressions expr1 et expr2 sont vraies.

Négation logique : not. `not expr` vaut vrai si et seulement si `expr` vaut faux.
L'opérateur de négation est le plus prioritaire, suivi de `and`, suivi de `or`. Par exemple `not a or b and c` est évalué comme `(not a) or (b and c)`.

1.5.3 Élément neutre des opérateurs booléens

L'élément neutre e d'un opérateur est celui tel que pour toute expression b, b OP e est égal à b.
Pour le ET l'élément neutre est donc True, car *True and b* vaut toujours b.
Pour le OU l'élément neutre est False, car *False or b* vaut toujours b.

1.5.4 Évaluation fainéante

En Python, les opérateurs `and` et `or` sont fainéants, c'est-à-dire que si l'évaluation de la première opérande permet déjà d'évaluer l'expression, alors la deuxième opérande n'est même pas évaluée.

```
# on suppose que la variable a n'est pas définie
>>> (2 == 1+1) or (a>=5)    # l'opérande gauche est vraie donc la disjonction est déjà vraie
True

>>> (3 == 1+1) or (a>5)     # l'opérande gauche est fausse donc il faut évaluer la droite
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> ( 2 == 1+1 ) and (a>5)  # la conjonction nécessite que les 2 opérandes soient évaluées à vrai
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> (3 == 1+1) and (a>5)    # l'opérande gauche est fausse donc la conjonction aussi
False
```

1.5.5 Lois de De Morgan

Les lois de De Morgan permettent de simplifier des négations d'expressions contenant une conjonction (and) ou une disjonction (or).

- not(expr1 or expr2) = not(expr 1) and not(expr2)
- not(expr1 and expr2) = not(expr 1) or not(expr2)

Exemples
not(a > 2 or b <= 4) équivaut à not(a>2) and not(b<=4) ce qui équivaut à (a <= 2) and (b > 4)
not(a > 2 and b <= 4) équivaut à not(a>2) or not(b<=4) ce qui équivaut à (a <= 2) or (b > 4)

1.5.6 Tables de vérité

Une table de vérité permet d'évaluer et de comparer des expressions booléennes. On commence par lister toutes les variables et toutes les combinaisons possibles de leurs valeurs (ci-dessous abrégées T pour True/vrai et F pour False/faux). Plus il y a de variables dans l'expression, et plus il y aura de combinaisons de valeurs possibles : pour n variables, il y a 2**n combinaisons, donc 4 lignes pour une table de vérité à 2 variables, 8 lignes pour 3 variables, etc.

Par exemple la table de vérité ci-dessous permet d'évaluer le 'ou exclusif' (noté xor) entre 2 variables a et b, défini comme 'soit a soit b est vraie' (`a or b`) 'mais pas les 2 en même temps' (`not (a and b)`).

a	b	a or b	a and b	not (a and b)	a xor b
V	V	V	V	F	F
V	F	V	F	V	V
F	V	V	F	V	V
F	F	F	F	V	F

TABLE 1.2 – Table de vérité pour le ou exclusif

1.6 Instructions conditionnelles 'if'

En programmation, on peut vouloir effectuer des actions différentes selon qu'une certaine condition est remplie ou pas. Par exemple : faire un traitement différent selon que la valeur d'une variable est positive ou non.

1.6.1 Syntaxe basique

L'instruction conditionnelle la plus simple en Python s'écrit comme ceci :

```
if condition :
    suite d'instructions si vrai
```

Dans ce cas, si la condition est vraie, la suite d'instructions est exécutées. Si la condition est fausse, rien n'est fait, le programme continue après cette instruction. La condition doit être une expression booléenne (par exemple le résultat d'une comparaison entre deux nombres, etc).

Attention à l'**indentation**, c'est-à-dire le décalage à droite et l'alignement de la suite d'instructions à l'intérieur du `if`. C'est l'indentation qui détermine dans quel bloc se trouve une instruction, et qui indique donc dans quel cas elle doit être exécutée.

1.6.2 Syntaxe si-alors-sinon

Si on veut faire quelque chose quand une condition est vraie, et autre chose quand elle est fausse, on peut rajouter un bloc `else` (sinon). Dans ce cas, si la condition est vraie, la première suite d'instructions est exécutée, alors que si la condition est fausse, c'est la deuxième suite d'instructions qui est exécutée. Le programme continue ensuite à l'instruction suivante. La syntaxe est la suivante :

```
if condition :
    suite d'instructions si vrai
else :
    suite d'instructions si faux
```

Indentation En Python, c'est l'**indentation** qui détermine à quel bloc appartient une instruction. Après la ligne `if condition`, les lignes suivantes sont indentées à droite pour signifier qu'elles appartiennent à ce bloc `if`. Si une instruction est alignée à gauche (au même niveau que le `if`), alors elle marque la fin de l'instruction conditionnelle. Pensez donc toujours à bien indenter votre code (ce qui le rend aussi beaucoup plus lisible).

Exemple

```
x = 5
if x > 0 :
    print(x, "est plus grand que 0")    # dans le bloc if
    print("il est strictement positif") # dans le bloc if
else :                                  # sinon (donc si la condition est fausse)
    print(x, "est négatif ou nul")      # dans le bloc else
print("Fin")                            # ni dans if ni dans else, toujours exécutée,
                                        # après la fin de l'instruction conditionnelle
```

Attention, on peut avoir un `if` sans `else`, mais pas l'inverse. Si on ne veut faire aucun traitement quand la condition est fausse, alors il ne faut pas écrire de `else`. On ne peut en aucun cas écrire un bloc `else` vide ! Si on ne veut exécuter une instruction que quand la condition est fausse, il faut écrire un `if` avec la négation de cette condition.

1.6.3 Cas multiples

Pour enchaîner les conditions, on dispose également du mot clé `elif` (contraction de `else if`). Les parties `elif` sont optionnelles, comme la partie `else`. On peut mettre plusieurs blocs `elif` pour distinguer autant de conditions que nécessaire. Par contre il ne peut y avoir qu'un seul `else` (cas par défaut), qui signifie que **toutes** les conditions précédentes sont fausses.

Remarque : la condition d'un bloc `elif` n'est évaluée que si les conditions précédentes ont échoué. Le programme ne pourra rentrer que dans **un seul** bloc d'une instruction conditionnelle (le premier dont la condition est vraie). Il n'est donc pas nécessaire de spécifier la négation des conditions précédentes dans les conditions suivantes. Par exemple :

```
note = int(input("Quelle est votre note en maths ?"))

# ce qu'il ne faut pas faire
if note<10:
    print("Vous n'avez pas la moyenne")
# on teste inutilement si note>=10
elif 10<=note<12:
    print("Pas mal...")
# on teste inutilement si note>=12
elif 12<=note<15:
    print("Mention Bien !")
# on teste inutilement si note>=15
elif note>=15:
    print("Vous avez la bosse des maths !")

# la version correcte
if note < 10:
    print("Vous n'avez pas la moyenne")
# si le premier test échoue, on sait déjà que note>=10
# on n'a donc pas besoin de le vérifier dans la condition du elif
elif note < 12:
    print("Pas mal...")
elif note < 15:
    print("Mention Bien !")
# si le 2e et le 3e tests échouent aussi, on sait que note >= 15, un else suffit
else:
    print("Vous avez la bosse des maths !")
```

1.6.4 Cas par défaut

Attention : en l'absence d'un bloc `else` (cas par défaut) il est possible que le programme ne rentre dans **aucun** des blocs d'une instruction conditionnelle (si toutes les conditions sont fausses). Il est donc en général préférable de toujours prévoir un cas par défaut (bloc `else`). Par exemple :

```
reponse = input("Faites-vous du sport régulièrement ? (oui/non) ")
if reponse=='oui':
    print("Super, c'est bon pour la santé !")
elif reponse=='non':
    print("Dommage, vous devriez vous y mettre...")
# en l'absence d'un bloc else, le programme n'affichera rien
# si l'utilisateur répond autrement que par 'oui' ou 'non'
# il faut rajouter un cas par défaut pour gérer les exceptions
else:
    print("Je n'ai pas compris la réponse")
```

Exemple : calculer le nombre de racines réelles d'un polynôme du second degré. On sait qu'il y a 3 cas selon que le déterminant est strictement positif, nul, ou strictement négatif.

```
a = 3.2   # coefficient du monome de degre 2
b = 5     # coefficient du monome de degre 1
c = -7.9  # coefficient du monome de degre 0
d = b**2 - 4*a*c
if d>0 :
    print("Deux racines reelles distinctes")
elif d==0 :
    print("Une seule racine reelle")
else :    # ici on a forcement d < 0
    print("Aucune racine reelle")
```

Attention : les `elif` sont suivis d'une condition, par contre le `else` n'est pas suivi d'une condition. Il correspond exactement à la négation de la condition du `if`, ou à la négation du `if` et de tous les `elif` précédents : on y rentre par défaut si on n'a pu rentrer dans aucun bloc précédent de l'instruction conditionnelle.

1.6.5 Imbrication

Remarque : on peut **imbriquer** les instructions conditionnelles, c'est-à-dire écrire un `if` dans le bloc d'instructions d'un autre `if`. Cela permet de distinguer des sous-cas, par exemple une fois que je sais que ma variable est positive, je peux vouloir distinguer les valeurs paires ou impaires.

Attention à l'indentation. A chaque nouveau bloc imbriqué dans le précédent, on décale d'un cran supplémentaire vers la droite.

Exemple

```
print("Testeur de parite")
x = int(input("Entre un entier positif"))
if x>=0 :           # x est positif
    if x%2==0 :     # x est positif ET pair
        print(x,"est pair")
    else :          # x est positif ET non pair (donc impair)
        print(x,"est impair")
else :              # x n'est pas positif (donc negatif)
    print("Erreur !",x,"est negatif")
```

1.6.6 Nombres aléatoires

Le module `random` permet de générer des nombres pseudo-aléatoires. Il fournit par exemple les fonctions suivantes qui seront utilisées en TD et TP :

- `randint(a,b)` : renvoie un entier pseudo-aléatoire entre les bornes a et b (incluses).
- `random()` : renvoie un réel pseudo-aléatoire entre 0 et 1.
- `choice(l)` : renvoie un élément au hasard parmi ceux de la liste l.

1.7 La boucle conditionnelle 'while'

Précédemment, nous avons appris à utiliser les instructions conditionnelles (`if-elif-else`), qui permettent d'exécuter une suite d'instructions uniquement **si** une certaine condition est vérifiée. On appelle ces instructions des **structures de contrôle**. En programmation, il existe d'autres structures de contrôle, les **boucles** (ou itérations), qui permettent de répéter un ensemble d'instructions plusieurs fois. En particulier la boucle *while* permet de répéter un bloc d'instructions **tant que** une certaine condition (une expression booléenne) reste vraie. On appelle donc cette structure une **boucle conditionnelle**. Comme pour le `if`, c'est l'**indentation** des instructions qui détermine si elles sont dans la boucle (exécutées tant que la condition est vraie) ou en dehors (exécutées après la fin de la boucle, quand la condition devient fausse). En Python, la syntaxe générale du `while` est la suivante :

```
while condition :
    instruction_1
    instruction_2
    ...
    instruction_n
# ici la condition est fausse, on sort de la boucle
```

Condition de boucle : si la condition est fausse dès le premier essai, alors on n'entre jamais dans la boucle et on passe directement à la suite du programme. Au contraire, si la condition reste toujours vraie, alors on ne sort jamais de la boucle, le programme ne se termine pas, on est coincé dans une **boucle infinie** : **attention danger !** Pour interrompre un programme qui boucle, il faut taper Ctrl-C au clavier pendant l'exécution.

Utilité : les boucles sont très utiles dans plusieurs cas : pour **filtrer** les entrées de l'utilisateur ; pour **rejouer** un programme ; pour **répéter** plusieurs fois les mêmes instructions.

1.7.1 While pour filtrer

On a vu qu'un programme pouvait interagir avec l'utilisateur en lui demandant d'entrer des valeurs. On a parfois besoin de vérifier que les valeurs entrées par l'utilisateur respectent bien certains critères, sous peine que le programme ne fonctionne pas correctement : c'est ce qu'on appelle **filtrer** les entrées.

Ce type de comportement est observé très souvent dans les programmes que nous utilisons tous les jours. Par exemple, un programme qui nous demande de nous identifier avec un mot de passe (par exemple : Skype), va continuer à afficher la page d'accueil tant qu'on n'a pas rentré les bons identifiants.

Considérons un exemple simple, qui demande à l'utilisateur deux entiers positifs A et B, et affiche le quotient de la division de A par B. Pour avoir un résultat correct, nous devons d'abord vérifier que B est différent de 0. Le programme peut être exprimé de la façon suivante :

```
a = int(input('Donnez la valeur de A : '))
b = int(input('Donnez la valeur de B : '))
if b != 0 :
    print('A / B = ', a // b)  # division entière
else:
    print('Division par 0 impossible')
```

Ce programme fonctionne correctement, mais si l'utilisateur entre une valeur B nulle, il n'a pas de deuxième chance. Comment le modifier afin qu'il continue de demander la valeur de B à l'utilisateur, **jusqu'à ce que** celle-ci ne soit pas égale à 0 ?

On pourrait répéter manuellement le même code un certain nombre de fois. Par exemple, pour donner à l'utilisateur 3 chances d'entrer une valeur correcte, on peut naïvement écrire :

```
a = int(input('Donnez la valeur de A : '))
b = int(input('Donnez la valeur de B : '))
if b == 0 :
    b = int(input('B ne peut etre nul, veuillez reessayer : '))
    if b == 0 :
        b = int(input('B ne peut etre nul, veuillez reessayer : '))
        if b == 0 :
            b = int(input('B ne peut etre nul, veuillez reessayer : '))
if b != 0 :
    print('A / B = ', a // b)
else:
    print('Division par 0 impossible')
```

Cependant, ce code est très répétitif (ce qui n'est pas bon !) et ne permet de répéter cette action qu'un nombre limité de fois (ici 3 fois). Ce n'est pas le résultat souhaité. A l'aide du `while`, on peut simplement ré-exprimer le programme comme suit :

```
a = int(input(‘Donnez la valeur de A : ‘))
b = int(input(‘Donnez la valeur de B : ’))
while b == 0 :
    b = int(input(‘B ne peut etre nul, veuillez reessayer : ’))
# sortie de la boucle quand b est different de 0
# maintenant on peut faire la division sans erreur
print(‘A / B = ‘, a // b)
```

Ce programme redemande la valeur de B *tant que* celle-ci est égale à 0 (c'est-à-dire **jusqu'à** ce qu'elle soit non nulle). Si on sort de la boucle, cela veut dire que b n'est plus égal à 0, et on peut alors effectuer notre division sans re-vérifier sa valeur.

1.7.2 While pour rejouer un programme

La boucle `while` peut aussi être utilisée pour répéter un programme un nombre indéterminé de fois, selon le choix de l'utilisateur. Par exemple voici un programme qui demande à l'utilisateur un caractère et affiche son code ASCII.

```
carac = input("Entre un caractere ?")
code = ord(carac)
print("Le code ASCII de",carac,"est",code)
```

Si on veut maintenant proposer à l'utilisateur de recommencer avec un nouveau caractère, puis un autre, etc, jusqu'à ce qu'il demande d'arrêter, on peut compléter le programme comme ceci :

```
play = ’oui’ # on initialise à ’oui’ pour jouer au moins une fois
while play==’oui’ :
    carac = input("Entre un caractere ?")
    code = ord(carac)
    print("Le code ASCII de",carac,"est",code)
    play = input("veux-tu rejouer ? oui/non ")
# sortie de boucle si l’utilisateur répond autre chose que ’oui’
print("Fin du programme")
```

Ce programme rentre au moins une fois dans la boucle puisqu'on initialise la variable play à la valeur 'oui', donc le premier test de la condition réussit. Ensuite, après chaque exécution, le programme demande à l'utilisateur de dire s'il veut continuer ou pas. Tant que l'utilisateur répond 'oui', le programme recommence en demandant un nouveau caractère. Dès que l'utilisateur répond 'non' (ou toute autre chaîne que 'oui') la boucle se termine, et l'instruction suivante est exécutée (affichage du message de fin).

1.7.3 While pour répéter, compteur de boucle

La boucle *while* peut également être utilisée pour répéter un bout de code un nombre **déterminé** de fois. Pour ce faire, on utilise une variable compteur qui s'incrémente à chaque itération. Par exemple, si on voulait afficher les 10 premières puissances de 2, on pourrait le faire de la manière suivante :

```
i = 0               # notre variable compteur
while i < 10 :      # pour 10 valeurs de i, entre 0 et 9
    print( 2 ** i ) # afficher 2 puissance i
    i = i + 1       # incrémenter compteur à chaque itération
```

Ici, la valeur du compteur i augmente de 1 (on dit que la variable i a été **incrémentée**) après chaque affichage. Le premier affichage est effectué lorsque i est égal à 0, puis tout de suite après i est incrémenté et prend la valeur 1. Puisque i est toujours inférieur à 10, la boucle continue et l'affichage est encore effectué pour $i = 1$, etc. L'affichage continue donc jusqu'à ce que i devienne supérieur ou égal à 10. En d'autres mots, la variable i va compter de 0 à 9.

Attention : si on oublie d'incrémenter le compteur, il ne deviendra jamais supérieur ou égal à 10, et la boucle ne s'arrêtera jamais, c'est une boucle infinie (mauvais !).
Il est également possible de compter avec un pas différent de 1. Par exemple, le programme suivant avance par pas de 2 afin d'afficher tous les entiers positifs **pairs** inférieurs à 100.

```
i = 0
while i < 100 :
    print(i)
    i = i + 2
```

Le pas peut également être négatif : on parcourt en ordre décroissant. Dans ce cas on initialise le compteur à la valeur supérieure, et on précise la borne inférieure comme condition de boucle. Par exemple, pour afficher les 10 premiers entiers strictement positifs dans l'ordre décroissant :

```
i = 10              # borne supérieure
while i > 0 :       # condition sur la borne inférieure
    print(i)
    i = i - 1       # décrémenter le compteur
```

1.7.4 Accumulateurs et drapeaux

Lorsqu'on parcourt un ensemble de valeurs, il arrive souvent qu'on ait besoin de garder en mémoire certaines informations sur les valeurs parcourues. Par exemple, on peut avoir besoin de calculer la somme ou le produit de ces valeurs. Ou encore, on pourrait avoir besoin de garder en mémoire la plus grande ou la plus petite valeur que nous ayons rencontrée dans une liste de valeurs. Ceci introduit la notion de variable **accumulateur**.

Accumulateurs

Prenons l'exemple du calcul d'une somme d'entiers. On veut écrire un programme qui calcule et affiche la somme des 10 premiers entiers strictement positifs (1 à 10). Comme dans l'exemple précédent, nous avons besoin d'une variable compteur (qu'on appellera i), qui va compter de 1 à 10. De plus, nous aurons besoin d'une variable accumulateur (qu'on appellera 'somme'), qui va accumuler progressivement la somme de toutes les valeurs de i que l'on rencontre. Généralement, on initialise l'accumulateur à l'élément neutre de l'opération que l'on veut effectuer. Dans notre exemple, l'accumulateur a été initialisé à 0 car c'est l'élément neutre de l'addition (0+x=0).

```
i = 1
somme = 0               #  initialement, la somme est égale à 0
while i <= 10 :
    somme = somme + i   # chaque valeur de i est rajoutée à la somme (accumulation)
    i = i + 1           # ne jamais oublier de mettre à jour le compteur
# affichage une seule fois, après la fin de la boucle
print('La somme des 10 premiers entiers est : ', somme)
```

Drapeaux

Parfois, l'information qu'on souhaite garder sur notre séquence n'est pas une valeur numérique mais une propriété. Par exemple, est-ce que tous les nombres parcourus sont impairs ? Dans ce cas, on peut utiliser une variable booléenne qui est égale à True si la propriété est vérifiée, et à False dans le cas où cette propriété est fausse. Cette variable, un accumulateur booléen, est aussi appelée "**drapeau**" (ou flag).

Exemple : on veut lire 10 entiers et vérifier qu'ils sont tous impairs. Pour que cette propriété soit vraie, il faut que tous les nombres lus soient impairs. Autrement dit, il faut que : `premier nombre est impair ET deuxieme nombre est impair ET ...` Il s'agit donc d'une accumulation utilisant l'opérateur ET (and).

Dans cet exemple, la drapeau *tous_impairs* est initialisé à True, car c'est l'élément neutre de l'opération "and" : si au moins un élément est pair, le drapeau sera égal à False, sinon il restera à True. Si le drapeau était une disjonction, c'est-à-dire qu'on utilise un opérateur OU (`or`) entre les éléments accumulés, alors l'élément neutre serait False : en effet il suffit que l'une des valeurs soit vraie pour que la disjonction soit vraie.

```
i=0
tous_impairs = True
while i < 10:
    x = int(input('Entrez un entier:'))
    tous_impairs = tous_impairs and x % 2 != 0
    i = i + 1
if tous_impairs:
    print('Tous les nombres entrés sont impairs')
else:
    print('Au moins un nombre entré n'était pas impair')
```

Lorsqu'on utilise l'opérateur 'and', il suffit que l'une des valeurs ne soit pas impaire pour que notre drapeau soit faux. Une manière équivalente et plus intuitive d'écrire ce type de programme est donc de mettre le booléen *tous_ impairs* a False dès qu'on tombe sur une valeur paire.

```
i=0
tous_impairs = True
while i < 10:
    x = int(input('Entrez un entier:'))
    if x % 2 == 0:
        tous_impairs = False
    i = i + 1
if tous_impairs:
    print('Tous les nombres entrés sont impairs')
else:
    print('Au moins un nombre entré n'était pas impair')
```

1.7.5 Boucle infinie, Break, Continue

Boucle infinie

Une boucle peut s'exécuter indéfiniment lorsque la condition du `while` est toujours vérifiée (i.e. ne devient jamais fausse). Cela peut être intentionnel, par exemple en utilisant comme condition le booléen True. Le programme suivant continue à afficher tout ce que l'utilisateur rentre au clavier, et ne s'arrête jamais.

```
while True:
    a = input()
    print(a)
```

Mais il arrive également qu'on rentre dans une boucle infinie par erreur, parce qu'on oublie de mettre à jour les compteurs de boucle. Dans l'exemple suivant on oublie d'incrémenter la valeur de i, ce qui fait que la condition $i < 10$ reste tout le temps vraie. Le programme va donc afficher 1 (2 à la puissance 0) à l'infini, jusqu'à être interrompu par l'utilisateur avec Ctrl-C.

```
i = 0
while i < 10:
    print(2 ** i)
```

Altération du fonctionnement de la boucle - Inconvénients

Les instructions `break` et `continue` permettent d'altérer le comportement normal de la boucle `while`. Cependant, leur utilisation rend le code plus difficile à lire et analyser, en particulier s'il contient plusieurs niveaux d'imbrications et/ou de longues instructions dans le `while`. De plus ces instructions n'ont pas toujours d'équivalent dans les autres langages de programmation. On essaiera donc autant que possible d'éviter d'utiliser `break` et `continue`.

Instruction Break

L'instruction *break* permet de sortir d'une boucle immédiatement, indépendamment de la condition du 'while'. Cependant le plus souvent son utilisation n'est pas nécessaire. Par exemple la boucle suivante se terminera dès que i est égal à 2, et donc n'affichera que les valeurs 0 et 1. Mais il aurait mieux valu changer la condition du `while` en $i<2$.

```
# avec break
i = 0
while i < 10:
    if i == 2:
        break
    print(i)
    i = i + 1
```

```
# version correcte sans break
i=0
while i<2:
    print(i)
    i+=1
```

Une autre utilisation typique de 'break' consiste à éviter les expressions booléennes compliquées. Par exemple, supposons qu'on veuille lire des entiers au clavier jusqu'à ce qu'on tombe sur un multiple de 2, 3, 5, 7, 11, ou 13. On peut écrire une longue condition booléenne, ou bien la séparer en plusieurs lignes en utilisant 'break'

```
# version 1: longue condition booléenne
a = int(input("Tape un entier"))
while a % 2 != 0 and a % 3 != 0 and a % 5 != 0 and a % 7 != 0 and a % 11 != 0 and a % 13 != 0:
    a = int(input("Retape un entier"))

# version 2: avec break, on teste chaque sous-condition à tour de rôle
a = int(input())
while True:
    if a % 2 == 0:
        break
    if a % 3 == 0:
        break
    if a % 5 == 0:
        break
    if a % 7 == 0:
        break
    if a % 11 == 0:
        break
    if a % 13 == 0:
        break
    a = int(input())
```

Dans certains cas, *break* peut aussi être utilisé pour éviter la répétition de code. Par exemple, supposons que l'on veuille écrire un programme qui lit trois nombres au clavier, et qui continue à afficher la moyenne des trois nombres jusqu'à ce que cette moyenne soit inférieure à 10.

```
# version 1: répétition de code
a = int(input())
b = int(input())
c = int(input())
moyenne = (a + b + c) / 3
while moyenne >= 10.:
    print(moyenne)
    a = int(input())
    b = int(input())
    c = int(input())
    moyenne = (a + b + c) / 3
```

```
# version 2: break évite la répétition
while True:
    a = int(input())
    b = int(input())
    c = int(input())
    moyenne = (a + b + c) / 3
    # dès que moyenne<10 on sort de la boucle
    if moyenne < 10:
        break
    print(moyenne)
```

Comment remplacer break A noter que l'on essaiera de se passer de l'utilisation de `break` dans la mesure du possible car cela peut rendre le programme difficile à lire et à comprendre. Une autre astuce qui permet d'éviter le cas précédent sans utiliser `break` est d'initialiser la condition de la boucle de sorte à être sûr d'y rentrer une première fois. On préférera cette solution à celle utilisant `break`.

```
moyenne = 100  # On initialise à une moyenne >= 10 pour être sûr
               # qu'on rentre dans la boucle une première fois
while moyenne >= 10:
    a = int(input())
    b = int(input())
    c = int(input())
    moyenne = (a + b + c) / 3
    if moyenne >= 10:
        print(moyenne)
```

On peut utiliser le caractère \ pour séparer une expression booléenne trop longue en plusieurs lignes :

```
a = int(input())
while a % 2 != 0 and \
        a % 3 != 0 and \
        a % 5 != 0 and \
        a % 7 != 0 and \
        a % 11 != 0 and \
        a % 13 != 0:
    a = int(input())
```

Enfin pour les conditions d'arrêt de boucle, on préférera les intégrer directement dans la condition de la boucle.

```
# version avec break (à éviter)
while cond:
    instructions
    if condstop :
        break
```

```
# réécriture sans break (à préférer)
stop=False
# on teste les 2 conditions
while (not condstop) and cond:
    instructions
```

Instruction Continue

Comme `break`, l'instruction `continue` permet d'altérer le comportement normal de la boucle. `continue` permet de passer directement à l'itération suivante de la boucle en ignorant toutes les instructions qui restent dans l'itération courante. Par exemple, le programme suivant affiche la somme des 10 premiers nombres entrés au clavier qui ne sont pas multiples de 2 ni de 3. Dans la première version avec `continue`, lorsque la condition du *if* est vérifiée, on repasse directement au début de la boucle pour lire un autre nombre, sans changer la somme, et sans incrémenter i. Dans la 2e version, on teste si la condition est fausse avant de comptabiliser n, ce qui donnera le même résultat.

```
# version avec l'instruction continue
i = 0
somme = 0  # initialisation de la somme
while i < 10:
    n = int(input())
    # si n est un multiple de 2 ou de 3
    if n % 2 == 0 or n % 3 == 0:
        # passer à l'itération suivante
        continue
    # donc ici n n'est multiple ni de 2 ni de 3
    somme = somme + n
    i = i + 1
```

```
# réécriture sans continue
i=0
somme=0
while i<10:
    n=int(input())
    # on utilise la négation de la condition
    # si n n'est multiple ni de 2 ni de 3
    if n%2!=0 and n%3!=0:
        # on l'ajoute à la somme
        somme += n
    i += 1
```

1.7.6 Imbrication de boucles

Il peut y avoir une boucle `while` parmi les instructions dans un bloc `while` : on parle alors de boucles *imbriquées*. Prenons comme exemple le programme ci-dessous. Ici, i est le compteur de boucle de la première boucle `while` (boucle externe) et j celui de la deuxième boucle while (boucle interne). (Attention les 2 compteurs doivent être des variables différentes !) La boucle externe ne passe à l'itération suivante (prochaine valeur de i), qu'après que la boucle interne a fini toutes ses itérations. Dans l'exemple, pour chaque valeur de i (de 1 à 3), j comptera de 1 à 2.

```
# exemple de programme
    i = 1
    while i <= 3:
        j = 1
        while j <= 2:
            print(i, ", ", j)
            j = j + 1
        i = i + 1
```

```
# affichage produit
    1, 1
    1, 2
    2, 1
    2, 2
    3, 1
    3, 2
```

Exemple d'application : le programme ci-dessous demande à l'utilisateur un entier N et affiche un carré de caractères '*' de N lignes et N colonnes.

```
N = int(input("Entrez la valeur de N: "))
i = 0                       # compteur de lignes
while i < N:                # N lignes
    j = 0                   # compteur d'étoiles par ligne
    while j < N:            # N étoiles par ligne
        print('*', end="")  # pas de retour chariot, étoiles sur la meme ligne
        j = j + 1           # passage a l'étoile suivante
    # fin de l'itération sur j qui a affiché N étoiles
    print()                 # maintenant, saut de ligne
    i = i + 1               # on passe à la ligne suivante
# fin de l'itération sur i qui a affiché N lignes
```

Exercice d'application Modifier ce programme pour afficher un rectangle de N lignes et M colonnes.

1.8 Les fonctions

On a pour l'instant développé uniquement des programmes *tout-en-un*. Les fonctions permettent de découper le code en plusieurs morceaux réutilisables. Le but des fonctions est de structurer son code lorsque l'on fait plusieurs fois la même chose (ou presque) :

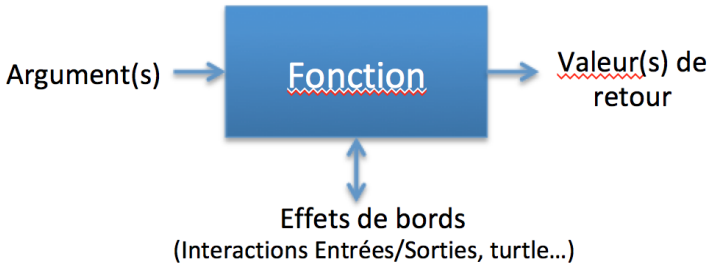
- Pour qu'il soit plus lisible (plusieurs morceaux)
- Pour qu'il soit plus facilement modifiable (pas de duplication de code)
- Pour qu'il soit plus facile à tester (tester chaque morceau séparément)

Un exemple du TP

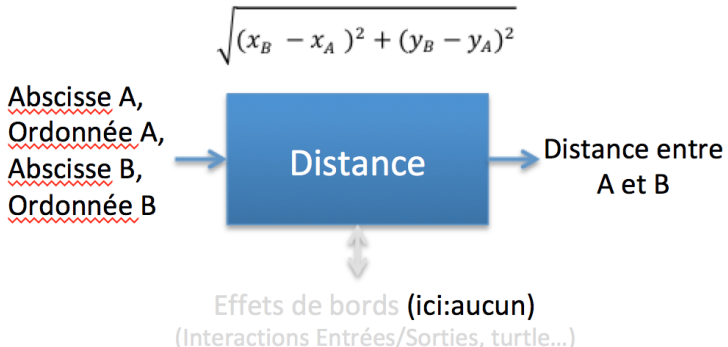
```
import turtle
# fonction qui trace un carre de taille egale a cote
def carre(cote) :
    i = 1               # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(cote)
        turtle.right(90)
        i=i+1
# programme principal
carre(100)              # carre de taille 100
turtle.up()
turtle.forward(130)
turtle.down()
carre(50)               # carre de taille 50
```

1.8.1 Principe d'une fonction

Une fonction est une suite d'instructions, encapsulées dans une «boîte», identifiée par un nom ; elle reçoit zéro, un ou des arguments / paramètres ; elle renvoie zéro, une ou plusieurs valeurs de retour ; et elle crée éventuellement des effets de bord modifiant l'environnement (interactions entrées/sorties, turtle, etc). On peut appeler cette fonction dans un programme pour réutiliser ces instructions plusieurs fois, sur des paramètres différents.



Exemple : fonction géométrique de calcul de distance entre 2 points



Voici le code de cette fonction, qui est à définir au-dessus de votre programme principal. Cette fonction reçoit 4 arguments : `absA, ordA, absB, ordB`. Elle renvoie 1 valeur de retour de type `float`. Elle n'a pas d'effets de bord (en particulier elle n'affiche rien).

```
def distance(absA, ordA, absB, ordB) :
    d=(absB-absA)**2 + (ordB-ordA)**2
    d=d**(1/2)
    return d
```

On peut alors écrire le programme principal suivant, qui appelle cette fonction en lui passant 4 valeurs pour ses 4 paramètres (dans le même ordre) :

```
# prog. principal
print(distance(1, 2, 1, 5))
xA=2
yA=3
z=distance(xA, yA, 0, 0)
print("Distance de (0,0) à A :", z)
```

On remarque que les valeurs passées en paramètres peuvent être soit des constantes (les entiers 1, 2, 1, 5 dans le premier appel), soit des variables (les variables xA,yA dans le deuxième appel). On remarque aussi que les variables n'ont absolument pas besoin d'avoir le même nom que celui utilisé dans la définition de la fonction. C'est l'ordre qui compte : la première valeur (constante ou variable) est affectée au premier argument, la 2e valeur au 2e argument, etc. Il faut donc passer exactement autant de valeurs lors de l'appel que le nombre de paramètres défini pour cette fonction.

Cette fonction renvoie une valeur, quand on l'appelle il faut donc faire quelque chose de cette valeur : soit on l'affiche directement (comme dans le premier appel), mais alors on ne pourra plus réutiliser cette valeur ; soit on la stocke dans une variable (ici dans la variable z pour le 2e appel). Attention, la fonction renvoie une **valeur**, qui est stockée dans la variable d dans le corps de la fonction, mais cette variable d est **locale** à la fonction, elle n'existe pas dans le programme principal. Il faut donc **affecter** la valeur de retour dans une nouvelle variable, définie dans le programme principal ou dans la fonction appelante.

On peut aussi appeler cette fonction directement depuis l'interpréteur.

```
>>> distance(0, 1, 3, 5)
5.0
>>> distance(1,2,4,7)
5.830951894845301
```

1.8.2 Définition d'une fonction

La syntaxe pour définir une nouvelle fonction est la suivante :

```
def nom_fonction(argument1, ..., argumentN) :
    instructions à exécuter
    return valeur de retour
```

Note : le `return` est facultatif (dans ce cas la fonction ne renvoie rien), ainsi que les arguments / paramètres (une fonction peut ne recevoir aucun paramètre), mais pas les parenthèses (qui sont alors vides).

Définition d'une fonction sans paramètres

Voici un exemple de fonction sans paramètre (on note qu'il y a des parenthèses vides après le nom de la fonction, ces parenthèses sont **obligatoires**). Cette fonction n'a aucune valeur de retour (pas de `return`), par contre elle a un effet de bord, l'affichage d'un message à l'écran.

```
def bonjour() :
    print("bonjour")
```

Ci-dessous, la fonction n'a pas de paramètre, elle a un effet de bord (affichage du message 'Quel est ton nom ?' et attente d'une saisie par l'utilisateur), et elle renvoie une valeur.

```
def demander_nom():
    nom=input("Quel est ton nom? ")
    return nom
```

La fonction suivante n'a ni paramètre ni valeur de retour, mais a des effets de bord (dessine un carré dans la fenêtre `turtle`).

```
import turtle
def carre_standard():
    i = 1 # compteur du nombre de cotes
    while i <= 4 :
        turtle.forward(100)
        turtle.right(90)
        i=i+1
```

1.8.3 Appel d'une fonction

Syntaxe Une fois définie, on peut appeler une fonction par son nom :

- Depuis le programme principal
- Depuis une autre fonction
- Directement depuis l'interpréteur

On peut appeler soit des fonctions qu'on a soi-même définies, soit des fonctions déjà fournies par Python ou par ses nombreuses librairies (à condition de les avoir importées avant avec `import`). On a par exemple déjà utilisé des fonctions des modules `turtle` et `random`.
La syntaxe pour appeler une fonction est la suivante :

```
nom_fonction(argument1, argument2, ...)
```

Passage des paramètres Lors de l'appel d'une fonction, les parenthèses doivent contenir exactement autant de valeurs (constantes ou variables) que la fonction a d'arguments. Ces valeurs sont affectées aux paramètres dans le même ordre. Si la fonction n'a aucun paramètre, les parenthèses sont vides.

Si on appelle une fonction en lui passant trop de valeurs ou pas assez (par rapport à son nombre de paramètres), on déclenche une erreur. Par exemple avec la fonction `distance` définie ci-dessus avec 4 paramètres :

```
>>> distance(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: distance() missing 2 required positional arguments: 'absB' and 'ordB'

>>> distance(1,2,3,4,5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: distance() takes 4 positional arguments but 5 were given
```

Appel d'une fonction sans paramètres Pour appeler une fonction sans paramètres, il faut quand même utiliser des parenthèses (mais vides) pour indiquer qu'on veut exécuter cette fonction. Par exemple avec la fonction `bonjour()` définie ci-dessus, directement dans l'interpréteur :

```
>>> bonjour()
bonjour
>>> bonjour
<function bonjour at 0x1048c3048>
```

Valeur de retour Si une fonction renvoie une valeur de retour, il faut soit **utiliser** cette valeur immédiatement, ou l'**enregistrer** dans une variable pour pouvoir s'en resservir plus tard. Si on se contente d'appeler la fonction, la valeur est perdue.

```
# definition d'une fonction addition
def addition(x,y):
    s = x+y
    return s

# appel de la fonction depuis l'interpréteur
>>> addition(3,4)       # calcule la somme mais ne sauve pas le résultat
>>> print(s)            # s est une variable locale à la fonction
                        # elle est inconnue ici !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined

# option 1 : afficher directement le résultat
print(addition(3,4))
7

# option 2 : enregistrer le résultat
a = addition(3,4)
print(a)
7
# on peut alors le réutiliser
b = addition(a,6)
print(b)
13
```

Valeur None Certaines fonctions n'ont aucune valeur de retour, mais uniquement des effets de bord : elles modifient leur environnement (affichage de valeurs, tracé turtle, etc) mais ne renvoient aucune valeur. On appelle parfois de telles fonctions des **procédures**. Dans ce cas il est inutile d'affecter leur résultat dans une variable (puisqu'il n'y a pas de résultat). Si on le fait quand même, la variable recevra la valeur spéciale `None`, qui signifie "aucune valeur" : la variable n'a pas de valeur, mais s'affiche comme "None".

```
>>> z=distance(2, 3, 4, 5)      # affectation de la valeur dans une variable
>>> print(z)                    # affichage de la variable contenant la valeur de retour
2.8284271247461903

>>> demander_nom()              # effets de bord et valeur de retour
Quel est ton nom? Carole
'Carole'
>>> name                        # on n'a pas sauvé le nom, impossible de l'utiliser
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
>>> name = demander_nom()       # version correcte
Quel est ton nom? Carole
'Carole'
>>> print("Bonjour",name)       # on peut utiliser le nom
Bonjour Carole

# procédure: pas de valeur de retour, pas d'affectation
>>> carre_standard()            # pas d'argument ni valeur de retour
>>> carre(50)                   # pas de valeur de retour mais des effets de bord (turtle)
>>> x = carre(50)               # si on affecte quand meme
>>> x                           # x n'a pas de valeur
>>> print(x)                    # il s'affiche comme la valeur spéciale 'None'
None
>>> x+2                         # on ne peut rien en faire
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Appel d'une fonction depuis une autre On peut appeler une fonction dans le programme principal, mais aussi dans le corps d'une autre fonction. Par exemple, en supposant que la fonction `carre` a été définie précédemment :

```
# fonction qui deplace le curseur sans tracer
# fait appel aux fonctions turtle: up, forward, down
def deplace(distance):
    up()
    forward(distance)
    down()

# fonction qui trace une ligne de carres
# fait appel aux fonctions definies plus haut: carre et deplace
def ligne_carres(nb_carres, cote):
    i=0                     # compte les carres déjà tracés
    while i<nb_carres:
        carre(cote)         # appel a la fonction carre
        deplace(cote+10)    # appel a la fonction deplace
        i=i+1               # au suivant !
```

Remarque : on peut même appeler une fonction depuis elle-même, c'est alors une fonction **récursive** (hors programme de cette UE).

1.8.4 Différence entre valeur de retour et effets de bord

On a vu qu'une fonction peut renvoyer une ou plusieurs valeurs de retour, ou aucune ; et qu'elle peut avoir des effets de bord, ou pas. Il faut bien distinguer ces deux concepts.

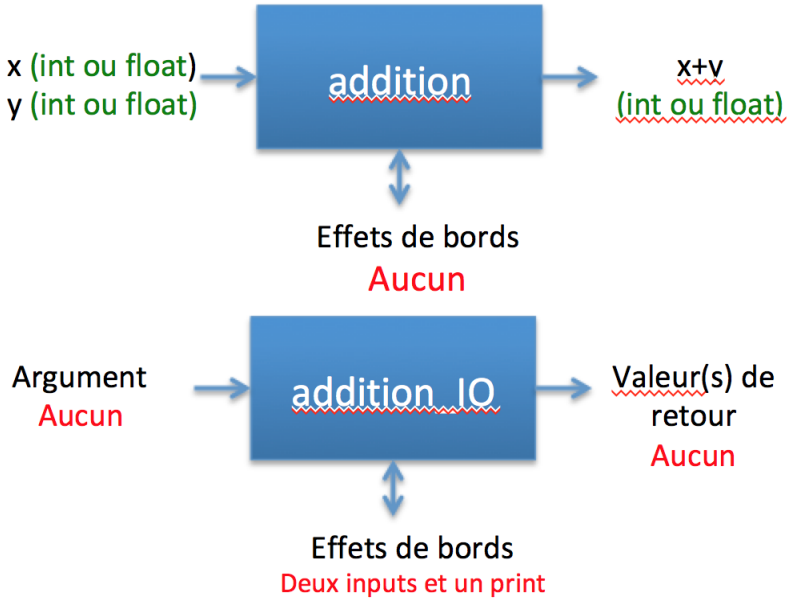
- Les **valeurs de retour** sont des valeurs renvoyées par la fonction, avec le mot-clé `return`. Ces valeurs peuvent (doivent) être utilisées dans la fonction ou le programme qui appelle cette fonction : on peut les afficher, les affecter dans des variables, etc.
- Les **effets de bord** sont des actions réalisées par la fonction qui modifient son environnement, comme d'afficher des éléments à l'écran (avec des instructions comme `print` ou avec les fonctions du module `turtle` par exemple). Les éléments affichés à l'écran ne sont pas utilisables dans la fonction ou le programme appelant ! Seul l'utilisateur peut les visualiser à l'écran quand il exécute le programme.

Regardons la différence sur un exemple. On peut écrire plusieurs fonctions d'addition, qui reçoivent les entiers à additionner en paramètre ou via une interaction avec l'utilisateur (effet de bord), et qui affichent le résultat (effet de bord) ou le renvoient (valeur de retour).

```
# fonction d'addition de 2 nombres, qui RENVOIE la somme
def addition(x,y):
    return x+y

# fonction d'addition de 2 nombres, qui AFFICHE la somme
def addition_aff(x,y):
    print(x+y)
```

```
# fonction d'addition, qui demande 2 entiers, et affiche leur somme
def addition_IO():
    x=float(input("x ?"))
    y=float(input("y ?"))
    print(x+y)
```

Les schémas ci-dessous illustrent les entrées, sorties, et effets de bord de ces fonctions d'addition.



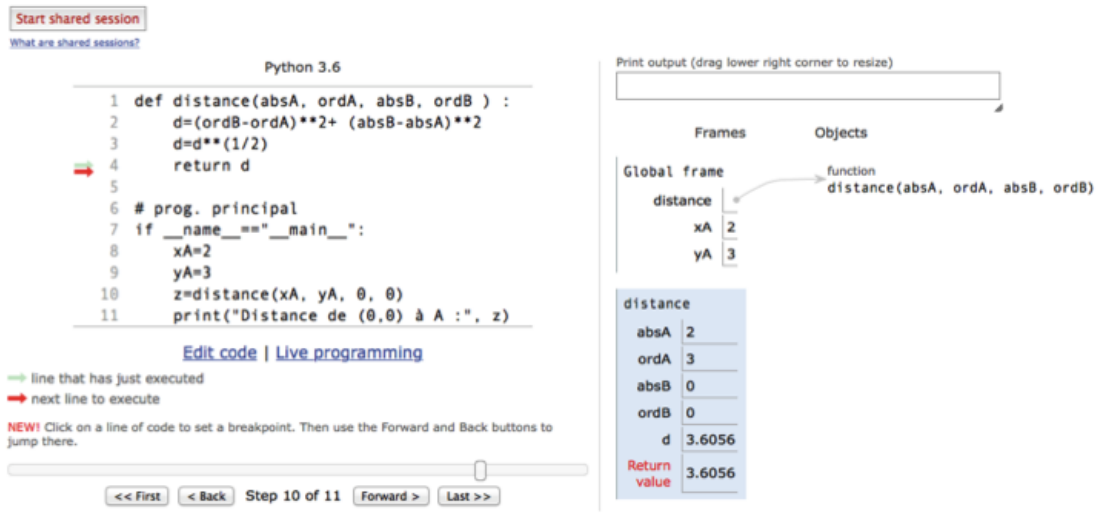
L'appel de ces fonctions depuis un programme principal est donc très différent.

```
# programme principal

# appel de la fonction d'addition qui renvoie la valeur
somme = addition(3,7)           # il faut affecter la valeur dans une variable
print("la somme vaut",somme)    # on peut alors afficher cette variable
# on peut aussi afficher directement la valeur de retour sans l'affecter
# mais alors on ne pourra pas la réutiliser (non stockée)
print("la somme de",5,"et",10,"vaut",somme(5,10))

# appel de la fonction qui affiche
# avec cette fonction on ne peut pas stocker la valeur pour la réutiliser
addition_aff(13,25)             # pas d'affectation ! la fonction affichera 38
a = 12
b = 25
addition_aff(a,b)               # la fonction affichera 37
addition_aff(10,a)              # la fonction affichera 22

# appel de la fonction qui demande et affiche
addition_IO()                   # pas d'affectation, pas de parametres
                                # la fonction interagira avec l'utilisateur
```

1.8.5 Portée des variables

Chaque fonction a son propre "lot" de variables auquel elle a le droit d'accéder. Une variable créée ou modifiée dans le corps d'une fonction, ou qui contient un argument de la fonction, est dite **locale**, et ne sera pas accessible depuis le programme principal, ni depuis une autre fonction.

L'utilisation de Python Tutor permet de visualiser les variables définies dans les différents environnements (`frames`) : l'environnement global (*global frame*) correspond au programme principal, et chaque fonction a son propre environnement. Ainsi sur la figure ci-dessous on voit l'environnement `distance frame` contenant les variables locales à la fonction `distance`.



Exemple de variable locale

```
def moyenne(x,y):
    # calcule la moyenne de x et y, et l'affecte dans une variable locale
    resultat=(x+y)/2
    # renvoie la VALEUR de cette variable locale
    return resultat

# programme principal
a=5
b=6
m=moyenne(a,b)  # affectation de la valeur de retour dans une variable
print(m)        # affiche 5.5
print(resultat) # provoque une erreur
```

L'exécution de ce programme provoque une erreur : *NameError : name 'resultat' is not defined*, ce qui signifie que la variable `resultat` n'est pas définie dans le programme principal, et donc l'instruction qui essaye d'afficher sa valeur échoue. En effet il s'agit d'une variable locale à la fonction `moyenne`, qui n'existe pas dans le programme principal. Par contre sa **valeur** (renvoyée par la fonction `moyenne`) a bien été sauvée dans la variable m lors de l'appel de la fonction.

Variables du programme principal Une variable (de type `int`, `float`, `bool` ou `str`) définie dans le programme principal ne peut pas être modifiée par une instruction qui se trouve à l'intérieur d'une fonction. Cela ne provoque pas d'erreur mais cela crée une nouvelle variable locale portant le même identificateur (nom) que l'autre variable. Par exemple :

```
    # fonction qui affiche sa variable locale a
    def affiche_a():
        a = 1
        print(a)

    # programme principal qui définit une variable a
    a = 2
    # et appelle la fonction ci-dessus
    affiche_a()             # la fonction affiche la valeur de son a local, soit 1
    # affichage de la variable a du programme principal
    print(a)                # le programme principal affiche la valeur de son a local, soit 2
```

Un autre exemple :

```
    def moyenne(x,y):
        # calcule la moyenne de x et y
        resultat=(x+y)/2
        # crée une nouvelle variable test, locale a la fonction
        test=resultat
        # renvoie la valeur de la variable resultat
        return resultat

    # programme principal
    a=5
    b=6
    # variable test, locale au programme principal
    test=0
    m=moyenne(a,b)
    print("m =", m)         # affiche "m=5.5"
    print("test =",test)    # affiche "test=0"
```

Ce programme principal initialise à 0 une variable locale `test`. Ensuite, l'appel à la fonction `moyenne` crée une nouvelle variable, aussi nommée `test`, mais locale à cette fonction. Cet appel ne modifie donc **pas** la valeur de la variable `test` du programme principal. L'instruction `print` affiche donc cette valeur non modifiée, c-à-d 0.

Variables globales Une variable peut être définie comme globale (mot-clé `global`) pour pouvoir être modifiée partout. Cependant, il faut éviter cette pratique. Cela complique le code inutilement et risque d'entraîner des modifications indésirables ailleurs dans le programme.

Attention ! Les variables définies dans le programme principal sont en fait accessibles en lecture seule (on peut les lire mais pas les modifier) depuis l'intérieur d'une fonction, mais ce comportement est dangereux car très subtil. On ne l'utilisera donc pas (sauf éventuellement pour des variables "constantes", initialisées une fois au début du programme et jamais modifiées ensuite). On préférera passer en arguments toutes les valeurs nécessaires.

Par exemple, voici 2 manières de définir une fonction qui décale une chaîne de n tirets. Dans la première version (incorrecte) on utilise la variable n définie dans le programme principal, que la fonction peut lire. Cela fonctionne, mais la fonction est mal définie, si elle a besoin de l'information n alors cela devrait être un argument. Dans la deuxième version (correcte), la valeur de n est reçue en paramètre, et affectée dans la variable n locale à la fonction. C'est la bonne façon de définir cette fonction.

```
# version incorrecte, utilise le n du programme principal en lecture seule
def decalage(s):
    return ("-" * n) + s

# depuis l'interpréteur
>>> n=5
>>> print(decalage("toto"))
-----toto

# version correcte, reçoit la valeur en paramètre et l'affecte dans le n local
def decalage(s, n):
    return ("-" * n) + s

# depuis l'interpréteur
>>> print(decalage("toto", 5))
-----toto
>>> print(decalage("maison",10))
----------maison
```

Variables modifiables Nous avons vu que les variables de types simples définies dans le programme principal ne sont pas modifiables dans les fonctions appelées. Il y a une exception avec les types complexes que nous verrons plus tard. Par exemple le type liste (Chapitre 1.10) permet de stocker une liste de plusieurs valeurs. Ces types complexes fonctionnent un peu différemment des types simples, et en particulier ils peuvent être modifiés par une fonction (ce sera un nouveau type d'effet de bord).

Résumé

- Une variable créée ou modifiée dans une fonction est **locale**, elle n'existe **que** dans la fonction.
- Une variable simple (de type int, float, str ou bool) du programme principal peut être lue mais ne peut pas être modifiée à l'intérieur d'une fonction. On verra plus tard que c'est différent pour les types complexes comme les listes.
- On passera en argument des fonctions toutes les valeurs nécessaires à leur fonctionnement.
- On évitera les variables globales.
- On affectera la valeur de retour d'une fonction dans une variable pour pouvoir la réutiliser ; les procédures ne renvoient rien et on les appelle donc sans affecter le résultat dans une variable (valeur `None`).

1.8.6 Importation de modules

Un grand nombre de fonctions sont déjà définies en Python, et sont rangées dans des modules sur un thème spécifique. Pour utiliser ces fonctions, il faut d'abord importer le module correspondant. Les syntaxes possibles pour importer un module sont les suivantes :

```
import nom_module   # il faut ensuite préfixer les fonctions par le nom du module
from nom_module import nom_fonction # importer une fonction specifique,
                                    # on n'aura pas besoin de prefixer son nom
from nom_module import * # importer toutes les fonctions du module,
                         # on n'aura pas besoin de prefixer leurs noms

import matplotlib.pyplot as plt    # renommer le module pour simplifier l'appel
```

Le mot clé `as` permet de créer un 'raccourci' pour appeler les fonctions d'un module. C'est pratique quand le nom du module est assez long, puisqu'il faut préfixer l'appel de chaque fonction par le nom du module. Par exemple pour `matplotlib.pyplot` ci-dessus, on pourra maintenant appeler les fonctions en les préfixant uniquement par `plt` au lieu du nom complet du module.

Modules utiles Quelques modules intéressants :

- `math` : regroupe les fonctions mathématiques les plus courantes
- `random` : génération de nombres pseudo-aléatoires
- `turtle` : librairie graphique pour l'apprentissage de la programmation (`https://docs.python.org/3/library/turtle.html`)
- `os` : manipulation de fichiers, dossiers, chemins d'accès, permissions...
- `Tkinter` : interface graphique par défaut

1.9 Chaînes de caractères

1.9.1 Type string

Les chaînes de caractères (`string` en Python) sont un type de données, plus complexe que ceux vus jusqu'à présent. Il s'agit d'un type *itérable*, c'est-à-dire dont on peut parcourir les valeurs (ici les différents caractères qui composent la chaîne). On verra d'autres types itérables plus tard (listes, dictionnaires).

Syntaxe On a déjà utilisé les chaînes de caractères, notamment dans les fonctions `print()` et `input()`. En Python, il existe 3 syntaxes pour les chaînes de caractères :

- Avec des guillemets doubles, ce qui permet d'utiliser des guillemets simples (apostrophes) dans le texte :

  ```
  print("je m'appelle toto")
  ```

- Avec des apostrophes, ce qui permet d'utiliser des guillemets doubles dans le texte :

  ```
  print('il a dit "bonjour" en arrivant')
  ```

- Avec des guillemets triples, ce qui permet de créer de longues chaînes contenant des guillemets, des apostrophes, des sauts de ligne... :

  ```
  print("""il m'a dit "je m'appelle toto" puis est parti""")
  ```

Exemples

```
>>> print("C'est toto")
C'est toto
>>> print('C'est toto')
SyntaxError : invalid syntax
>>> print("Il a dit "hello" !")
SyntaxError : invalid syntax
>>> print('Il a dit "hello" !')
Il a dit "hello" !
>>> print("""C'est toto qui a dit "hello" !""")
C'est toto qui a dit "hello" !
>>> print("""C'est toto qui a dit "hello"""")
SyntaxError : ...
```

1.9.2 Caractères

Table ASCII Une chaîne est formée de caractères, qui sont représentés chacun par un code ASCII unique (cf tableau ci-dessous). Ainsi les lettres minuscules ont des codes ASCII entre 97 ('a') et 122 ('z'), les lettres majuscules ont des codes ASCII entre 65 ('A') et 90 ('Z'). Le tableau contient aussi des caractères numériques ('0', '1', etc) et divers caractères spéciaux.

Fonctions ord() et chr() Ce tableau est à la fin du chapitre, mais il n'est pas nécessaire de connaître ces codes ASCII ! Les opérateurs `ord()` et `chr()` permettent de les retrouver :

```
>>> ord('a')    # trouver le code ASCII d'un caractere
97
>>> ord('@')
64
>>> chr(99)     # trouver le caractere correspondant a un code ASCII
'c'
>>> chr(123)
'{'
```

Caractères d'échappement Le caractère \ permet d'utiliser des caractères spéciaux dans une chaîne de caractères :

- $\backslash'$ est une apostrophe mais ne ferme pas la chaîne de caractères (même si entre apostrophes)
- $\backslash"$ est une guillemet double, mais il ne ferme pas la chaîne de caractères
- $\backslash n$: retour à la ligne
- $\backslash t$: tabulation
- $\backslash\backslash$: si on veut insérer un caractère \ ('*backslash*')

Exemples

```
>>> print("il a dit \"bonjour\" en arrivant")
il a dit "bonjour" en arrivant
```

1.9.3 Opérateurs sur les chaînes

Concaténation L'opérateur + permet de concaténer plusieurs chaînes de caractères, c'est-à-dire de les coller l'une après l'autre pour former une seule chaîne.

```
>>> nom = input("Ton nom ? ")
Ton nom ? toto
>>> salutation = "bonjour "+nom
>>> print(salutation)
bonjour toto

>>> "abc" + "def"
'abcdef'
```

Répétition L'opérateur * permet de concaténer plusieurs fois la même chaîne.

```
>>> x = int(input("Combien ? "))
Combien ? 7
>>> print(x*'*')    # affiche x symboles etoile
*******
>>> print('+'*x)    # affiche x symboles +
+++++++

>>> "ta " * 4
'ta ta ta ta'
```

Comparaison Les opérateurs de comparaison <,>,<=,>=,==,!= peuvent s'utiliser entre des chaînes de caractères. Il s'agit alors d'une comparaison dans l'ordre de la table ASCII (voir le tableau plus loin) : la comparaison respecte l'ordre alphabétique, les majuscules sont avant les minuscules, les chaînes plus courtes sont avant les chaînes plus longues qui ont le même début.

```
>>> 'abc' < 'a'
False
>>> 'abc' < 'abca'
True
>>> 'abc' < 'z'
True

>>> 'a' < 'A'               # les majuscules sont avant dans la table ASCII
False
>>> 'A' == 'a'              # respect de la casse
False
>>> 'A' < 'a'
True
>>> 'toto' < 'toto aaa'
True

>>> x = 'R'             # affectation
>>> 'a' <= x <= 'z'     # x est-elle une lettre minuscule de l'alphabet?
False
>>> 'A' <= x <= 'Z'     # x est-elle une lettre majuscule?
True
```

1.9.4 Fonctions de manipulation de chaînes

On retrouvera ces mêmes fonctions sur les listes.

Longueur La fonction len() renvoie la longueur d'une chaîne de caractères.

```
>>> s = "abcde"
>>> len(s)
5
>>> len('toto')
4
```

Test d'appartenance Le mot-clé `in` permet de vérifier si une chaîne (ou un caractère) est inclue dans une autre. Par exemple :

```
>>> "a" in "toto"
False
>>> "o" in "toto"
True
>>> "to" in "toto"
True
>>> "abc" in "toto"
False
>>> "abc" in "abcd"
True
>>> "acb" in "abcd"
False
```

Compter On peut aussi vouloir compter le nombre d'occurrences (d'apparitions) d'une chaîne dans une autre, avec la fonction `count()`. Par exemple :

```
>>> "toto".count("o")
2
>>> "toto".count("to")
2
>>> "toto".count("ot")
1
>>> "toto".count("a")
0
```

Test de casse On appelle *casse* d'une chaîne de caractère le fait de savoir si elle est en majuscules (haut de casse) ou en minuscules (bas de casse). Python fournit plusieurs fonctions pour tester ou modifier la casse d'une chaîne.

```
>>> 'toto'.isupper()    # est en majuscules ?
False
>>> 'Toto'.isupper()
False
>>> 'TOTO'.isupper()
True
>>> 'toto'.islower()    # est en minuscules ?
True
>>> 'toTo'.islower()
False
>>> 'ToTo azerty'.lower()           # passer en minuscules
'toto azerty'
>>> 'ToTo azerty'.upper()           # passer en majuscules
'TOTO AZERTY'
>>> 'bonjour a tous'.capitalize()   # une majuscule au début
'Bonjour a tous'
```

Découpage et recollage Il existe différentes fonctions pour découper des chaînes de caractères. La fonction`list()` fournit la liste de tous ses caractères (espaces compris). La fonction `split()` renvoie une liste de chaînes résultant du découpage autour d'un séparateur optionnel fourni en paramètre (ou espace par défaut) ; le séparateur n'apparaît plus dans les éléments. A partir d'une liste de chaînes, `join()` fait l'opération inverse et les recolle ensemble, séparées par une chaîne. Les exemples ci-dessous illustrent la syntaxe de ces fonctions.

```
>>> s = "bonjour a tous"
>>> list(s)                 # liste des caractères de s
['b', 'o', 'n', 'j', 'o', 'u', 'r', ' ', 'a', ' ', 't', 'o', 'u', 's']
>>> s.split()               # découpage de s autour du caractère espace (par défaut)
['bonjour','a','tous']
>>> s.split('o')            # découpage autour de la lettre 'o'
['b', 'nj', 'ur a t', 'us']
>>> ls=['toto','titi','abc','azerty'] # liste de chaînes de caractères
>>> '-'.join(ls)
'toto-titi-abc-azerty'
>>> ''.join(ls)             # recollage autour d'une chaîne vide = concaténation
'totititiabcazerty'
```

1.9.5 Parcours de chaînes (nécessite le cours sur les boucles)

Les chaînes de caractères sont des structures itérables (comme les listes ou les dictionnaires que nous verrons plus tard). Avec une chaîne s, on peut accéder à l'élément à une position i donnée (i est appelé l'**indice**) avec la notation `s[i]`

On peut aussi parcourir tous les éléments d'une chaîne, par itération sur les indices avec une boucle `while` (cf chapitre 1.7), ou par itération directement sur les éléments avec une boucle `for` (cf chapitre 1.11).

```
>>> s = "bonjour a tous"
>>> s[0]            # le premier caractère de la chaîne
'b'
>>> s[-1]           # le dernier caractère de la chaîne
's'
>>> s[3]            # le caractère à l'indice 3 (le 4e de la chaîne)
'j'

>>> s = 'hello'
>>> i=0                 # initialisation du compteur de boucle
>>> while i<len(s):     # répétition jusqu'à la longueur de la chaîne
>>>     print(s[i])     # afficher le i-ième caractère de s
>>>     i+=1            # penser à incrémenter le compteur
h
e
l
l
o

>>> for e in s:         # itération directement sur les caractères
>>>     print(e)
h
e
l
l
o
```

table ASCII

000	NUL (Null Character)	033	!	065	A	097	a
001	SOH (Start of Header)	034	“	066	B	098	b
002	STX (Start of Text)	035	#	067	C	099	c
003	ETX (End of Text)	036	$	068	D	100	d
004	EOT (End of Transmission)	037	%	069	E	101	e
005	ENQ (Enquiry)	038	&	070	F	102	f
006	ACK (Acknowledgement)	039	'	071	G	103	g
007	BEL (Bell)	040	(	072	H	104	h
008	BS (Backspace)	041	)	073	I	105	i
009	HT (Horizontal Tab)	042	*	074	J	106	j
010	LF (Line Feed)	043	+	075	K	107	k
011	VT (Vertical Tab)	044	,	076	L	108	l
012	FF (Form Feed)	045	-	077	M	109	m
013	CR (Carriage Return)	046	.	078	N	110	n
014	SO (Shift Out)	047	/	079	O	111	o
015	SI (Shift In)	048	0	080	P	112	p
016	DLE (Data Link Escape)	049	1	081	Q	113	q
017	DC1 (XON) (Device Control 1)	050	2	082	R	114	r
018	DC2 (Device Control 2)	051	3	083	S	115	s
019	DC3 (XOFF) (Device Control 3)	052	4	084	T	116	t
020	DC4 (device control 4)	053	5	085	U	117	u
021	NAK (Negative Acknowledgement)	054	6	086	V	118	v
022	SYN (Synchronous Idle)	055	7	087	W	119	w
023	ETB (End of Transmission Block)	056	8	088	X	120	x
024	CAN (Cancel)	057	9	089	Y	121	y
025	EM (End of Medium)	058	:	090	Z	122	z
026	SUB (Substitute)	059	;	091	[	123	{
027	ESC (Escape)	060	<	092	\	124	|
028	FS (File Separator)	061	=	093	]	125	}
029	GS (Group Separator)	062	>	094	^	126	~
030	RS (Request to Send)	063	?	095	_	127	DEL
031	US (Unit Separator)	064	@	096	`		
032	SP (Space)						

1.10 Listes

1.10.1 Type liste en Python

Types simples vs types complexes. On a manipulé jusqu'à maintenant plusieurs types de données : les entiers (int), les réels (float), les booléens (bool), les chaînes de caractères (str). Il s'agit de types simples, c'est-à-dire ne contenant qu'une seule valeur. Mais on a parfois besoin de manipuler des structures de données plus complexes, comme des listes, des ensembles, des tableaux à plusieurs dimensions (matrices), etc.

```
>>> a = 6
>>> type(a)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type(True)
<class 'bool‘>
>>> type('bonjour')
<class 'str'>
```

Listes. En Python, une liste est un ensemble **ordonné** d'éléments. Elle est nommée par un identificateur (comme n'importe quelle variable). Une liste peut grandir (ou se réduire) dynamiquement, c'est-à-dire qu'on peut y ajouter ou en enlever des éléments : sa taille n'est pas fixe. Une liste peut contenir des éléments de types différents, y compris d'autres listes. Les éléments sont notés entre crochets, séparés par des virgules.

```
# initialisation de diverses listes
weekend=["Samedi","Dimanche"]                  # liste de chaînes de caractères
multiple3 = [3, 6, 9, 12]                      # liste d'entiers
romain = [[1,'I'],[2,'II'], [3,'III'],[4,'IV']] # liste de listes
iv = 4
fourreTout = ["Un", 2, 3.0, iv]                # liste contenant plusieurs types d'éléments
vide = []                                      # liste vide
```

Opérateurs utiles Certains opérateurs arithmétiques fonctionnent sur les listes : + correspond à une concaténation, * à une répétition. Cela permet d'initialiser rapidement des listes répétitives.

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l1+l2
[1,2,3,4,5,6]
>>> l = ['C'] + 10*['I']         # utile pour le TP 'propagation de nouvelle'
>>> l
['C', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I', 'I']
>>> len(l)
11
>>> 3 * [2]
[2,2,2]
```

1.10.2 Numérotation des éléments et accès

Une liste est un ensemble **ordonné**. Cela signifie que les éléments sont dans un ordre donné, et qu'on peut accéder à un élément à partir de son numéro, qu'on appelle son **indice**. Les éléments sont indexés (numérotés) à partir de 0, c-à-d que le premier élément a pour indice (position, numéro) 0. En conséquence, pour une liste de n éléments, le dernier élément est à l'indice n-1. Il n'y a pas d'élément à l'indice n.

Accès à un élément Pour accéder à l'élément d'indice i d'une liste `maliste`, on écrit `maliste[i]`. Attention, i doit être strictement inférieur à la taille de la liste, sinon on obtient une erreur. Chaque élément est une variable qui peut donc être lue et modifiée.

```
>>> weekend=["samedi","dimanche"]
>>> weekend[0]
'samedi'
>>> weekend[1]
'dimanche'
>>> weekend[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

On peut aussi accéder aux éléments en comptant à partir de la fin de la liste, avec un indice négatif. L'indice -1 correspond au dernier élément, l'indice -2 à l'avant-dernier, etc. Là aussi, si l'indice va trop loin, on déclenche une erreur.

```
>>> weekend[-1]
'dimanche'
>>> weekend[-2]
'samedi'
>>> weekend[-3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Longueur d'une liste Pour connaître la longueur d'une liste, on utilise la fonction `len()`. Par exemple avec les listes précédentes :

```
>>> len(weekend)
2
>>> len(romain)
4
```

1.10.3 Recherche dans une liste

Test d'appartenance d'un élément : in Le mot-clé `in` permet de tester si un élément donné est dans une liste donnée. L'expression `elem in l` est évaluée à `True` si la liste l contient l'élément `elem`, et à `False` si elle ne le contient pas.

```
>>> ma_liste=[2,5,8,12,17,25,2,7,2,1]
>>> 2 in ma_liste
True
>>> 3 in ma_liste
False
```

Compter un élément : count Par contre `in` ne nous dit pas combien de fois l'élément apparaît, s'il est contenu plusieurs fois dans la liste. Pour cela on dispose de la fonction `count`.

```
>>> ma_liste.count(2)
3
>>> ma_liste.count(31)
0
```

Position d'un élément : index Si on veut savoir non seulement si un élément est dans une liste, mais où il se situe dans cette liste (son indice) alors on utiliser la fonction `index`. Les indices commencent à 0, le premier élément est en position 0, le 2e en position 1, etc. Si un élément n'est pas dans la liste, la fonction déclenche une erreur. Si un élément est plusieurs fois dans la liste, la fonction renvoie l'indice de sa première occurrence (première apparition).

```
>>> ma_liste.index(5)
1
>>> ma_liste.index(2)
0
>>> ma_liste.index(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 9 is not in list
```

On peut définir une fonction `indice` qui vérifie qu'un élément est dans la liste **avant** de chercher sa position. Par convention, cette fonction renvoie la position -1 pour signifier que l'élément n'est pas trouvé dans la liste.

```
def indice(liste,i):
    if i in liste:
        return liste.index(i)
    else:
        return -1

# dans l'interpréteur
>>> indice(ma_liste,2)
0
>>> indice(ma_liste,9)
-1
```

1.10.4 Affichage

Affichage standard avec print La fonction `print` déjà utilisée pour afficher des variables de types simples est aussi capable d'afficher des listes. Par contre le paramètre optionnel `sep` ne sert à rien : il s'agit d'un séparateur entre les paramètres de `print`, or ici il n'y en a qu'un, la liste.

```
>>> l1=[1,2,3]                  # crée une liste
>>> print(l1)                   # l'affiche sous forme standard
[1,2,3]
>>> print(l1,sep='-')           # le séparateur ne sert à rien, un seul argument
[1, 2, 3]
>>> print(l1,"toto",sep='-')    # séparateur entre la liste et la chaîne
[1, 2, 3]-toto
```

Affichage avec boucle while Si on veut personnaliser l'affichage, il faut parcourir les éléments un par un avec une boucle.

```
# procédure d'affichage d'une liste reçue en paramètre
def afficheET(liste):
    i = 0
    while i<len(liste):
        print(liste[i],"et",end=" ")    # affichage personnalisé sans retour chariot
        i += 1                          # élément suivant
    print()                             # retour à la ligne final
```

```
# appel dans l'interpréteur
>>> prenoms = ['toto','titi','yoyo','mumu']
>>> afficheET(prenoms)      # pour eviter le et final il faudrait traiter le dernier elmt à part
toto et titi et yoyo et mumu et
```

Affichage avec boucle for Les boucles `for` (chapitre 1.11) permettent d'itérer directement sur les éléments d'une liste plutôt que sur leur indice, et donc de simplifier l'écriture. Par exemple pour afficher un élément par ligne plutôt que l'affichage standard entre crochets.

```
def afficheFOR(liste):
    for elem in liste:
        print(elem)         # retour à la ligne automatique à chaque élément

# appel depuis l'interpréteur
>>> l = ['a','b','c']
>>> afficheFOR(l)
a
b
c
```

1.10.5 Modifier et ajouter des éléments

Modification d'une valeur Pour modifier la valeur d'un élément, il suffit de lui affecter une nouvelle valeur. Attention encore à utiliser un indice existant. On ne peut pas affecter de valeur à un élément qui n'est pas encore dans la liste, là aussi on déclenche une erreur (index out of range = dépassement des limites de la liste). (Ce n'est donc pas comme cela qu'il faut procéder pour ajouter un nouvel élément dans la liste ; on dispose pour cela d'opérateurs dédiés.)

```
>>> weekend[0] = 'saturday'
>>> weekend
['saturday', 'dimanche']
>>> weekend[1] = 'sunday'
>>> weekend
['saturday', 'sunday']
>>> weekend[2] = 'lundi'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Ajout d'un seul élément en fin de liste : append Pour ajouter un seul élément e à la fin d'une liste l, on utilise la fonction `l.append(e)`. Cette fonction modifie directement la liste (effet de bord), elle ne renvoie rien. Attention donc à ne surtout pas affecter son résultat dans la liste elle-même, sinon on l'écrase !

```
>>> Multiple3 = [3, 6, 9]
>>> Multiple3.append(12)
>>> Multiple3
[3, 6, 9, 12]
>>> Multiple3.append(21)
>>> Multiple3
[3, 6, 9, 12, 21]
>>> x = Multiple3.append(7)
>>> Multiple3               # 7 a bien été ajouté en fin de liste
[3,6,9,12,7]
>>> x                       # mais x n'a reçu aucune valeur
>>> print(x)
None
>>> Multiple3 = Multiple3.append(18)    # à ne SURTOUT PAS faire
>>> Multiple3                           # on a perdu notre liste
>>> print(Multiple3)
None
```

Insertion d'un élément à une position voulue : insert Pour **insérer** un seul élément, non pas en fin de liste mais à une position donnée, on utilise `liste.insert(index,element)`. Si l'indice donné est trop grand (dépasse la taille de la liste), alors l'élément est inséré en fin de liste (pas d'erreur). On peut aussi donner une position négative, qui est alors comptée à partir de la fin de la liste. De même, si cet indice négatif est trop petit (avant le début de la liste), l'élément est inséré en tout début de liste. Cette fonction modifie directement la liste (effet de bord) mais ne renvoie rien.
Remarque : l'insertion d'un élément est plus coûteuse en calcul que `append` car elle implique de décaler les autres éléments en mémoire.

```
>> multiple3 = [3, 6, 9, 21]
>> multiple3.insert(3,15)       # insérer en position 3 l'élément 15
>> multiple3
[3, 6, 9, 15, 21]
>> multiple3[3]                 # on vérifie
15
>>> multiple3
[3, 6, 9, 15, 21]
>>> multiple3.insert(7,24)      # 7 est trop grand: insertion à la fin
>>> multiple3
[3, 6, 9, 15, 21, 24]
```

```
>>> multiple3.insert(-7,12)     # -7 est trop petit: insertion au début
>>> multiple3
[12, 3, 6, 9, 15, 21, 24]
```

Ajout de plusieurs éléments : extend Pour ajouter plusieurs éléments d'un seul coup à la fin d'une liste, on peut utiliser `liste.extend(liste2)` qui reçoit une deuxième liste en paramètre, et ajoute tous ses éléments à la fin de la liste. Attention, ce n'est pas l'élément `liste2` (un élément de type liste) qu'on ajoute comme un élément de `liste`, mais bien les éléments qu'elle contient (plusieurs éléments de type entier). Attention, `extend` reçoit un seul paramètre, qui doit être une liste (même si elle contient un seul élément).

```
# avec extend, les éléments de la 2e liste sont ajoutés dans la 1e
>>> multiple3 = [3,6,9,15,21]
>>> multiple3.extend([24,27])
>>> multiple3
[3, 6, 9, 15, 21, 24, 27]
# avec append, c'est la liste qui est ajoutée en fin de liste
>>> multiple3 = [3,6,9,15,21]
>>> multiple3.append([24,27])
>>> multiple3
[3, 6, 9, 15, 21, [24,27]]
# extend reçoit un seul paramètre de type liste
>>> multiple3.extend(24,27)         # erreur, 2 entiers au lieu d'1 liste
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: extend() takes exactly one argument (2 given)
>>> multiple3.extend(24)            # erreur, 24 est un entier et pas une liste
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
# il faut passer une liste, même si c'est un singleton
>>> multiple3.extend([30])
>>> multiple3
[3, 6, 9, 15, 21, 24, 27, 30]
```

1.10.6 Supprimer des éléments

Supprimer un élément à une position donnée : pop La fonction `liste.pop(index)` retire l'élément présent à la position index et le renvoie. La fonction pop modifie directement la liste (effet de bord), et renvoie l'élément supprimé (valeur de retour) ; on peut donc l'affecter dans une variable pour le récupérer. Si l'indice donné est hors des limites possibles, l'appel déclenche une erreur. Si on ne fournit pas d'indice (paramètre optionnel), la fonction retire et renvoie le dernier élément de la liste.

```
>>> multiple3 = [3, 6, 9, 15, 21, 24, 27, 24, 24]
>>> a = multiple3.pop(0)
>>> a
3
>>> multiple3
[6, 9, 15, 21, 24, 27, 24, 24]
>>> b = multiple3.pop(27)   # erreur, indice 27 trop grand
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
>>> b = multiple3.pop()
>>> b
24
>>> multiple3
[6, 9, 15, 21, 24, 27, 24]
```

Supprimer un élément de valeur donnée : remove La fonction `liste.remove(element)` retire l'élément de valeur donnée. S'il est présent plusieurs fois dans la liste, uniquement la première valeur trouvée est supprimée. S'il n'est pas présent, l'appel déclenche une erreur. La fonction remove modifie directement la liste (effet de bord) et ne renvoie rien.

```
>>> multiple3
[6, 9, 15, 21, 24, 27, 24, 24]
>>> multiple3.remove(24)   # retirer le premier 24 trouvé
>>> multiple3
[6, 9, 15, 21, 27, 24, 24]
>>> multiple3.remove(47)   # erreur, tentative de supprimer un élément non présent
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

1.10.7 Copie et clone de listes

Renommage de liste Attention : l'opérateur = (affectation) permet juste de donner 2 noms à la même liste. Avec une simple affectation, on donne un deuxième identificateur à la même liste, mais on n'en crée pas de nouvelle. Les deux identificateurs réfèrent toujours à la même liste, et donc les modifications apportées à une des variables après l'affectation s'appliquent également à l'autre variable.

```
liste1=[1,2,3]              # création de liste1
liste2=liste1               # liste2 est un autre nom pour liste1
liste2.append("bip")        # si on modifie liste2
>>> liste2
[1, 2, 3, 'bip']
>>> liste1                  # on modifie aussi liste1
[1, 2, 3, 'bip']
```

Copie/clonage de surface : list() La fonction `list()` opère une **copie de surface** : elle crée une nouvelle liste contenant les mêmes éléments que la liste initiale. Il s'agira bien de 2 listes distinctes. Les modifications apportées à une des listes après la copie **n'affecteront pas** l'autre liste.

```
liste1=[1,2,3]              # création de liste1
liste3=list(liste1)         # copie de liste1 dans liste3
liste3.append("bip")        # modification de liste3
>>> liste3
[1, 2, 3, 'bip']
>>> liste1                  # liste1 n'a pas été modifiée
[1, 2, 3]
```

Visualisation On peut utiliser Python Tutor pour bien visualiser ce qui se passe. Sur la première capture d'écran ci-dessous, on a utilisé une simple affectation `liste2=liste1`, les 2 variables pointent donc vers la même liste : on lui a donné 2 noms. Sur la deuxième capture d'écran on a utilisé `list` pour faire une copie (de surface) de liste1 dans liste3. Les 2 variables correspondent bien à 2 listes différentes.

Αυτό είναι για μία απλή λίστα και όχι για πίστα με άλλες λίστες στο εσωτερικό της.

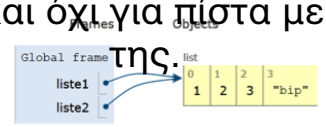


FIGURE 1.1 – Affectation (renommage) : liste2 = liste1

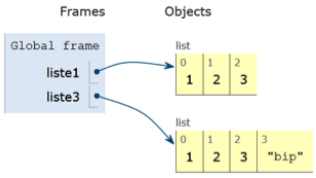


FIGURE 1.2 – Copie de surface : liste3 = list(liste1)

Le module copy Le module `copy` contient plusieurs fonctions génériques permettant de copier des variables de différents types. Pour les utiliser il faut d'abord importer ce module avec l'instruction `import copy`. La fonction `copy.copy()` permet de réaliser une copie *de surface*. La fonction `copy.deepcopy()` permet de réaliser une copie *profonde*, ou récursive.

```
>>> import copy
>>> liste = [1,2,3]
>>> copie = copy.copy(liste)
>>> copie.append(4)
>>> copie
[1, 2, 3, 4]
>>> liste
[1, 2, 3]
```

Copie de surface vs copie profonde Si une liste contient d'autres listes, pour la copier correctement il faut utiliser la fonction `deepcopy` du module `copy`, qui va faire une copie *profonde*, récursive, c'est-à-dire qu'elle va aussi faire des copies des listes contenues dans la liste copiée. Pour une liste simple, une copie de surface suffit.

```
>>> liste1
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> surface = copy.copy(liste1)
>>> profonde = copy.deepcopy(liste1)
>>> surface                          # copie de surface, la liste à l'indice 3 n'est pas clonée
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> profonde
[1, 2, 3, ['a', 'b', 'c'], 4]
>>> surface[3].append('d')           # si on modifie la liste surface[3]
>>> surface
[1, 2, 3, ['a', 'b', 'c', 'd'], 4]
```

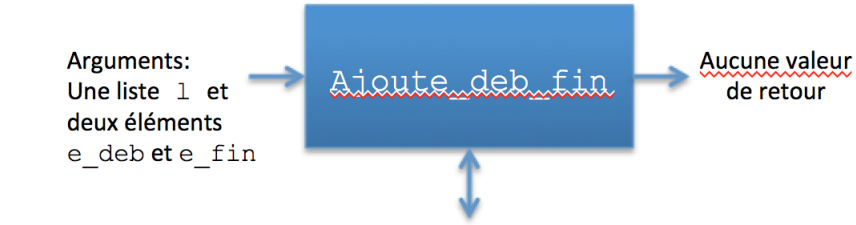
```
>>> liste1
[1, 2, 3, ['a', 'b', 'c', 'd'], 4]
>>> profonde[3].append('e')
>>> profonde
[1, 2, 3, ['a', 'b', 'c', 'e'], 4]
>>> liste1
[1, 2, 3, ['a', 'b', 'c', 'd'], 4]
```

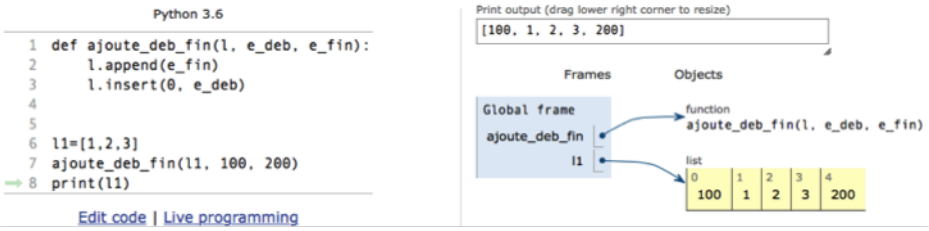
```
# alors on la modifie aussi dans liste1 (c'est la même)

# copie profonde/récursive: la liste a été clonée
# si on modifie profonde[3]

# on ne modifie pas liste[3]
```

1.10.8 Effets de bord avec les listes

Une fonction peut modifier une liste passée en argument, indépendamment de sa valeur de retour. C'est une nouvelle forme d'effet de bord. Les modifications apportées à la liste dans la fonction sont conservées après la sortie de la fonction, donc dans le programme appelant. **Attention** aux effets de bord non désirés ! Il faut donc éviter qu'une fonction modifie une liste si ce n'est pas prévu, ou alors bien penser à copier la liste initiale avant de la passer en argument si on ne souhaite pas qu'elle soit modifiée par la fonction.

Par exemple imaginons une fonction qui insère un élément en début et fin de liste.





Exemple avec oubli de la copie

```
import random

def ajoute_random(l):
    """ Renvoie une liste obtenue a partir de l en ajoutant un entier aleatoire entre 5 et 10"""
    x=random.randint(5,10)
    l.append(x)
    return l
```

Cette fonction modifie la liste reçue en paramètre (effet de bord) **et** renvoie la liste modifiée (valeur de retour). En fait il est inutile de renvoyer la liste, puisqu'on l'a modifiée directement. Ainsi

```
>>> l = [1,2,3]
>>> l2 = ajoute_random(l)
>>> l2
[1, 2, 3, 8]
>>> l
[1, 2, 3, 8]
>>> l.append(5)
>>> l
[1, 2, 3, 8, 5]
>>> l2
[1, 2, 3, 8, 5]
```

On remarque qu'en fait on a ici donné un deuxième identificateur (l2) pour nommer la même liste (l). Et surtout, on a modifié notre liste initiale l alors qu'on ne le souhaitait pas. La version correcte de cette fonction est donc la suivante :

```
import random
def ajoute_random(l):
    """ Renvoie une liste obtenue a partir de l en ajoutant un entier aleatoire entre 5 et 10"""
    x=random.randint(5,10)
    ma_liste=list(l)    # faire une copie de l
    ma_liste.append(x)  # modifier la copie et pas l'original
    return ma_liste     # renvoyer la nouvelle liste, differente de l'original
```

Cette nouvelle version de la fonction crée et renvoie une nouvelle liste. La liste reçue en argument n'est pas modifiée. Il n'y a donc pas d'effets de bord. Attention il faut bien utiliser une fonction de copie et pas une simple affectation (`copie = l`), qui ne crée pas de copie et donc conduit à modifier quand même la liste initiale.

1.11 Boucle inconditionnelle : for

1.11.1 Boucles conditionnelles vs inconditionnelles

Dans une boucle `while`, c'est la condition qui détermine le nombre de fois que la boucle sera exécutée. Il s'agit d'une boucle **conditionnelle**. La boucle continue **tant que** la condition est vraie, c-à-d **jusqu'à** ce que la condition soit fausse. Le plus souvent on ne sait pas combien de fois la boucle va se répéter.

Si on connaît à l'avance le nombre de répétition, on peut utiliser le `for` qui est une boucle **inconditionnelle**. Au lieu de spécifier une condition de répétition, on spécifie directement le nombre d'itérations ; la boucle se répète exactement ce nombre de fois. On peut aussi parcourir des structures itérables (comme les listes) : la boucle se termine alors à la fin du parcours.

1.11.2 Boucle for pour répéter n fois, fonction range()

Fonction range(). La fonction `range(deb, fin, pas)` reçoit 3 arguments de type entier, et génère une séquence d'entiers compris entre `deb` inclus et `fin` exclus, avec le `pas` choisi. Les paramètres `deb` et `pas` sont optionnels. La borne inférieure `deb` vaut 0 par défaut, et le `pas` vaut 1 par défaut. On peut choisir un pas négatif pour une séquence décroissante d'entiers ; dans ce cas fin doit être <= deb, sinon la séquence est vide.

- `range(a)` : séquence des entiers dans [0, a[, c'est-à-dire dans [0, a-1] (borne sup exclue) avec un pas de 1.
- `range(b,c)` : séquence des valeurs [b, c[, c'est-à-dire dans [b, c-1] (b inclue, c exclue) avec un pas de 1
- `range(e, f, g)` : séquence des valeurs [e, f[avec un pas de g

Remarque : comme les arguments ne sont pas nommés, c'est la position des valeurs qui détermine à quel argument elles sont affectées (cf chapitre 1.13). On ne peut donc pas omettre la valeur de début si on veut préciser le pas.

Utilisation avec for Quand on veut écrire une boucle inconditionnelle avec n répétitions, le plus simple est donc d'utiliser `range(n)` pour générer une séquence de n entiers.

```
for var in range(n) :
    instructions
```

Plus généralement, si on veut parcourir les entiers entre deb et fin, avec un pas donné, on peut spécifier les 3 arguments de range.

```
for var in range(deb, fin, pas) :
    instructions
```

Syntaxe. Comme pour le `if` et le `while`, c'est l'**indentation** des instructions qui détermine si elles sont **dans** le bloc `for` (et donc répétées), ou **après** la fin du bloc (et donc exécutées seulement quand on sort de la boucle).

Bornes incohérentes En cas d'incohérence dans les bornes du `range`, la séquence est vide, la boucle est donc ignorée, et l'on passe directement aux instructions suivantes. Par exemple :

```
# de 200 à 210 avec un pas négatif, séquence vide, n'affiche rien
for k in range(200, 210, -2) :
    print(k)

# de 110 à 100 avec un pas positif, séquence vide, n'affiche rien
for k in range(110,100,2) :
    print(k)
```

Exemples

```
# affiche les entiers de 1 à 5 (6 exclus) séparés par une virgule
for i in range(1,6) :
    print (i,end=",")       # affichage de chaque entier suivi d'une virgule
print()                     # retour à la ligne a la fin de la sequence

# affiche les entiers pairs de 0 à 100 (un par ligne)
for i in range(0,101,2) :
    print(i)

# affiche les entiers en ordre décroissant de 10 (inclus) à 0 (exclus)
for i in range(10,0,-1) :
    print(i)
```

Boucle while vs for Quand on connaît à l'avance le nombre d'itérations souhaitées, la boucle `for` est beaucoup plus concise. Par exemple, si on veut afficher les entiers de 0 à n.

```
# avec un while
i=0             # initialisation compteur
while i<=n :    # condition
    print(i)
    i = i+1     # incrementation compteur
# fin de la boucle quand i depasse n

# avec un for
for i in range(n+1):  # range avec n+1 valeurs a partir de 0
    print(i)
# fin du for apres exactement n+1 iterations
```

1.11.3 Parcours de structures itérables

La boucle `for` permet aussi de parcourir des structures : listes, chaînes de caractères, etc. Dans ce cas au lieu d'un compteur de boucle, on utilise une variable qui prend successivement pour valeurs tous les **éléments** contenus dans la structure parcourue (tous les éléments d'une liste, toutes les lettres d'un mot, etc).

```
# parcourt les entiers de la liste, calcule la somme
s = 0
for e in [1, 4, 5, 0, 9, 1] :
    s+=e
print(s)            # affichage en sortie de boucle
# parcourt les lettres de la liste, affiche 1 par ligne
for e in ["a", "e", "i", "o", "u", "y"]:
    print(e)
# parcourt les lettres du mot, affiche 1 par ligne
for e in "python":
    print(e)
```

Algorithmes sur les listes La boucle `for` permet ainsi d'écrire des programmes intéressants pour manipuler des listes : trier une liste pour mettre ses éléments dans l'ordre ; chercher l'élément minimum ou maximum d'une liste ; compter le nombre d'occurrences d'un éléments donné dans une liste (combien de fois il apparaît dans la liste) ; etc. *(cf exercices de TD et TP)*

1.11.4 Très important : itération et modification

Attention ! Ne jamais modifier la variable de boucle dans le corps d'une boucle for ! Quoi qu'il arrive dans le corps de la boucle, la variable de boucle (celle qui parcourt la séquence ou la structure itérable) prend la valeur suivante (l'entier suivant de la séquence, l'élément suivant de la liste ou autre structure itérable) à chaque nouvelle étape de la boucle.

Exemple (ce qu'il ne faut pas faire) Ce programme affiche les entiers 1,2,3,4 (un entier par ligne), c'est-à-dire qu'il parcourt la séquence générée par range(1,5). La modification de la variable i dans le corps de la boucle est immédiatement écrasée quand i prend la valeur suivante de la séquence avant de recommencer les instructions du corps de la boucle. Vous pouvez exécuter ce programme dans Python Tutor pour mieux visualiser.

```
for i in range(1, 5) :
    print(i)
    i = i*2
```

Exemple corrigé (avec un while) Le programme ci-dessus est mauvais. Il ne faut jamais (jamais !) modifier le compteur de boucle dans un for. Dans cet exemple il vaudrait mieux écrire une boucle while dans laquelle on peut multiplier i par 2 à chaque itération (il faudra initialiser i avant le while, et la condition du while devra spécifier la valeur maximale souhaitée). Cette version fonctionne et affiche les entiers 1,2,4 (un par ligne).

```
i=1
while i<5:
    print(i)
    i=i*2
```

Exemple corrigé (avec un for) Avec une boucle for, il suffit de se rendre compte qu'on parcourt en fait les puissances de 2, et d'écrire la version suivante, qui itère sur l'exposant, et affiche les 3 premières puissances de 2 (en partant de puissance 0), donc 1, 2, 4.

```
for i in range(3):
    print(2**i)
```

Attention : ne jamais modifier la structure pendant qu'elle est parcourue !

Exemple Dans l'exemple ci-dessous, on définit une liste de 5 entiers, puis on écrit une boucle qui affiche chaque élément puis le supprime de la liste.

```
>>> liste = [1,2,3,4,5]
>>> for i in liste :
        print(i)
        liste.remove(i)

# affichage produit
1
3
5

>>> liste
[2, 4]
```

Le comportement obtenu n'est pas ce qu'on attendait : seul un entier sur 2 est affiché et supprimé de la liste. En effet, il ne faut jamais modifier la structure (ici la liste) en même temps qu'on la parcourt.

1.12 Listes avancées

1.12.1 Fonction map

La fonction `map()` permet d'appliquer une fonction à tous les éléments d'une liste, et d'obtenir la liste de tous les résultats. Le résultat est de type `map`, qu'il faut convertir en liste. On peut utiliser en paramètre le nom d'une fonction existante, ou bien une fonction anonyme avec `lambda`. Les fonctions anonymes peuvent aussi servir à combiner plusieurs fonctions. (Voir les exemples ci-dessous.)

```
>>> list(map(len,["alex","cyril","elsa"]))
[4, 5, 4]
>>> list(map(math.sqrt,[1,4,9,16]))
[1.0, 2.0, 3.0, 4.0]
>>> list(map(lambda x:x**2,[1,4,9,16]))
[1, 16, 81, 256]
>>> list(map(lambda x:int(math.sqrt(x)),[1,4,9,16]))
[1, 2, 3, 4]
```

1.12.2 Compréhension de listes

Python offre une syntaxe abrégée pour manipuler les listes : la compréhension de listes. C'est une syntaxe plus concise pour écrire une boucle `for` qui crée une liste. Cela permet par exemple de réécrire la fonction `map`. Quelques exemples :

```
>>> [x**2 for x in range(10)]        # carres des entiers de 0 a 9
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> prenoms = ["alex","cyril","elsa"]
>>> l = [len(x) for x in prenoms]
[4, 5, 4]
>>> ["a" in x for x in prenoms]
[True,False,True]
```

Exercice bonus : essayer d'écrire les fonctions du TD sur les boucles en un minimum d'instructions en utilisant des compréhensions de listes (à éviter en général dans un vrai programme, dans un souci de lisibilité).

1.13 Fonctions : compléments

1.13.1 Le mot-clé None

Procédures Il existe une valeur constante en Python qui s'appelle None. Cela correspond à "rien", "aucune valeur". Lorsqu'une fonction n'a pas d'instruction `return`, elle renvoie la valeur `None`.

```
def dit_bonjour():
    print("Bonjour!")
    print("Bienvenue")
    # pas de return
# programme Principal
test=dit_bonjour()
print("Test vaut", test)
```

```
Affichage obtenu lorsque l'on lance le module:
Bonjour!
Bienvenue.
Test vaut None
```

Initialisation Le mot-clé `None` peut aussi servir à initialiser une variable lorsque l'on ne sait pas encore précisément quelle valeur on souhaite lui attribuer. Attention, `None` n'est pas une chaîne de caractères en Python, donc il ne faut pas de guillemets !

```
reponse=None # on n'a pas encore de reponse
while reponse!="non":
 x=float(input("Veuillez entrer un nombre:"))
 print("Le carré de ce nombre est:", x*x)
 reponse=input("Voulez-vous recommencer?")
print("Terminé")
```

1.13.2 Fonction avec plusieurs valeurs de retour

On a vu qu'une fonction pouvait ne rien renvoyer. Une fonction peut aussi renvoyer plus d'une valeur. Dans ce cas, quand on l'appelle, il faut affecter le retour dans le bon nombre de variables. Par exemple la fonction ci-dessous calcule et renvoie à la fois le quotient et le reste de la division de a par b. Le programme principal affecte donc son retour dans 2 variables.

Syntaxe La syntaxe générale est la suivante :

```
# Syntaxe (dans le corps de la fonction) pour renvoyer N valeurs
return valeur1, valeur2, ... , valeurN

# Syntaxe pour récupérer les N valeurs de retour lors d'un appel
var1, var2, ... , varN = nom_fonction(arguments)
```

Exemple Par exemple cette fonction calcule et renvoie à la fois le quotient et le reste d'une division entière.

```
def division(a,b) :
 # renvoie le quotient et le reste
 # de la division de a par b
 quotient=a//b
 reste= a%b
 return quotient, reste

# programme principal
q,r = division(22,5)
print("q=", q, "et r=", r)
```

1.13.3 Fonction avec des paramètres optionnels

On a déjà vu plusieurs fonctions avec des paramètres optionnels : print (paramètres sep et end) ; range (paramètres deb et pas).

Syntaxe de la définition d'une fonction à paramètres optionnels Pour rendre un argument optionnel lors de la définition d'une fonction, il faut ajouter après le nom de l'argument le signe = suivi de la valeur par défaut. Ci-dessous `arg_opt` est un argument optionnel de la fonction (valeur par défaut précisée), alors que `arg1` n'est pas optionnel (pas de valeur par défaut précisée).

```
# syntaxe standard d'une fonction avec argument optionnel
def nom_fonc(arg1, arg_opt=valeur_par_defaut):
    instructions
```

Exemple La carte de MisterPizza comporte de multiples saveurs de pizzas, chacune pouvant être commandée en taille normale au prix de 9 euros, ou en taille maxi au prix de 12 euros. La très grande majorité des clients choisit des pizzas de taille normale, on veut donc que ce soit la valeur par défaut si rien n'est précisé. Dans ce cas on peut utiliser un argument optionnel `taille` qui a pour valeur par défaut `"normale"`.

```
def affiche_pizza(saveur, taille="normale"):
    """ Affiche saveur, taille et prix de la pizza
    """
    print("Pizza", saveur, "taille:", taille)
    if taille=="normale":
        prix=9
    elif taille=="maxi":
        prix=12
    print("Prix", prix, "euros.")
```

Syntaxe de l'appel d'une fonction à paramètres optionnels Quand on appelle une fonction à paramètres optionnels, on peut préciser une valeur pour le(s) paramètre(s) optionnel(s), ou bien l'omettre et dans ce cas il(s) reçoi(ven)t leur(s) valeur(s) par défaut (spécifiées dans la définition de la fonction). Si on reprend la fonction de l'exemple ci-dessus, voici comment l'appeler, en précisant ou pas la valeur du paramètre optionnel.

```
>>> affiche_pizza("4 fromages")
Pizza 4 fromages taille: normale
Prix 9 euros.
>>> affiche_pizza("4 fromages", "maxi")
Pizza 4 fromages taille: maxi
Prix 12 euros.
>>> affiche_pizza("Reine", "normale")
Pizza Reine taille: normale
Prix 9 euros.
```

Définition d'une fonction avec plusieurs paramètres optionnels MisterPizza souhaite parfois afficher le prix en Francs, lorsque le client est âgé, mais il s'agit d'une situation peu fréquente. On peut donc rajouter un autre paramètre optionnel `afficheF` de type booléen, faux par défaut, qui précise s'il faut afficher le prix en francs ou euros.

```
def affiche_pizza(saveur, taille="normale", afficheF=False):
    """ Affiche saveur, taille et prix de la pizza
    """
    print("Pizza", saveur, ", taille: ", taille)
    if taille=="normale":
        prix=9
    elif taille=="maxi":
        prix=12
    if afficheF:
        prixFrancs=round(prix*6.55957, 2)
        print("Prix", prix, "euros (", prixFrancs, " F).")
    else:
        print("Prix", prix, "euros.")
```

Appel d'une fonction avec plusieurs paramètres optionnels Les valeurs des paramètres sont affectées dans l'ordre où elles sont reçues. Pour appeler la fonction précédente, qui a deux paramètres optionnels, on a donc un problème dans le cas où on veut spécifier le 3e argument (prix en francs) mais pas le 2e (taille normale). En effet la valeur du 2e argument étant omise, c'est la valeur qu'on a donnée pour le 3e (booléen `True`) qui est affectée dans le 2e argument (variable `taille`). Cela déclenche une erreur car la valeur `True` ne correspond à aucune des 2 valeurs possibles (dans le if), donc la variable `prix` ne reçoit pas de valeur, ce qui empêche de l'afficher ensuite.

```
>>> affiche_pizza("4 fromages")
Pizza 4 fromages taille: normale
Prix 9 euros.

>>> affiche_pizza("4 fromages", "maxi")
Pizza 4 fromages taille: maxi
Prix 12 euros.

>>> affiche_pizza("Reine", "maxi", True)
Pizza Reine , taille:  maxi
Prix 12 euros ( 78.71  francs).

>>> affiche_pizza("4 saisons", True)
Erreur car True est pris pour la taille (2eme arg.)
```

Arguments nommés Pour résoudre ce problème, il faut nommer les arguments, afin de spécifier de quel argument on reçoit la valeur. En effet, lors d'un appel de fonction, on peut préciser le nom de l'argument concerné par une valeur, comme ceci :

```
>>> affiche_pizza("4 fromages", afficheF=True)
```

Dans cet appel on a 2 types d'arguments :

- La chaîne de caractères "4 fromages" est un argument non nommé, ou argument **positionnel** : c'est sa position qui détermine à quel argument correspond cette valeur (ici la valeur est en 1e position et sera donc affectée dans le 1e argument, à savoir `saveur`)
- `afficheF=True` est un argument **nommé** : c'est le nom qui détermine à quel argument correspond la valeur qui suit (ici la valeur `True`, en 2e position mais nommée, est donc affectée à l'argument `afficheF`, en 3e position, ce qui permet de ne pas spécifier la valeur du 2e argument, `taille`)

Pour que cela fonctionne, les arguments non-nommés doivent **toujours** être **tous** avant les arguments nommés dans l'appel. L'ordre des arguments nommés ensuite n'est pas déterminant, seule compte leur position (tant qu'ils sont **après** les arguments non-nommés).

Exemples d'appels

```
>>> affiche_pizza("4 fromages", afficheF=True)
Pizza 4 fromages , taille:  normale
Prix 9 euros ( 59.04  francs).

>>> affiche_pizza("4 fromages", taille="maxi", afficheF=True)
Pizza 4 fromages , taille:  maxi
```

```
Prix 12 euros ( 78.71  francs).

>>> affiche_pizza("Chorizo", taille="maxi", True)
Erreur: il y a un argument non-nommé après un argument nommé

>>> affiche_pizza("Reine", afficheF=True, taille="maxi")
Pizza Reine , taille:  maxi
Prix 12 euros ( 78.71  francs).

>>> affiche_pizza("Chorizo", True, taille="maxi")
Erreur car taille est définie deux fois (True est pris pour taille car 2eme argument non-nommé)
```

L'exemple de la fonction print En fait, nous avions déjà rencontré une fonction avec des arguments optionnels nommés : la fonction `print`

```
>>> print("Mon age est", 18)
>>> print("Mon age est", 18, sep="égal à")
>>> print("Mon age est", 18, end=".")
>>> print("Mon age est", 18, sep=":", end=".")
```

`sep` et `end` sont des arguments optionnels de `print`. Par défaut, `sep` vaut " " (espace) et `end` vaut "$\backslash n$" (retour à la ligne). Note : `sep` et `end` doivent toujours être nommés car la fonction `print` a un nombre variable d'arguments non-nommés (les valeurs à afficher), donc on ne peut pas déterminer la position de ces arguments.

1.13.4 Utilisation de Docstring

Une docstring est une chaîne de caractères, encadrée par des triples guillemets, placée au tout début d'une fonction, et qui permet de décrire cette fonction.

Syntaxe générale

```
def nom_fonction(argument1, argument2, ...):
 """ docstring
 """
 instructions de la fonction
```

On peut l'afficher grâce à la commande help, qui prend en paramètre le nom de la fonction (uniquement le nom, pas de parenthèse ni de paramètre) dont on veut obtenir la documentation :

```
help(nom_fonction):
Help on function nom_fonction in module nom_module:
nom_fonction(argument1, argument2, ...)
    docstring de la fonction affichée ici
```

Exemple Si on spécifie la documentation au moment de la définition de la fonction, l'utilisateur de notre code pourra alors dans l'interpréteur (ou dans un programme) appeler l'instruction `help` pour afficher cette documentation.

```
def division(a,b) :
 """ Renvoie le quotient et le reste
  de la division de a par b """
 quotient=a//b
 reste= a%b
 return quotient, reste

>>> help(division)
Help on function division in module __main__:
division(a, b)
    Renvoie le quotient et le reste
    de la division de a par b
```

Docstring sur une fonction existante En cas de doute sur le principe d'une fonction existante, vous pouvez utiliser la commande help pour afficher sa documentation. Par exemple si on ne se rappelle plus si les bornes sont incluses lors de la génération d'un entier pseudo-aléatoire.

```
>>> import random
>>> help(random.randint)
Help on method randint in module random:
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Docstring et Help sur fonction avec paramètres optionnels La documentation affichée pour notre fonction précédente montre bien le paramètre optionnel taille et sa valeur par défaut.

```
>>> help(affiche_pizza)
Help on function affiche_pizza in module __main__:
affiche_pizza(saveur, taille='normale')
    Affiche saveur, taille et prix de la pizza
```

1.14 Dictionnaires

Tout comme une liste, un dictionnaire permet de sauvegarder en mémoire plusieurs valeurs de types quelconques. Cependant, contrairement à une liste, les valeurs d'un dictionnaire ne sont pas stockées de manière ordonnée, mais sont associées à des clés. On accède donc aux éléments par leur clé et pas par leur indice. Chaque clé est **unique**.

1.14.1 Création

Déclaration La syntaxe générale pour déclarer et initialiser un dictionnaire D contenant certaines paires clé-valeur est la suivante :

```
D = { cle1: valeur1, cle2: valeur2, ..., cleN: valeurN }
```

Exemple On veut créer un dictionnaire contenant la note moyenne de chaque étudiant dans une certaine matière. Les clés de ce dictionnaire seront les noms des étudiants, et les valeurs seront leurs notes. Chaque valeur (note) est donc associée à un étudiant (clé). Deux étudiants peuvent avoir la même note, mais chaque étudiant ne peut avoir qu'une seule note (on suppose que chaque prénom est unique dans la classe). Ci-dessous la variable `notes` est un dictionnaire contenant les notes de deux étudiants. Les chaînes de caractères 'nathan' et 'quentin' sont les clés de ce dictionnaire. 12.0 et 15.5 sont les valeurs du dictionnaire. Les éléments du dictionnaire ne sont pas ordonnés : on n'y accède pas par un numéro (indice, position) mais par leur clé.

```
>>> notes = { 'nathan': 12.0, 'quentin': 15.5 }
>>> notes
{'nathan': 12.0, 'quentin': 15.5}
```

Types des clés et valeurs Les clés d'un dictionnaire ne peuvent être que de certains types. Dans ce cours on se limitera aux entiers et aux chaînes de caractères. Par contre comme pour les listes, les valeurs dans un dictionnaire peuvent être de n'importe quel type, y compris de type dictionnaire.

```
pc_tp = {
    'ram': 16,                                      # valeur de type int
    'cpu': 3.5,                                     # valeur de type float
    'portable': False,                              # valeur de type bool
    'os': 'windows',                                # valeur de type str
    'ports': ['usb3.0', 'jack', 'ethernet', 'hdmi'],    # valeur de type list
    'carte_graphique': {                            # valeur de type dict
          'vram': 4,
          'nom': 'gtx970',
          'bus': 256
    }
}
```

Exemple Si on veut stocker les moyennes des étudiants dans plusieurs matières, on peut avoir un dictionnaire de dictionnaires, contenant soit un dictionnaire par étudiant (dont les clés seront les matières), soit un dictionnaire par matière (dont les clés seront les étudiants).

```
# un dico par etudiant
notes_e = {
    'nathan': {'maths':15, 'info': 17},
    'quentin': {'info':13, 'bio':18}
}
# un dico par matière
notes_m = {
    'maths': {'nathan':15},
    'info': {'nathan':17,'quentin':13},
    'bio': {'quentin':18}
}
```

1.14.2 Utilisation d'un dictionnaire

Accès aux valeurs L'accès à une valeur du dictionnaire se fait non pas par sa position (indice), mais grâce à sa clé. Par exemple dans notre premier dictionnaire de notes, on peut utiliser la chaîne de caractères 'quentin' (la clé) pour accéder à la valeur qui y est associée dans le dictionnaire `notes` (la note de Quentin, 15.5). Les dictionnaires sont aussi appelés **listes associatives**, car ils permettent d'associer à chaque clé une valeur de type quelconque.

```
>>> notes = {'nathan': 12.0, 'quentin': 15.5}
>>> notes['quentin']
15.5
```

Erreurs de clé Attention, si on tente d'accéder à une entrée qui n'existe pas dans le dictionnaire, le programme renvoie une erreur de clé (`KeyError`).

```
>>> lettres = {'a': 103, 'b': 8, 'e': 150}
>>> lettres['k']                        # erreur, pas de clé 'k' dans le dico
KeyError: 'k'
>>> lettres['u'] = lettres['u'] + 1     # erreur, pas de clé 'u'
KeyError: 'u'
```

Vérifier l'existence d'une entrée Avant d'accéder à une valeur, on prendra donc l'habitude de toujours vérifier l'existence de la clé, avec l'opérateur `in` comme pour les listes. **Attention !** L'opérateur `in` vérifie l'existence d'une clé, et non pas d'une valeur.

```
# recherche d'une cle : ok
>>> prix = {'asus': 450, 'alienware': 1200, 'lenovo': 680}
>>> 'asus' in prix
True
>>> 'toshiba' in prix
False
# recherche d'une valeur : echec
>>> 1200 in prix
False
```

1.14.3 Modification d'un dictionnaire

Ajout d'une entrée Pour rajouter une nouvelle entrée (une paire clé :valeur) dans un dictionnaire existant, il suffit d'utiliser l'opérateur = (affectation) pour associer la valeur à la clé, comme suit. On peut donc faire une affectation avec une clé qui n'existe pas encore dans le dictionnaire, pour l'y ajouter.

```
>>> D = {}          # crée un dictionnaire vide
>>> D
{}
>>> D['a'] = 1      # ajout de la nouvelle entrée
>>> D
{'a': 1}
```

Modification d'une entrée Pour modifier une entrée déjà présente dans le dictionnaire (c'est-à-dire modifier la valeur d'une clé présente dans le dictionnaire), on procède de la même manière, en utilisant aussi l'affectation =. L'ancienne valeur de cette clé est écrasée, remplacée par la nouvelle valeur.

```
>>> D
{'a': 1}
>>> D['a'] = 3
>>> D
{'a': 3}
>>> D['a']
3
```

Suppression d'une entrée L'opérateur `del` permet de supprimer une association d'un dictionnaire. Comme pour l'accès aux valeurs, il faut que la clé spécifiée existe, sinon on déclenche une erreur de clé.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> del D['a']
>>> D
{'b': 2, 'c': 3}
>>> del D['j']      # erreur, pas de clé 'j'
KeyError: 'j'
```

1.14.4 Parcourir un dictionnaire

Itération avec for. La boucle `for` peut être utilisée pour parcourir toutes les **clés** d'un dictionnaire (qui permettent d'accéder aux valeurs associées).

```
# affichage des associations clé:valeur, une par ligne
for key in D:
    print('La clé', key, 'a pour valeur: ', D[key])

# affichage des dates d'anniversaire
dates_naissance={'ingrid':[12,6,1995],'marc':[27,8,1996],'brice':[11,10,1995]}
# la variable nom parcourt les clés du dictionnaire (les prénoms)
for nom in dates_naissance :
    # date est la valeur associée au nom, la liste représentant la date de naissance
    date = dates_naissance[nom]
    print(nom, 'fetera son anniversaire le',date[0],'/',date[1])
```

cle

Ce programme affiche les lignes suivantes :

```
ingrid fetera son anniversaire le  12 / 6
marc fetera son anniversaire le  27 / 8
brice fetera son anniversaire le  11 / 10
```

Liste des clés et des valeurs La fonction `keys()` permet d'accéder aux clés d'un dictionnaire, dans une structure de type `dict_keys`. De même la fonction `values()` permet d'accéder aux valeurs d'un dictionnaire, dans une structure de type `dict_values`. Ces deux structures ne sont pas indexables, mais on peut les convertir en liste avec la fonction `list()`. Par exemple :

```
>>> dico = {'a':1,'b':0,'c':0,'d':0,'e':1}
>>> dk = dico.keys()
>>> dk
dict_keys(['a', 'b', 'c', 'd', 'e'])
>>> dk[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_keys' object does not support indexing
>>> dico.values()
dict_values([1, 0, 0, 0, 1])

>>> lc = list(dico.keys())
>>> lc
['a', 'b', 'c', 'd', 'e']
>>> lv = list(dico.values())
>>> lv
[1, 0, 0, 0, 1]
>>> lv[1]
0
```

1.14.5 Un exemple détaillé

Dans l'exemple ci-dessus on veut compter chaque lettre dans un mot ou une liste. La fonction reçoit la lettre à compter et le dictionnaire de compteurs, qu'elle met à jour en conséquence (la fonction ne renvoie pas le dictionnaire, mais a pour effet de bord de le modifier). Le programme principal initialise le dictionnaire de compteurs, parcourt la chaîne de caractères, et appelle cette fonction sur chaque lettre lue.

```
# fonction qui reçoit la lettre à compter
# et le dictionnaire de compteurs, qu'elle met à jour
# pas de valeur de retour, modif du dico par effet de bord
def vu_lettre(l,cpts):
    # verification d'existence
    if l in cpts:
        # si existe, incrementation
        cpts[l] = cpts[l] + 1
    else:
        # sinon, initialisation
        cpts[l] = 1

# programme principal
phrase = "bonjour a tous"
# initialiser un dictionnaire vide de compteurs
cpts = {}
for lettre in phrase:
    # appel de la fonction sur chaque lettre vue
    vu_lettre(lettre,cpts)
# affichage du dictionnaire de compteurs
print(cpts)
```

L'exécution de ce programme affiche le dictionnaire de compteurs 'b' :1,'o' :3,'n' :1,'j' :1,'u' :2,'r' :1,' ' :2,'a' :1,'t' :1,'s' :1. Seuls les caractères qui ont été rencontrés ont été ajoutés dans le dictionnaire (y compris l'espace).

1.14.6 Copie de dictionnaires

Affectation Comme dans le cas des listes, l'affectation d'un dictionnaire vers une variable ne fait que référencer le **même** dictionnaire par un nouvel identificateur. Si on modifie l'un, on modifie aussi l'autre.

```
>>> D = {1: 10, 2: 20, 3: 30}
>>> E = D
>>> E[5] = 50
>>> E
{1: 10, 2: 20, 3: 30, 5: 50}
>>> D
{1: 10, 2: 20, 3: 30, 5: 50}
```

Copie Pour créer une copie de surface d'un dictionnaire, on utilise `dict()`.

```
>>> F = dict(D)                         # creation d'une copie de D nommee F
>>> F[6] = 60                           # ajout d'une cle dans F
>>> F                                   # F a ete modifie
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60}
>>> D                                   # mais pas D
{1: 10, 2: 20, 3: 30, 5: 50}
>>> G = F                               # creation d'un 2e nom pour le meme dico
>>> G[7] = 1                            # ajout d'une cle dans G
>>> G                                   # G a été modifié
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60, 7: 1}
>>> F                                   # mais F aussi
{1: 10, 2: 20, 3: 30, 5: 50, 6: 60, 7: 1}
```

1.15 Introduction aux fichiers

Jusqu'à présent, nous avons utilisé `input()` et `print()` pour lire les entrées du programme, et afficher les résultats obtenus. Mais parfois, les données d'entrée sont stockées dans un fichier et on aimerait pouvoir y accéder directement, sans les saisir manuellement au clavier (surtout quand il y en a beaucoup, ou quand on teste plusieurs fois un programme). Souvent, il est aussi utile de sauvegarder les résultats d'un programme dans des fichiers afin de pouvoir y accéder plus tard (par ex : sauvegarde d'une partie dans un jeu vidéo). En Python, il est très facile de lire et d'écrire des données dans des fichiers.

1.15.1 Ouverture d'un fichier texte

Ouvrir un fichier Avant de commencer la lecture d'un fichier, il faut d'abord l'**ouvrir**. Ouvrir un fichier veut simplement dire que l'on crée une variable qui permet de le manipuler. La fonction `open()` est utilisée pour ouvrir un fichier. Par exemple pour ouvrir un fichier appelé «data.txt», on écrit :

```
>>> f=open('data.txt')
>>> f
<_io.TextIOWrapper name='data.txt' mode='r' encoding='UTF-8'>
```

Mode d'ouverture Par défaut, `open()` ouvre un fichier en mode lecture (mode = 'r' pour 'read'), c'est-à-dire qu'on peut lire son contenu mais on ne peut pas le modifier. Il faut donc que le fichier existe. Si on tente d'ouvrir un fichier inexistant en mode «lecture», on déclenche une erreur.

```
>>> f = open('toto')            # le fichier 'toto' n'existe pas
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'toto'
```

1.15.2 Lecture d'un fichier texte

Une fois notre fichier texte ouvert, il existe plusieurs manières de lire son contenu.

Lecture de tout le texte On peut lire en une seule fois tout le contenu du fichier et le stocker dans une chaîne de caractères, qui contiendra le texte entier.

```
texte = f.read()
```

Lecture de toutes les lignes On peut aussi lire en une fois toutes les lignes du fichier et les stocker dans une liste de chaînes de caractères (une chaîne par ligne). Après cette instruction, `lignes` est une liste qui contient toutes les lignes du fichier. On peut y accéder une par une, par exemple `lignes[0]` contient la première ligne, etc.

```
lignes = f.readlines()
```

Lecture itérative par ligne Enfin on peut aussi lire le fichier ligne par ligne de manière itérative, avec une boucle for. Le programme ci-dessous lit une ligne après l'autre dans le fichier et les affiche au fur et à mesure. On ne stocke donc jamais l'intégralité du fichier en mémoire, seulement une ligne à la fois.

```
for ligne in f:
    print(ligne) # affiche une ligne du fichier
```

Exemple Soit le fichier «nombres.txt» qui contient les entiers suivants (un par ligne) : 15 18 30 55 16 3 12 13. Le programme ci-dessous calcule la somme de ces entiers.

```
fichier = open('nombres.txt')          # ouvrir le fichier en lecture
somme = 0                              # initialiser la somme
for nombre in fichier:                 # nombre parcourt les lignes une par une
    # nombre est une chaîne, ne pas oublier de la convertir en entier !
    somme = somme + int(nombre)
# apres la fin de la boucle de calcul, afficher une seule fois la somme
print(somme)
# ne pas oublier de fermer le fichier apres utilisation
fichier.close()
```

1.15.3 Écriture dans un fichier texte

Ouverture en mode écriture Quand un fichier est ouvert en mode lecture, on ne peut que le lire mais pas le modifier. Pour pouvoir écrire dans un fichier, il faut l'ouvrir en mode écriture. Si le fichier n'existe pas encore il est créé (on ne déclenche donc pas d'erreur). On remarque que le mode vaut maintenant 'w' pour 'write', ce qui indique qu'on peut écrire dans ce fichier.

```
>>> f = open('data.txt','w')
>>> f
<_io.TextIOWrapper name='data.txt' mode='w' encoding='UTF-8'>
```

Ouverture en mode ajout Si le fichier ouvert en mode «écriture» n'existe pas, il est **créé** (vide). Mais **attention !** S'il existe déjà, tout son contenu est **effacé**. Pour pouvoir écrire à la suite du texte présent dans un fichier déjà existant, il faut l'ouvrir en mode 'ajout' ('append').

```
>>> f = open('data.txt','a')
>>> f
<_io.TextIOWrapper name='data.txt' mode='a' encoding='UTF-8'>
```

Écriture La fonction permettant d'écrire dans un fichier texte est `write()`. Contrairement à `print()`, la fonction `write()` ne saute pas de ligne automatiquement. Elle écrit dans le fichier exactement la chaîne de caractères reçue en argument. Pour sauter une ligne dans le fichier, il faut écrire un saut de ligne manuellement avec le caractère spécial $\backslash n$ (voir d'autres caractères spéciaux dans le cours sur les chaînes de caractères).

```
f.write('ce texte sera écrit dans le fichier')
f.write('\n') # ceci permet de passer à la ligne
```

Argument de write La fonction `write()` reçoit exactement un argument, qui doit obligatoirement être une chaîne de caractères. Pour écrire un entier ou une valeur d'un autre type, il faut le convertir en chaîne de caractères en utilisant `str()` et la concaténer à la chaîne à écrire.

```
>>> m = 12
>>> f.write("m=",m)         # on ne peut pas passer plusieurs arguments
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() takes exactly one argument (2 given)
>>> f.write(m)              # on ne peut pas écrire un entier
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not int
>>> s = "m="+str(m)         # creation de la chaine a ecrire
>>> f.write(s)              # ecriture : ok
4
```

Valeur de retour de write La fonction `write()` renvoie le nombre de caractères effectivement écrits dans le fichier. Dans l'exemple ci-dessus on a écrit la chaîne "m=12" donc 4 caractères.

1.15.4 Fermeture

Une fois la lecture/écriture terminée, il faut fermer le fichier en utilisant la fonction `close()`. Une fois fermé, on ne peut plus lire ou écrire dans le fichier.

```
f = open('fichier.txt')
# lecture iterative ligne par ligne
for ligne in f:
    print('une ligne lue :', ligne)
f.close()

>>> f.write('tata')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Exemple La fonction suivante permet de sauvegarder les 10 premières puissances de 2 dans un fichier texte nommé «puis.txt» (une par ligne). Après exécution de ce programme, le fichier texte `puis.txt` contiendra donc les entiers suivants (un par ligne) : 1 2 4 8 16 32 64 128 256 512, c"est-à-dire les puissances de 2 entre 2^0 et 2^9.

```
# ouverture en ecriture (creation si inexistant, ecrasement si existe)
fichier = open('puis.txt', 'w')
# 10 iterations pour les 10 puissances
for i in range(0,10):
    # il faut convertir l'entier en chaîne de caractères
    # et passer a la ligne explicitement
    fichier.write(str(2 ** i) + '\n')
# fin de la boucle, on peut fermer le fichier
fichier.close()
```

Chapitre 2

Mémo (fourni aux examens)

Mémo Python - UE INF101 / INF131 / INF204

Opérations sur les types

`type()` : pour connaître le type d'une variable
`int()` : transformation en entier
`float()` : transformation en flottant
`str()` : transformation en chaîne de caractères

Définition d'une fonction

```
def nomFonction(arg) :
    instructions
    return v
```

Fonction qui renvoie la valeur ou variable v.

Infini

`float('inf')` : valeur infinie positive ($+\infty$)
`float('-inf')` : valeur infinie négative ($-\infty$)

Écriture dans la console

`print(a1,a2,...,an, sep=xx, end=yy)`

- Pour imprimer une suite d'arguments de a1 à an
- sep : permet de définir le séparateur affiché entre chaque argument (optionnel, par défaut " ")
- end : permet de définir ce qui sera affiché à la fin (optionnel, par défaut : saut de ligne)

Lecture dans la console

`res = input(message)`

- Pour lire une suite de caractères au clavier terminée par $< Enter >$
- Ne pas oublier de transformer la chaîne en entier (int) ou réel (float) si nécessaire.
- La chaîne de caractères résultante doit être affectée (ici à la variable `res`).
- L'argument est optionnel : c'est un message explicatif destiné à l'utilisateur

Opérateurs booléens

`and` : et logique
`or` : ou logique
`not` : négation

Opérateurs de comparaison

```
== égalité              != différence
<  inférieur,           <= inférieur ou égal
>  supérieur,           >= supérieur ou égal
```

Instructions conditionnelles

```
if condition :
    instructions

if condition :
    instructions
else :
    instructions

if condition1 :
    instructions
elif condition2 :
    instructions
else :
    instructions
```

Opérateurs arithmétiques

```
+ : addition,          - : soustraction
* : multiplication,    ** : puissance,
/ : division,          // : quotient div entière,
% : reste de la division entière (modulo)
```

Caractères

`ord(c)` : renvoie le code ASCII du caractère c
`chr(a)` : renvoie le caractère de code ASCII a

Chaînes de caractères

`len(s)` : renvoie la longueur de la chaîne s
`s1+s2` : concatène les chaînes s1 et s2
`s*n` : construit la répétition de n fois la chaîne s
exemple : "ta"* 3 donne "tatata"
`list(chaine)` : renvoie la liste des caractères de la chaîne
`ch.split(arg)` : retourne la liste des sous-chaînes de ch, en coupant à chaque occurrence de `arg` (par défaut `arg=" "`)
`ch.join(liste)` : concatène les chaînes de `liste`, en utilisant `ch` comme séparateur, et renvoie la chaîne résultante
`ch.upper()`: passe `ch` en majuscules
`ch.lower()`: passe `ch` en minuscules

Itération tant que

```
while condition :
    instructions
```

Itération for, et range

```
for e in conteneur :
     instructions

for var in range (deb, fin, pas) :
     instructions
```

Itère les instructions avec e prenant chaque valeur dans le conteneur (liste, chaîne ou dictionnaire) ; ou avec var prenant les valeurs entre `deb` et `fin` avec un `pas` donné.

`range(a)` : séquence des valeurs [0, a[
`range (b,c)` : séquence des valeurs [b, c[(pas=1, $c > b$)
`range (b, c, g)` : idem avec un pas = g
`range(b,c,-1)`: valeurs décroissantes de b (incl.) à c (excl.), pas=-1 ($c < b$)

Listes

`maListe = []`: création d'une liste vide
`maListe = [e1,e2,e3]` : création d'une liste, ici à 3 éléments e1, e2, et e3

`maListe[i]`: obtenir l'élément à l'index i ($i >= 0$).
Les éléments sont indexés à partir de 0. Si $i < 0$, les éléments sont accédés à partir de la fin de la liste. Ex : `maListe[-1]` permet d'accéder au dernier élément de la liste

`maListe.append(elem)`: ajoute un élément à la fin
`maListe.extend(liste2)`: ajout de tous les élements de la liste `liste2` à la fin de la liste `maListe`
`maListe.insert(i,elem)`: ajout d'un élément à l'index i

`res = maListe.pop(index)`: retire l'élément présent à la position `index` et le renvoie, ici dans la variable `res`
`maListe.remove(element)`: retire l'élément donné (le premier trouvé)

`len(maListe)`: nombre d'éléments d'une liste
`elem in maListe`: teste si un élément est dans une liste (renvoie True ou False)

`l2 = maListe`: crée un synonyme (2ème nom pour la liste)
`l3 = list(maListe)`: crée une copie de surface (un clone)
`l4 = copy.deepcopy(maListe)`: crée une copie profonde (récursive)

Aléatoire

`random.randint(inf,sup)`: entier aléatoire entre bornes inf et sup incluses
`random.shuffle(maListe)`: mélange la liste (effet de bord), ne renvoie rien
`random.choice(maListe)`: renvoie un élément au hasard de la liste

Dictionnaires

`monDico = {}` : création d'un dictionnaire vide
`monDico = { c1:v1, c2:v2, c3:v3 }` : création d'un dictionnaire, ici à 3 entrées (clé c1 avec valeur v1, etc)

`e = monDico[c1]` : les valeurs du dictionnaire sont accessibles par leurs clés. Ici, e prendra la valeur v1. Provoque une erreur si la clé n'existe pas.

`monDico[c3] = v3`: ajoute une nouvelle valeur au dictionnaire (ici v3) avec une clé (ici c3). Si la clé existe déjà, la valeur associée est modifiée.

`del monDico[C3]`: supprime une association dans le dictionnaire. La clé doit exister.

`c in monDico`: vérifie l'existence d'une clé dans le dictionnaire, renvoie True ou False.

`dic2 = monDico`: crée un synonyme (2ème nom au dico)
`dic3 = dict(monDico)`: crée une copie de surface (clone)
`dic4 = copy.deepcopy(monDico)`: crée une copie profonde (récursive)

Gestion des fichiers

`f=open('data.txt')`: ouvrir un fichier en lecture seule
`f=open('data.txt','w')`: ouvre un fichier en écriture (attention s'il existe il est écrasé, sinon il est créé)
`f=open('data.txt','a')`: ouvre un fichier en écriture (ajoute le texte à la fin)

`texte = f.read()`: lire tout le fichier en une seule fois
`lignes = f.readlines()`: lire en 1 fois toutes les lignes du fichier et les stocker dans une liste (un élém=une ligne)

```
for ligne in f:
    instructions
```

Lire le fichier ligne par ligne dans une boucle for

`f.write(texte)`: écrire dans un fichier (`texte` doit obligatoirement être une `string`).
Ne saute pas de ligne automatiquement à la fin du texte. `'\n'` code un saut de ligne.

`f.close()`: ferme un fichier