**Commissariat Général à l'Energie Atomique**
**Centre Régional d'Etudes Nucléaires de Kinshasa**
**C.G.E.A./C.R.E.N-K.**

# Gestion de données en Python

**Prof. Dr Saint-Jean DJUNGU**

**Février 2024**

*Tél : +243 818899608* *email :saintjean.djungu.ahuka@cgea-rdc.org*

# Chapitre 1 : Fonctions et modules

## 1.1. Définitions

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles permettent également de rendre le code plus lisible et plus clair en le fractionnant en blocs logiques. Nous avons déjà fait allusion à certaines fonctions internes à Python comme *range()* ou *len()*.

Par définition, une *fonction* est un morceau de code que l'on peut appeler à tout moment dans une autre partie de son code.

Lorsque l'on a un nombre important de fonctions, il est possible de les regrouper dans des *modules.*

**Syntaxe** :

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

**Exemple 1**

```
# Le but de cette fonction est d'écrire dans la console
def HelloWorld() :
    print("Hello world")
```

**Exemple 2**

```
def table(base):
    n = 1
    while n < 11:
        print(n * base, end = ' ')
        n = n + 1
```

La fonction *table()* telle que définie ci-dessus utilise le paramètre *base* pour calculer les dix premiers termes de la table de multiplication correspondante.

## 1.2. Utilisation d'une fonction

Le but d'une fonction est de pouvoir être appelée à différents endroits sans pour autant devoir répéter les instructions.

```
a = 1
while a < 20:
        table(a)
        a = a + 1
```

**Exemple complet**

```
# Le but de cette fonction est d'écrire dans la console
def nonConduite() :
        print("Vous ne pouvez pas conduire seul")


# Le programme suivant va poser des questions et déterminer si la personne
# peut conduire
if input("Etes-vous majeur ? (1/0) \n >>>")==1 :
        if input("Avez-vous le permis ? (1/0) \n >>>")==1 :
                if input("Etes-vous sous forte medication ? (1/0) \n >>>")==1 :
                        noConduite()
                else :
                        print("Vous pouvez conduire seul")
        else :
                noConduite()
else ;
        noConduite()
```

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

### 1.3. Fonction avec valeurs de retour

Une fonction peut également utiliser des arguments (données d'entrée) et renvoyer des résultats.

**Exemple 1**

```
# definition d'une fonction carre()
def carre(x):
        y = x**2
        return y
```

```
# programme principal
z = 5
resultat = carre(5)
print(resultat)
```

**Exemple 2**

```
# Trouve les coefficients directeurs d'une droite
def coefDroite(x1, x2, y1, y2) :
    a = (y2 – y1) / (x2 – x1)
    b = y1 – a * x1
    return a, b

# Debut du programme
x, y = coefDroite(1.0, 2.0, 0.0, 1.0)
print (x, y)
```

**Exemple 3 : Fonction qui retourne une liste**

```
def table(base) :
    resultat = []                    # resultat est d'abord une liste vide
    n = 1
    while n  < 11 :
        b = n * base
        resultat.append(b)   # ajout d'un terme à la liste
        n = n + 1
    return resultat
tab9 = table(9)
print(tab9)
print(tab9[0])
print(tab9[3])
print(tab9[2 :5])
```

**Résultats** *:*

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

### 1.4. Variables locales et variables globales

Lorsqu'on manipule des fonctions, il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite *locale* lorsqu'elle est créée dans une fonction. Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.

Une variable est dite *globale* lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.

Nous allons prendre un exemple simple qui vous aidera à mieux saisir ces concepts. Observez le code suivant :

```
# définition d'une fonction carre()
def carre(x ):
        y = x **2 4
        return y

# programme principal
z = 5
resultat = carre(z)
print (resultat)
```

La variable z est globale, alors que y est une variable locale.

### 1.5. Fonctions prédéfinies

Tout langage de programmation propose un certain nombre de *fonctions***.** Certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs – et pénibles - algorithmes.

Python possède une série de *fonctions prédéfinies* afin de réaliser les fonctions les plus courantes. Dès lors, les *fonctions prédéfinies* sont :

- soit des fonctions faisant partie intégrante du langage Python comme *print* , *len* et *str*, par exemple ;
- soit des fonctions ayant été écrites par d'autres programmeurs et mises à disposition en important le module qui les contient, comme la fonction *sqrt* du module *math* , par exemple.

**Exemples***:*

*>>>import math*

*>>>math.e*

*2.718281828459045*

*>>>math.pi*

*3.141592653589793*

*>>>math.sqrt(16)*

*4.0*

| **Nom de la fonction** | **Description** |
|---|---|
| abs() | Valeur absolue |
| ceil() | Arrondissement par excès d'un nombre réel |
| cos() | Cosinus, en radians |
| floor() | Arrondissement par défaut d'un nombre réel |
| log() | Logarithme naturel, en base e |
| log10() | Logarithme naturel, en base 10 |
| max() | Maximum entre deux valeurs |
| min() | Minimum entre deux valeurs |
| round() | Génératrice des nombres aléatoires |
| sin() | Sinus, en radians |
| sqrt() | Racine carrée d'un nombre |
| pow(x, y) | x à la puissance y |

Tableau 1.1 : Fonctions mathématiques classiques

## 1.6. Modules

### 1.6.1. Définition

Les *modules* sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi *bibliothèques* ou *libraries*). Ce sont des « boîtes à outils » qui vont vous être très utiles.

Les développeurs de Python ont mis au point de nombreux modules qui effectuent une quantité phénoménale de tâches. Pour cette raison, prenez toujours le réflexe de vérifier si une partie du code que vous souhaitez écrire n'existe pas déjà sous forme de module.

Pour mieux comprendre la distinction entre la définition d'une fonction et son utilisation au sein d'un programme, il faut :

- placer vos définitions de fonctions dans un module Python, et
- le programme qui les utilise dans un autre

### 1.6.2. Exemple et utilisation

**Exemple d'un module**

```
def table(base):
    n = 1
    while n < 11:
        print(n * base, end = ' ')
        n = n + 1
```

Engirestrez le code ci-dessus au nom de *fonctionTable.py*

**Utilisation d'un module**

```
from fonctionTable import *
a = 1
while a < 20:
    table(a)
    a = a + 1
```

### 1.6.3. Quelques modules courants

Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En voici une liste non exhaustive. Pour la liste complète, reportez-vous à la page des modules sur le site de Python :

- *math* : fonctions et constantes mathématiques de base (*sin, cos, exp, pi*. . . ).
- *sys* : interaction avec l'interpréteur Python, passage d'arguments
- *os* : dialogue avec le système d'exploitation
- *random* : génération de nombres aléatoires.

- *time* : accès à l'heure de l'ordinateur et aux fonctions gérant le temps.
- *urllib* : récupération de données sur internet depuis Python.
- *Tkinter* : interface python avec *Tk*. Création d'objets graphiques
- *re* : gestion des expressions régulières.

### 1.7. Exercices

**Exercice 1.1**

Ecrire une fonction qui retourne la somme de deux entiers et un algorithme qui l'utilise.

**Exercice 1.2**

Écrire une fonction qui convertit les kilomètres en miles (1 mile = 1,609 km).

**Exercice 1.3**

Écrire une fonction qui convertit les degrés Fahrenheit en degrés centigrades.

Formule: $t_c = \frac{5}{9}(t_F - 32)$

**Exercice 1.4**

Écrire une fonction qui calcule le volume d'une sphère.

**Exercice 1.5**

Écrire une fonction ayant en paramètres le nombre d'heures effectuées par un salarié dans la semaine et son salaire horaire, qui retourne sa paye hebdomadaire. On prendra en compte les heures supplémentaires (au-delà de 35 heures) payées à 150%.

**Exercice 1.6**

Ecrire une procédure qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de voyelles contenues dans cette phrase.

**Exercice 1.7**

Ecrire une procédure qui indique le nombre d'apparition d'une voyelle dans une phrase.

**Exercice 1.8**

Ecrire un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces.

**Exercice 1.9**

Ecrire un algorithme qui demande une phrase à l'utilisateur. Celui-ci entrera ensuite le rang d'un caractère à supprimer, et la nouvelle phrase doit être affichée.

**Exercice 1.10**

Un professeur note les résultats d'un test portant sur 50 questions en utilisant la table suivante:

| Bonnes réponses | 0 - 10 | 11 - 20 | 21 - 30 | 31 - 40 | 41 - 50 |
|---|---|---|---|---|---|
| Note | E | D | C | B | A |

Écrire une fonction qui retourne la note, étant donné un nombre de bonnes réponses.

**Exercice 1.11**

Écrire une fonction ayant en paramètre un entier représentant une année et retournant 1 (vrai) si l'année est bissextile et 0 (faux) sinon.

**Exercice 1.12**

Écrire une fonction ayant en paramètre deux entiers représentant un mois et une année et retournant le nombre de jours du mois de cette année.

**Exercice 1.13**

Un des plus anciens systèmes de cryptographie (aisément déchiffrable) consiste à décaler les lettres d'un message pour le rendre illisible. Ainsi, les A deviennent des B, les B des C, etc. Ecrire une procédure qui demande une phrase à l'utilisateur et qui la code selon ce principe.

**Exercice 1.14**

Une amélioration (relative) du principe précédent consiste à opérer avec un décalage

non de 1, mais d'un nombre quelconque de lettres. Ainsi, par exemple, si l'on choisit un décalage de 12, les A deviennent des M, les B des N, etc.

Réaliser une procédure sur le même principe que le précédent, mais qui demande en plus quel est le décalage à utiliser.

**Exercice 1.15**

Une technique ultérieure de cryptographie consista à opérer non avec un décalage systématique, mais par une substitution aléatoire. Pour cela, on utilise un alphabet-clé, dans lequel les lettres se succèdent de manière désordonnée, par exemple :

HYLUJPVREAKBNDOFSQZCWMGITX

C'est cette clé qui va servir ensuite à coder le message. Selon notre exemple, les A deviendront des H, les B des Y, les C des L, etc.

Ecrire une procédure qui effectue ce cryptage (l'alphabet-clé sera saisi par l'utilisateur, et on suppose qu'il effectue une saisie correcte).

**Exercice 1.16**

Écrire une procédure qui lit au clavier un verbe du premier groupe (il s'assurera qu'il est bien terminé par *er*) et qui en affiche la conjugaison au présent de l'indicatif. On supposera qu'il s'agit d'un verbe régulier. Autrement dit, on admettra que l'utilisateur ne fournit pas un verbe tel que *manger* (dans ce cas, le programme affichera *nous mangons* !). Les résultats se présenteront ainsi :

donnez un verbe régulier du premier groupe : dire

*** il ne se termine pas par er - donnez-en un autre : chanter

je chante

tu chantes

il/elle chante

nous chantons

vous chantez

ils/elles chantent

**Exercice 1.17**

Écrire une fonction qui retourne le plus grand commun diviseur (pgcd) de deux nombres entiers positifs.

L'algorithme d'Euclide est basé sur le principe suivant :

$$\text{pgcd}(a, b) = \begin{cases} a & \text{si } b = 0 \\ \text{pgcd}(b, a \bmod b) & \text{sinon} \end{cases}$$

**Exercice 1.18**

Écrire une fonction qui retourne le plus petit diviseur premier d'un nombre.

**Exercice 1.19**

Écrire une fonction qui calcule le n$^{e}$ terme de la suite suivante :

$$u_0 = \frac{1}{3}$$

$$u_{n+1} = \frac{4}{3}u_n - 1$$

**Exercice 1.20**

Écrire une fonction qui retourne le n$^{e}$ terme d'une suite de Fibonacci initialisée par a et b.

$$u_0 = a$$

$$u_1 = b$$

$$u_n = u_{n-1} + u_{n-2}\ pour\ n \geq 2$$

# Chapitre 2 : Listes et chaînes de caractères

## 2.1. Définition

Une *liste* est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de type différent (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.

Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets. En voici quelques exemples :

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

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie. Il existe deux types de listes :

- celles qui ne peuvent être modifiées, appelées *tuples*
- celles qui sont modifiables, appelées simplement *listes*

### a. Tuples

Les tuples sont notés sous forme d'éléments entre parenthèses séparés par des virgules

```
a = ('un', 2, 'trois')
```

Une liste d'un seul élément correspond à l'élément lui-même. La fonction *len()* renvoie le nombre d'éléments de la liste

```
>>> a = ('un', 2, 'trois')
>>>a
```

*(’un’, 2, ’trois’*

*>>>ln(a)*

*3*

**b. Listes modifiables**

Elles sont notées sous forme d’éléments entre crochets séparés par des virgules. Elles correspondent à des objets contrairement aux tuples.

*>>> a = [’un’, 2, ’trois’]*

*>>>a*

*[’un’, 2, ’trois’]*

*>>>ln(a)*

*3*

**2.2. Utilisation**

Un des gros avantages d’une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé *indice* (ou *index*) de la liste.

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

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

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

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

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

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

### 2.3. Opération sur les listes

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

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

L'opérateur + est très pratique pour concaténer deux listes. Vous pouvez aussi utiliser la fonction *append()* lorsque vous souhaitez ajouter un seul élément à la fin d'une liste. Dans l'exemple suivant nous allons créer une liste vide puis lui ajouter deux éléments, d'abord avec la concaténation :

```
>>> l = [] # liste vide
>>> l
[]
>>> l = l + [15]
>>> l
[15]
>>> l = l + [-5]
>>> l
[15, -5]
```

puis avec la fonction *append()* :

```
>>> l = []
>>> l
[]
>>> l.append(15)
>>> l
[15]
>>> l.append(-5)
>>> l
[15, -5]
```

Dans l'exemple ci-dessus nous obtenons le même résultat pour faire grandir une liste en utilisant l'opérateur de concaténation ou la fonction *append()*. Nous vous conseillons néanmoins dans ce cas précis d'utiliser la fonction *append()* dont la syntaxe est plus élégante.

**Nota** : A part la fonction *append()*, on peut utiliser :

- *pop()* pour enlèver le dernier élément
- *pop(i)* pour retirer le *ième* élément
- *extend* pour concatèner deux listes
- *sort* pour trier les éléments
- *reverse* pour inverser l'ordre des éléments
- *index(e)* pour retourner la position de l'élément *e*

**Exemples**

```
>>> liste = ['cochon','vache', 'chien','chat']
>>> liste.append("poule")
>>> liste
['cochon','vache','chien','chat','poule']
>>> liste.insert(2, "mouton")
>>> liste
['cochon','vache','mouton','chien','chat','poule']
>>> del liste[3]
>>> liste
```

*['cochon','vache','mouton','chat','poule']*

*>>> liste.remove("vache")*

*>>> liste*

*['cochon','mouton','chat','poule']*

*>>> liste.reverse()*

*>>> liste*

*['poule','chat','mouton','cochon']*

*>>> liste.sort()*

*>>> liste*

*['chat','cochon','mouton','poule']*

### 2.4. Indiçage négatif et tranches

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

*liste : ['girafe', 'tigre', 'singe', 'souris']*

*indice positif : 0 1 2 3*

*indice négatif : -4 -3 -2 -1*

ou encore :

*liste : ['A','C','D','E','F','G','H','I','K','L']*

*indice positif : 0 1 2 3 4 5 6 7 8 9*

*indice negatif : -10 -9 -8 -7 -6 -5 -4 -3 - 2 -1*

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de cette liste. L'avant-dernier élément a lui l'indice -2.

*>>> animaux = ['girafe','tigre','singe','souris']*

*>>> animaux[-1]*

*'souris'*

*>>> animaux[-2]*

*'singe'*

Pour accéder au premier élément de la liste, il faut par contre connaître le bon indice :

```
>>> animaux[-4]
'girafe'
```

Dans ce cas, on utiliserait plutôt *animaux[0]*. Un autre avantage des listes est la possibilité de sélectionner une partie en utilisant un indiçage construit sur le modèle *[m:n+1]* pour récupérer tous les éléments, du émième au énième (de l'élément *m* inclus à l'élément *n+1* exclus). On dit alors qu'on récupère une tranche de la liste, par exemple :

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

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

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

```
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3] [
1, 4]
```

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

### 2.5. Fonctions range() et list()

L'instruction *range()* est une fonction spéciale en Python qui va nous permettre de générer des nombres entiers compris dans un intervalle lorsqu'elle est utilisée en combinaison avec la fonction *list()* . Par exemple :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La commande *list(range(10))* a généré une liste contenant tous les nombres entiers de 0 inclus à 10 exclus. Dans l'exemple ci-dessus, la fonction *range()* a pris un argument, mais elle peut également prendre deux ou trois arguments, voyez plutôt :

```
>>> list(range(0,5))
[0, 1, 2, 3, 4]
>>> list(range(15,20))
[15, 16, 17, 18, 19]
>>> list(range(0,1000,200))
[0, 200, 400, 600, 800]
>>> list(range(2,-2,-1))
[2, 1, 0, -1]
```

L'instruction *range()* fonctionne sur le modèle *range([début,] fin[, pas])*. Les arguments entre crochets sont optionnels. Pour obtenir une liste, il faut l'utiliser systématiquement avec la fonction *list()*. Enfin, prenez garde aux arguments optionnels par défaut (0 pour début et 1 pour pas) :

```
>>> list(range(10,0))
[]
```

Ici la liste est vide car Python a pris la valeur du pas par défaut qui est de 1. Ainsi, si on commence à 10 et qu'on avance par pas de 1, on ne pourra jamais atteindre 0. Python génère ainsi une liste vide. Pour éviter ça, il faudra absolument préciser le pas de -1 pour les listes décroissantes :

```
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

**2.6. Listes de listes**

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

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

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

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

Pour accéder à un élément de la sous-liste, on utilise un double indiçage :

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

Il existe en Python les dictionnaires qui sont très pratiques pour faire ce genre de choses. Il existe aussi un module nommé numpy permettant de gérer des listes ou tableaux de nombres (vecteurs et matrices), ainsi que de faire des opérations avec.

### 2.7. Tableaux

Un *tableau* à une dimension correspond à une liste. Il n'existe pas de tableau à deux dimensions en Python comme dans d'autres langages de programmation. Un tableau à deux dimensions correspond à une liste de liste.

**Exemple**

```
nbLignes = 5
nbColonnes = 4
tableau2D = []
for i in range(0, nbLignes):
        tableau2D.append([])
        for j in range(0, nbColonnes):
                tableau2D[i].append(0)
tableau2D[2][3]='a'
tableau2D[4][3]='b'
print(tableau2D)
```

Ce qui donne: *[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 'a'], [0, 0, 0, 0], [0, 0, 0, 'b']*

**Nota** : Il est possible de généraliser à des tableaux de dimensions supérieures

### 2.8. Chaînes de caractères et listes

Les chaînes de caractères peuvent être considérées comme des listes (un peu particulières).

```
>>> animaux = "girafe tigre"
>>> animaux
'girafe tigre'
>>> len(animaux)
12
>>> animaux[3]
```

*'a'*

Nous pouvons donc utiliser certaines propriétés des listes comme les tranches :

```
>>> animaux = "girafe tigre"
>>> animaux[0:4]
'gira'
>>> animaux[9:]
'gre'
>>> animaux[:-2]
'girafe tig'
```

Mais a contrario des listes, les chaînes de caractères présentent toutefois une différence notable, ce sont des listes non modifiables. Une fois définie, vous ne pouvez plus modifier un de ses éléments. Le cas échéant, Python renvoie un message d'erreur :

```
>>> animaux = "girafe tigre"
>>> animaux[4]
'f'
>>> animaux[4] = "F"
Traceback (most recent call last):
    File "", line 1, in TypeError: 'str'
object does not support item assignment
```

Par conséquent, si vous voulez modifier une chaîne, vous êtes obligés d'en construire une nouvelle. Pour cela, n'oubliez pas que les opérateurs de concaténation (+) et de duplication (*) peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne de caractères.

### 2.9. Méthodes associées aux chaînes de caractères

Voici quelques *méthodes* spécifiques aux objets de type string :

```
>>> x = "girafe"
>>> x.upper()
```

*'GIRAFE'*

*>>> x*

*'girafe'*

*>>> 'TIGRE'.lower()*

*'tigre'*

Les fonctions *lower()* et *upper()* renvoient un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces fonctions n'altèrent pas la chaîne de départ mais renvoie la chaîne transformée. Pour mettre en majuscule la première lettre seulement, vous pouvez faire :

*>>> x[0].upper() + x[1:]*

*'Girafe'*

ou encore plus simple avec la fonction Python adéquate :

*>>> x.capitalize()*

*'Girafe'*

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la fonction split() :

*>>> animaux = "girafe tigre singe"*

*>>> animaux.split()*

*['girafe', 'tigre', 'singe']*

*>>> for animal in animaux.split():*

*... print animal*

*...*

*girafe*

*tigre*

*singe*

La fonction *split()* découpe la ligne en plusieurs éléments appelés champs, en utilisant comme séparateur les espaces ou les tabulations. Il est possible de modifier le séparateur de champs, par exemple :

```
>>> animaux = "girafe:tigre:singe"
>>> animaux.split(":")
['girafe', 'tigre', 'singe']
```

La fonction *find()* recherche une chaîne de caractères passée en argument.

```
>>> animal = "girafe"
>>> animal.find('i')
1
>>> animal.find('afe') 3
>>> animal.find('tig')
-1
```

Si l'élément recherché est trouvé, alors l'indice du début de l'élément dans la chaîne de caractères est renvoyé. Si l'élément n'est pas trouvé, alors la valeur -1 est renvoyée. Si l'élément recherché est trouvé plusieurs fois, seul l'indice de la première occurrence est retourné :

```
>>> animaux = "girafe tigre"
>>> animaux.find("i")
1
```

On trouve aussi la fonction *replace()*, qui serait l'équivalent de la fonction de substitution de la commande Unix *sed* :

```
>>> animaux = "girafe tigre"
>>> animaux.replace("tigre", "singe")
'girafe singe'
>>> animaux.replace("i", "o")
'gorafe togre'
```

Enfin, la fonction *count()* compte le nombre d'occurrences d'une chaîne de caractères passée en argument :

```
>>> animaux = "girafe tigre"
>>> animaux.count("i")
2
>>> animaux.count("z")
0
>>> animaux.count("tigre")
1
```

## 2.10. Exercices

### Exercice 2.1

Ecrire un programme Python qui détermine le nombre d'occurrences de *x* dans un tableau de dix entiers.

### Exercice 2.2

Ecrire un programme Python qui permet de faire la somme de deux vecteurs de taille n.

### Exercice 2.3

Ecrire un programme Python qui permet le produit scalaire de deux vecteurs de taille n.

### Exercice 2.4

Votre tante fortunée vous envoie désormais un montant chaque mois. A la fin de l'année vous souhaitez faire un bilan de vos richesses. Ecrire un programme Python qui affiche la somme mensuelle moyenne reçue, le montant minimal reçu sur l'année et le montant maximal reçu sur l'année.

### Exercice 2.5

Pour sa naissance, la grand-mère de Gabriella place une somme de 1000 dollars sur son compte épargne rémunéré au taux de 2.25% (chaque année le compte est augmenté de 2.25%). Développer un programme Python permettant d'afficher un tableau sur 20 ans associant à chaque anniversaire de Gabriela la somme acquise sur son compte.

**Exercice 2.6**

Ecrire un programme Python permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.

**Exercice 2.7**

Ecrire un programme Python calculant la somme des valeurs d'un tableau (on suppose que le tableau a été préalablement saisi).

**Exercice 2.8**

Ecrire un programme Python permettant, à l'utilisateur de saisir les notes des étudiants d'une promotion. Le programme, une fois la saisie terminée, renvoie le nombre de ces notes supérieures à la moyenne de la promotion.

**Exercice 2.9**

Ecrire un programme Python qui calcule le plus grand écart dans un tableau d'entiers. Rappel : L'écart entre deux entiers $x$ et $y$ est la valeur absolue de leur différence $|x - y|$.

**Exercice 2.10**

Ecrire un programme Python qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi (« les premiers seront les derniers… »).

**Exercice 2.11**

Ecrire un programme Python qui permette à l'utilisateur de supprimer une valeur d'un tableau préalablement saisi. L'utilisateur donnera l'indice de la valeur qu'il souhaite supprimer. Attention, il ne s'agit pas de remettre une valeur à zéro, mais bel et bien de la supprimer du tableau lui-même.

**Exercice 2.12**

Ecrire un programme Python qui permet de calculer la trace (somme des éléments de la diagonale principale) d'une matrice carrée.

**Exercice 2.13**

Soit un tableau Tab à deux dimensions (10,6) préalablement rempli de valeurs numériques. Écrire un programme Python qui recherche la plus grande valeur au sein de ce tableau.

**Exercice 2.14**

Ecrire un programme Python qui permet de calculer le produit d'une matrice par un vecteur.

**Exercice 2.15**

Ecrire un programme Python qui permet de calculer le produit de deux matrices.

**Exercice 2.16**

On se propose d'établir les résultats d'examen d'un ensemble d'étudiants d'une promotion. Chaque étudiant sera représenté par un objet comportant obligatoirement les champs suivants :

- le nom de l'étudiant,
- son admissibilité à l'examen, sous forme d'une valeur d'un type énuméré comportant les valeurs suivantes : N (non admis), P (passable), AB (*Assez bien)*, B (*Bien*), TB (*Très bien)*.

Idéalement, les noms des étudiants pourraient être contenus dans un fichier. Ici, par souci de simplicité, nous les supposerons fournis par un tableau de chaînes.

On demande d'écrire un programme Python qui :

- pour chaque étudiant, lit au clavier, par exemple 3 notes d'examen, en calcule la moyenne et renseigne convenablement le champ d'admissibilité, suivant les règles usuelles :
    - moyenne < 10 : Non admis
    - 10 <= moyenne <12 : Passable
    - 12 <= moyenne <14 : Assez bien
    - 14 <= moyenne <16 : Bien
    - 16 <= moyenne : Très bien

- affiche l'ensemble des résultats en fournissant en clair la mention obtenue.

Voici un exemple d'exécution d'un tel programme :

Donnez les trois notes de l'étudiant Donel

11.5 14.5 10

Donnez les trois notes de l'étudiant Amel

9.5 10.5 9

Donnez les trois notes de l'étudiant Grael

14.5 12 16.5

Donnez les trois notes de l'étudiant Coel

6 14 11

Donnez les trois notes de l'étudiant Anael

17.5 14 18.5

Résultats :

Donel - Assez bien

Amel - Non admis

Grael - Bien

Coel - Passable

Anael - Très bien

**Exercice 2.17**

Soit la liste ['girafe', 'tigre', 'singe', 'souris']. Avec une boucle, affichez chaque élément ainsi que sa taille (nombre de caractères).

# Chapitre 3 : Fichiers

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste) ou en cours de route par l'utilisateur. Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'informations plus importante.

## 3.1. Utilité des fichiers

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

L'idée la plus simple consiste à placer chacun de ces textes dans une variable, en début de programme, avec des instructions d'affectation du genre :

```
a = "Quelle est la capitale de la RD Congo ?"
b = "Qui a succédé à Mobutu ?"
c = "Combien font 26 x 43 ?"
... etc.
```

Cette idée est malheureusement beaucoup trop simpliste. Tout va se compliquer en effet lorsque nous essayerons d'élaborer la suite du programme, c'est-à-dire les instructions qui devront servir à sélectionner au hasard l'une ou l'autre de ces questions pour les présenter à l'utilisateur. Employer par exemple une longue suite d'instructions *if ... elif ... elif ...* comme dans l'exemple ci-dessous n'est certainement pas la bonne solution (ce serait d'ailleurs bien pénible à écrire : n'oubliez pas que nous souhaitons traiter un grand nombre de questions !) :

```
if choix == 1:
        selection = a
elif choix == 2:
        selection = b
elif choix == 3:
        selection = c
        ... etc.
```

La situation se présente déjà beaucoup mieux si nous faisons appel à une liste :

liste = ["Qui a vaincu Mobutu ?", "Comment traduit-on ’informatique’ en anglais ?", "Quelle est la formule chimique du méthane ?", ... etc ...]

On peut en effet extraire n’importe quel élément de cette liste à l’aide de son indice. Exemple :

*print(liste[2]) ===> "Quelle est la formule chimique du méthane ?"*

Même si cette façon de procéder est déjà nettement meilleure que la précédente, nous sommes toujours confrontés à plusieurs problèmes gênants :

- La lisibilité du programme va se détériorer très vite lorsque le nombre de questions deviendra important. En corollaire, nous accroîtrons la probabilité d’insérer une erreur de syntaxe dans la définition de cette longue liste. De telles erreurs seront bien difficiles à débusquer.
- L’ajout de nouvelles questions, ou la modification de certaines d’entre elles, imposeront à chaque fois de rouvrir le code source du programme. En corollaire, il deviendra malaisé de retravailler ce même code source, puisqu’il comportera de nombreuses lignes de données encombrantes.
- L’échange de données avec d’autres programmes (peut-être écrits dans d’autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.

Cette dernière remarque nous suggère la direction à prendre : il est temps que nous apprenions à séparer les données et les programmes qui les traitent dans des fichiers différents.

Pour que cela devienne possible, nous devrons doter nos programmes de divers mécanismes permettant de créer des fichiers, d’y envoyer des données et de les récupérer par la suite.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches. Lorsque les quantités de données deviennent très importantes, il devient d'ailleurs rapidement nécessaire de structurer les relations entre ces données, et l'on doit alors élaborer des systèmes appelés bases de données relationnelles, dont la gestion peut s'avérer très complexe. Lorsque l'on est confronté à ce genre de problème, il est d'usage de déléguer une bonne part du travail à des logiciels très spécialisés tels que *Access, Oracle, SQLServer, MySQL, PostgreSQL,* etc. Python est parfaitement capable de dialoguer avec ces systèmes, mais nous laisserons cela pour un peu plus tard.

Nos ambitions présentes sont plus modestes. Nos données ne se comptent pas encore par centaines de milliers, aussi nous pouvons nous contenter de mécanismes simples pour les enregistrer dans un fichier de taille moyenne, et les en extraire ensuite.

### 3.3. Travailler avec des fichiers

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

Tout ce que nous venons de dire des livres s'applique également aux fichiers informatiques. Un fichier se compose de données enregistrées sur votre disque dur, sur une clé USB ou un CD. Vous y accédez grâce à son nom (lequel peut inclure aussi un nom de répertoire). En première approximation, vous pouvez considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères.

### 3.4. Ouverture d'un fichier

La fonction "*open*" renvoie un objet de type *file* et sert à ouvrir les fichiers en :

- "*r*" lecture
- "*w*" écriture : le fichier est créé s'il n'existe pas, sinon il est écrasé

- "*a*" ajout : le fichier est créé s'il n'existe pas, sinon il est ouvert et l'écriture se fait à la fin

**Nota** : Pour vérifier que l'ouverture du fichier se fait correctement, il faut traiter une exception de type Exception (elle peut fournir une description de l'erreur).

```
try:
        fichier = open("lecture_fichier.txt","r")
except Exception, message:
        print(message)
```

### 3.5. Lecture dans un fichier

Une grande partie de l'information en biologie, par exemple, est stockée sous forme de texte dans des fichiers. Pour traiter cette information, vous devez le plus souvent lire ou écrire dans un ou plusieurs fichiers. Python possède pour cela de nombreux outils qui vous simplifient la vie.

#### a. Méthode .readlines()

L'instruction *.readlines()* retourne une liste des différentes lignes du fichier.

Créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire courant avec le nom *zoo.txt* et le contenu suivant :

*girafe*
*tigre*
*singe*
*souris*

**Exemple1**

```
filin = open("zoo.txt", "r ")
lignes=filin.readlines()
```

```
print(lignes)
filin.close()
```

Il y a plusieurs commentaires à faire sur cet exemple :

- ✓ Ligne 1 : L'instruction *open()* ouvre le fichier *zoo.txt*. Ce fichier est ouvert en lecture seule, comme l'indique le second argument *r* (pour *read*) de la fonction *open()*. Remarquez que le fichier n'est pas encore lu, mais simplement ouvert (un peu comme lorsqu'on ouvre un livre, mais qu'on ne l'a pas encore lu). Le curseur de lecture est prêt à lire le premier caractère du fichier. L'instruction *open("zoo.txt", "r")* suppose que le fichier *zoo.txt* est dans le répertoire depuis lequel l'interpréteur Python a été lancé. Si ce n'est pas le cas, il faut préciser le chemin d'accès au fichier. Par exemple, */home/nael/zoo.txt* pour Linux ou Mac OS X ou *C:\Users\nael\zoo.txt* pour Windows.
- ✓ Ligne 2 : La méthode *.readlines()* agit sur l'objet *filin* en déplaçant le curseur de lecture du début à la fin du fichier.
- ✓ Ligne 3 : Elle affiche une liste contenant toutes les lignes du fichier (dans notre analogie avec un livre, ceci correspondrait à lire toutes les lignes du livre).
- ✓ Ligne 4 : Enfin, on applique la méthode *.close()* sur l'objet *filin*, ce qui, vous vous en doutez, ferme le fichier (ceci correspondrait à fermer le livre). Vous remarquerez que la méthode *.close()* ne renvoie rien mais modifie l'état de l'objet *filin* en fichier fermé.

Après exécution, on a:

```
['girafe \n', 'tigre \n', 'singe \n', 'souris \n ']
```

**Exemple2**

```
filin = open ("zoo.txt ", "r ")
lignes=filin.readlines()
for uneLigne in lignes :
        print(lignes)
filin.close()
```

Après exécution, on a cette fois-ci:

*girafe*
*tigre*
*singe*
*souris*

**Nota**: Il existe d'autres méthodes que *.readlines()* pour lire (et manipuler) un fichier. Il y a par exemple, les méthodes:

- *.read() :* lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.
- *.readline() :* lit une ligne d'un fichier et la renvoie sous forme de chaîne de caractères. À chaque nouvel appel de *.readline()*, la ligne suivante est renvoyée.

**b. Itération directe sur le fichier**

Il existe en Python le mot-clé *with* qui permet d'ouvrir et de fermer un fichier de manière efficace. Si pour une raison ou une autre l'ouverture ou la lecture du fichier conduit à une erreur, l'utilisation de *with* garantit la bonne fermeture du fichier, ce qui n'est pas le cas dans le code précédent. Voici donc le même exemple avec with :

```
>>> with open ("zoo.txt", 'r ') as filin :
...     lignes = filin.readlines()
...     for ligne in lignes :
...         print (ligne)
...
girafe
tigre
singe
souris
>>>
```

**Nota** :

- L'instruction *with* introduit un bloc d'indentation. C'est à l'intérieur de ce bloc que nous effectuons toutes les opérations sur le fichier.
- Une fois sorti du bloc d'indentation, Python fermera automatiquement le fichier. Vous n'avez donc plus besoin d'utiliser la *méthode.close().*

### 3.6. Écriture dans un fichier

Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

```
>>> animaux2 = ["poisson", "abeille", "chat"]
>>> with open ("zoo2.txt" , "w") as filout :
...     for animal in animaux2 :
...         filout.write(animal)
...
7
7
4
```

Quelques commentaires sur cet exemple :

Ligne 1 : Création d'une liste de chaînes de caractères *animaux2*.

Ligne 2 : Ouverture du fichier *zoo2.txt* en mode écriture, avec le caractère *w* pour *write*. L'instruction *with* crée un bloc d'instructions qui doit être indenté.

Ligne 3 : Parcours de la liste *animaux2* avec une boucle *for*.

Ligne 4 : À chaque itération de la boucle, nous avons écrit chaque élément de la liste dans le fichier. La méthode *.write()* s'applique sur l'objet *filout*. Notez qu'à chaque utilisation de la méthode *.write()*, celle-ci nous affiche le nombre d'octets (équivalent au nombre de caractères) écrits dans le fichier (lignes 6 à 8). Ceci est valable uniquement dans l'interpréteur, si vous créez un programme avec les mêmes lignes de code, ces valeurs ne s'afficheront pas à l'écran. Si nous ouvrons le fichier *zoo2.txt* avec un éditeur de texte, voici ce que nous obtenons : *poissonabeillechat* Ce n'est pas exactement le résultat attendu car implicitement nous voulions le nom de chaque animal sur une ligne. Nous avons oublié d'ajouter le caractère fin de ligne après chaque nom d'animal. Pour ce faire, nous pouvons utiliser l'écriture formatée :

```
>>> animaux2 = ["poisson", "abeille", "chat"]
>>> with open ("zoo2.txt" , "w") as filout :
...     for animal in animaux2 :
...         filout.write(f"{animal}\n ")
...
8
8
5
```

Ligne 4 : L'écriture formatée vue au chapitre 2 *Affichage* permet d'ajouter un retour à la ligne (\n) après le nom de chaque animal.

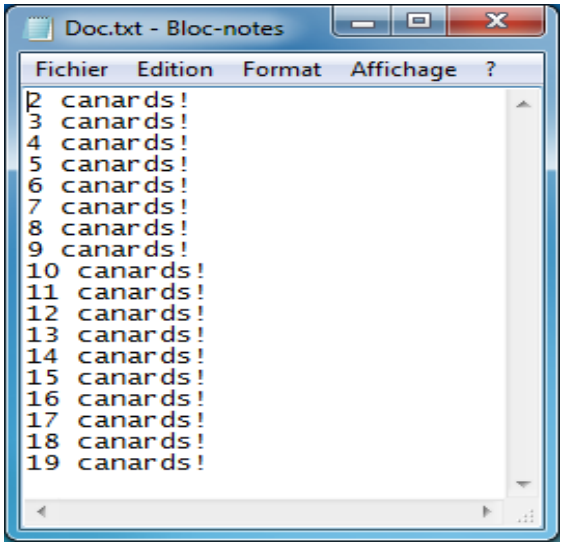
Lignes 6 à 8 : Le nombre d'octets écrits dans le fichier est augmenté de 1 par rapport à l'exemple précédent car le caractère retour à la ligne compte pour un seul octet. Le contenu du fichier *zoo2.txt* est alors :

```
poisson
abeille
chat
```

Vous voyez qu'il est extrêmement simple en Python de lire ou d'écrire dans un fichier.

Nota : Pour écrire dans un fichier, il est parfois nécessaire d'utiliser les alinéas (*'\t'* [*tabulation*]) ou les retours à la ligne ('\n' [*new line*]).

```
with open ("Doc.txt", "w") as fil :
    for i in range(2, 20):
        fil.write("%i canards!\n"%(i))
```




### 3.7. Ouverture de deux fichiers avec l'instruction with

On peut avec l'instruction *with* ouvrir deux fichiers (ou plus) en même temps. Voyez l'exemple suivant :

```
with open ("zoo1. txt", "r ") as fichier1 , open ("zoo2. txt ", "w ") as fichier2 :
    for ligne in fichier1 :
        fichier2.write ("* " + ligne )
```

Si le fichier *zoo1.txt* contient le texte suivant :

*souris*
*girafe*
*lion*
*singe*

alors le contenu de zoo2.txt sera :

** souris*
** girafe*
** lion*
** singe*

Dans cet exemple, *with* permet une notation très compacte en s'affranchissant de deux méthodes *.close()*. Si vous souhaitez aller plus loin, sachez que l'instruction *with* est plus générale et est utilisable dans d'autres contextes.

### 3.8. Importance des conversions de types avec les fichiers

Vous avez sans doute remarqué que les méthodes qui lisent un fichier (par exemple *.readlines()*) vous renvoient systématiquement des chaînes de caractères. De même, pour écrire dans un fichier, il faut fournir une chaîne de caractères à la méthode *.write()*.

Pour tenir compte de ces contraintes, il faudra utiliser les fonctions de conversions de types vues au chapitre 2 du Module 1 de ce cours : *int(), float()* et *str()*. Ces fonctions de conversion sont essentielles lorsqu'on lit ou écrit des nombres dans un fichier.

En effet, les nombres dans un fichier sont considérés comme du texte, donc comme des chaînes de caractères, par la méthode *.readlines()*. Par conséquent, il faut les convertir (en entier ou en float) si on veut effectuer des opérations numériques avec.

**Exemple**

```
with open("personne. txt", "w") as fich :
    for i in range(0, 3) :
        print("Saisir nom  prenom et age")
        nom = input("nom")
        fich.write(nom)
        fich.write(' ')
        nom = input("prenom")
        fich.write(prenom)
        fich.write(' ')
        age = int(input("age "))
        fich.write(str(age) + \n)
```

```
Enyungu Kevin 23
Kalonji Carine 17
Kasoro Josh 12
```

### 3.9. Formats de fichiers

#### 3.9.1. Types

Il existe différents formats standards de stockage de données. Pour ce faire, il est recommandé de favoriser ces formats car il existe déjà des modules Python permettant de simplifier leur utilisation.

De plus, ces formats sont adoptés par d'autres programmes avec lesquels vous serez peut-être amené à travailler.

**Exemples**

- Format CSV : format permettant de stocker des tableaux dans un fichier texte
- Format JSON : format de données très répandu permettant de stocker des données sous une forme structurée
- Format XLS : format de données Excel de Microsoft

### 3.9.2. Lecture dans un fichier cvs

Prenons le fichier *noms.cvs* ayant la forme suivante :

*Nom ; Prenom ; Age*

*Kakule ; Dana ; 29*

*Lueteta ; Kevin ; 47*

*Mutombo ; Nadia ; 51*

*Kasoro ; Josh ; 14*

Un programme Python permettant de lire dans le fichier *noms.cvs* peut prendre la forme suivante :

```
import csv
# Ouvrir le fichier csv
with open("noms.csv", "r") as f :
    # Créer un objet csv à partir du fichier
    obj = csv.reader(f)
    for ligne in obj :
        print(ligne)
f.close()
```

```
['Nom;Prenom;Age']
['Kakule;Dana;29']
['Lueteta;Kevin;47']
['Mutombo;Nadia;51']
['Kasoro;Josh;14']
```

**Nota** : CSV utilise Excel par défaut pour afficher le contenu

### 3.9.3. Ecriture dans un fichier cvs

Voici un exemple d'écriture :

```
import csv
print("Un programme qui utilise csv.writer() pour écrire dans un fichier")
print("\n")
# Les données que nous allons écrire
data = [('Nonge','Anael',40), ('Enyungu','Sarah',32), ('Mbungu','Nnael',38)]
# Ouvrir le fichier en mode écriture
with open("noms.csv", "a") as f :
    print("Nous avons ouvert le fichier noms")
    # Créer l'objet fichier
    obj = csv.writer(f)
    # Chaque élément de data correspond à une ligne
    for element in data :
        obj.writerow(element)
f.close()
```

## 3.10. Exercices

### Exercice 3.1

On travaille avec le fichier du carnet d'adresses en champs de largeur fixe. Ecrire un programme Python qui permet à l'utilisateur de saisir au clavier un nouvel individu qui sera ajouté à ce carnet d'adresses.

### Exercice 3.2

Ecrire un programme Python permettant de créer séquentiellement un fichier « répertoire » comportant pour chaque personne :

- nom (20 caractères maximum) ;
- prénom (20 caractères maximum) ;
- âge (entier) ;
- numéro de téléphone (10 caractères maximum)

Les informations relatives aux différentes personnes seront lues au clavier.

**Exercice 3.3**

Ecrire un programme Python permettant, à partir du fichier créé par l'exercice 3.2, de retrouver les informations correspondant à une personne de nom donné.

**Exercice 3.4**

On dispose d'un fichier disque dont le contenu doit être affiché. Chaque enregistrement du fichier contient les informations suivantes :

matricule nom de l'employé nombre d'heures prestées

Ecrire un programme Python permettant d'afficher tout le contenu du fichier.

**Exercice 3.5**

Ecrire un programme Python permettant, à partir du fichier créé par l'exercice 3.4, d'afficher le nombre total d'heures prestées à la fin de l'état.

**Exercice 3.6**

On dispose d'un fichier disque dont chaque enregistrement contient les informations suivantes :

no client code article quantité prix unitaire

Ecrire un programme Python permettant de calculer le prix d'achat de chaque client et d'afficher pour chaque client :

no client code article quantité prix d'achat

Il faut également afficher le nombre total des clients ainsi que le montant total dû par l'ensemble des clients à la fin de l'état.

**Exercice 3.7**

Le contenu d'un fichier disque doit être affiché. Chaque enregistrement contient les informations suivantes :

matricule nom code sexe montant

Le code sexe sera M ou F selon qu'il s'agit d'un homme ou d'une femme.

a) Afficher les noms et montants uniquement pour les femmes.

b) Afficher uniquement les noms des employés qui ont un montant supérieur ou égal à 1000 dollars.

c) Afficher uniquement les matricules des hommes qui ont un montant inférieur à 1500 dollars

d) Afficher uniquement les matricules des hommes qui ont un montant supérieur à 1000 dollars et les matricules des femmes qui ont un montant inférieur à 2500 dollars.

**Exercice 3.8**

Le contenu d'un fichier agents. Chaque enregistrement contient les informations suivantes :

matricule nom code sexe nombre d'heures

Le code sexe sera M ou F selon qu'il s'agit d'un homme ou d'une femme. Pour chaque cas suivant, afficher :

matricule nom salaire brut

a) Calculer le salaire brut des femmes, sachant que le salaire horaire est de 50 dollars
b) Calculer le salaire brut des hommes qui ont travaillé plus de 20 heures, le taux horaire de 60 dollars et d'afficher pour chacune d'eux :
c) Calculer le salaire brut de tous les employés, sachant que le salaire horaire est de 50 dollars pour les femmes et 60 dollars pour les hommes. Pour chaque employé, il faut afficher :
d) Calculer le salaire brut et le salaire net de tous les employés, sachant que :
    a. le salaire horaire est de 60 dollars pour les hommes qui ont travaillé au moins 20 heures ;
    b. le salaire horaire est de 50 dollars pour les femmes qui ont travaillé plus de 10 heures ;
    c. le salaire horaire est de 40 dollars pour les autres agents ;
    d. le taux d'impôt est de 20% pour les hommes et de 10% pour les femmes sur le salaire brut.

Pour chaque employé, il faut afficher :

matricule nom nombre heures sal. hor. Sal. brut impôt sal. net

**Exercice 3.9**

On dispose d'un fichier disque. Chaque enregistrement contient :

- code article
- libellé de l'article
- stock au début de la journée
- achats de la journée
- ventes de la journée
- stock de sécurité

Calculer pour chaque enregistrement le stock en fin de journée et d'afficher :

code article stock fin de journée astérisque

**Attention** : Si le stock en fin de journée est inférieur au stock de sécurité, il faut en outre afficher un astérisque à côté de stock fin de journée.

**Exercice 3.10**

On dispose d'un fichier disque des étudiants d'une promotion dont les enregistrements contiennent :

matricule nom taille en cm poids en kg

Afficher le contenu du fichier ainsi que, à la fin de d'état, la taille et le poids moyens des étudiants de la promotion.

**Exercice 3.11**

Un fichier disque de paie de juillet contient les enregistrements dont le format est le suivant :

matricule nom type-heures nombre heures

Le type-heures est *nor, sup, nut* selon qu'il s'agit des heures normales, supplémentaires ou de nuit prestées. Par ailleurs, il est clair qu'un agent peut avoir presté plusieurs types d'heures durant le mois concerné.

On demande, par type d'heures prestées, de compter le nombre d'enregistrements, de calculer le nombre d'heures prestées et de calculer le montant à payer, sachant que le taux horaire est de 15 dollars, 5 dollars et 20 dollars respectivement pour les heures normales, supplémentaires et de nuit.

Afficher ensuite le résultat du traitement selon le format suivant :

Type-hres nbre enregistrements nbre tot hres taux montant

A la fin de l'état, il faut aussi afficher le montant total dû pour l'ensemble des types d'heures prestées.

**Exercice 3.12**

On dispose d'un fichier disque Agents sur disque dont les enregistrements sont structurés comme suit :

matricule nom grade salaire base prime

Le taux de l'IPR dû à l'Etat par un agent est fonction de son salaire de base, soit :

- 10 % si salaire de base inférieur à 1000 dollars
- 15 % si salaire de base supérieur à 1000 dollars et inférieur à 2500 dollars
- 30 % si salaire de base supérieur à 2500 dollars et inférieur à 5000 dollars
- 50 % si salaire de base supérieur ou égal à 5000 dollars

Calculer et afficher pour chaque agent :

matricule grade salaire base prime salaire brut impôt salaire net

Donner également, à la fin de l'état, le nombre total d'agents, les montants totaux de salaire de base, primes, salaire brut, impôts et salaire net.

**Exercice 3.13**

On travaille avec le fichier du carnet d'adresses en champs de largeur fixe. Ecrire un algorithme qui permet à l'utilisateur de saisir au clavier un nouvel individu qui sera ajouté à ce carnet d'adresses. Cependant, le carnet est supposé être trié par ordre alphabétique. L'individu doit donc être inséré au bon endroit dans le fichier.

**Exercice 3.14**

Ecrire un programme Python qui permet de modifier un renseignement (pour simplifier, disons uniquement le nom de famille) d'un membre du carnet d'adresses. Il faut donc demander à l'utilisateur quel est le nom à modifier, puis quel est le nouveau nom, et mettre à jour le fichier. Si le nom recherché n'existe pas, le programme devra le signaler.

**Exercice 3.15**

Ecrire un programme Python qui trie les individus du carnet d'adresses par ordre alphabétique.

**Exercice 3.16**

Soient adresse1.txt et adresse2.txt deux fichiers dont les enregistrements ont la même structure. Ecrire un programme Python qui recopie tout le fichier adresse1 dans le fichier adresse3, puis à sa suite, tout le fichier adresse2 (concaténation de fichiers).

**Exercice 3.17**

Ecrire un programme Python qui supprime dans notre carnet d'adresses tous les individus dont le mail est invalide (pour employer un critère simple, on considèrera que sont invalides les mails ne comportant aucune arobase, ou plus d'une arobase).

**Exercice 3.18**

Les enregistrements d'un fichier contiennent les deux champs nom (chaîne de caractères) et montant (entier). Chaque enregistrement correspond à une vente conclue par un commercial d'une société.

Le fichier de départ est déjà trié alphabétiquement par vendeur. Un vendeur peut avoir plusieurs enregistrements pour avoir réalisé plusieurs ventes. Calculer et afficher, pour chaque vendeur, le total de ventes.

# Chapitre 4 : Gestion d'une base de données

## 4.1. Importance des bases de données

Les *bases de données* sont des outils de plus en plus fréquemment utilisés. Elles permettent de stocker des données nombreuses dans un seul ensemble bien structuré. Lorsqu'il s'agit de bases de données relationnelles, il devient en outre tout à fait possible d'éviter l'«enfer des doublons ». Vous avez sûrement été déjà confrontés à ce problème : des données identiques ont été enregistrées dans plusieurs fichiers différents. Lorsque vous souhaitez modifier ou supprimer l'une de ces données, vous devez ouvrir et modifier tous les fichiers qui la contiennent ! Le risque d'erreur est très réel, qui conduit inévitablement à des incohérences, sans compter la perte de temps que cela représente. Les bases de données constituent la solution à ce type de problème. Python vous fournit les moyens d'utiliser les ressources de nombreux systèmes, mais nous n'en examinerons qu'un dans nos exemples : MySQL.

MySQL est un *système de gestion de bases de données relationnelles* **(***SGBDR***)** open source populaire qui est utilisé pour stocker et gérer des données. Il a été conçu pour être rapide, fiable et facile à utiliser. Avec MySQL, vous pouvez créer et gérer des bases de données, ainsi qu'exécuter des instructions SQL pour gérer les enregistrements de données dans la base de données. Mais avant de parler MySQL, nous allons d'abord examiner ce qu'on la bibliothèque standard de Python à matière de bases de données.

## 4.2. SQLite

### 4.2.1. Introduction

La bibliothèque standard de Python inclut un moteur de base de données relationnelles très performant nommé *SQLite*, qui a été développé indépendamment en C, et implémente en grande partie le standard *SQL-92*.

Cela signifie donc que vous pouvez écrire en Python une application contenant son propre SGBDR intégré, sans qu'il soit nécessaire d'installer quoi que ce soit d'autre, et que les performances seront au rendez-vous.

Nous verrons en fin de chapitre comment les choses se présentent si votre application doit utiliser plutôt un serveur de bases de données hébergé par une autre machine, mais les principes resteront les mêmes. Tout ce que vous aurez appris à faire avec *SQLite* sera

transposable sans modification, si vous devez plus tard travailler avec un SGDBR plus «imposant» tel que MySQL, PostgreSQL ou Oracle.

Commençons donc tout de suite à explorer les bases de ce système. Nous écrirons ensuite un petit script pour gérer une base de données simple à deux tables.

### 4.2.2. Création de la base de données

Nous commençons par créer une petite base de données nommée *essai.sq3*, suivant le chemin *C:/CoursUniv/Python/Exemples/BDD/essai.sq3*, ayant une seule table appelée *membres*. Et ensuite, nous y insérons trois enregistrements.

```
import sqlite3
fichierDonnees ="C:/CoursUniv/Python/Exemples/BDD/essai.sq3"
# Connexion à la base de données
con =sqlite3.connect(fichierDonnees)
# Créer un curseur
cur = con.cursor()
# Créer la table et ses champs
cur.execute("CREATE TABLE membres(age INTEGER, nom TEXT, taille REAL)")
# Insérer des lignes
cur.execute("INSERT INTO membres(age,nom,taille) VALUES(15,'Donel',1.83)")
cur.execute("INSERT INTO membres(age,nom,taille) VALUES(13,'Amel',1.57)")
cur.execute("INSERT Into membres(age,nom,taille) VALUES(10,'Anael',1.69)")
# Transferer les données dans la bdd
con.commit()
# Fermer le curseur et la connexion
cur.close()
con.close()
```

**Nota** : On peut insérer plusieurs lignes comme suit :

```
data =[(17,"Dana",1.74),(22,"Nael",1.71),(20,"Winner",1.65)]
```

```
for tu in data:
        cur.execute("INSERT INTO membres(age,nom,taille) VALUES(?,?,?)", tu)
```

### 4.2.3. Connexion à une base de données existante

À la suite des opérations ci-dessus, un fichier nommé *essai.sq3* aura été crée à l'emplacement indiqué dans votre machine. Vous pourriez dès lors quitter Python, et même éventuellement éteindre votre ordinateur : les données sont enregistrées.

Maintenant, comment faut-il procéder pour y accéder à nouveau ? C'est très simple : il suffit d'utiliser exactement les mêmes instructions précédentes. L'interrogation de la base s'effectue bien évidemment à l'aide de requêtes SQL, que l'on confie à la méthode *execute()* du curseur, toujours sous la forme de chaînes de caractères.

```
import sqlite3
# Connexion à la base de données
con =sqlite3.connect("C:/CoursUniv/Python/Exemples/BDD/essai.sq3")
# Créer un curseur
cur = con.cursor()
# Sélectionner et afficher
cur.execute("SELECT * FROM membres")
for l in cur:
        print(l)
con.commit()
# Fermer le curseur et la connexion
cur.close()
con.close()
```

**Résultats :**

```
(15, 'Donel', 1.83)
(13, 'Amel', 1.57)
(10, 'Anael', 1.69)
```

### 4.2.4. Modification ou suppression d'un enregistrement

#### a. Modification

Pour modifier un ou plusieurs enregistrements, exécutez une requête du type :

```
cur.execute("UPDATE membres SET nom ='Coel' WHERE nom='Anael'")
```

**b. Suppression**

Pour supprimer un ou plusieurs enregistrements, utilisez une requête telle que :

*cur.execute("DELETE FROM membres WHERE nom='Coel'")*

**4.2.5. Requêtes en ligne**

Le petit script ci-dessous est fourni à titre purement indicatif. Il s'agit d'un client SQL rudimentaire, qui vous permet de vous connecter à la base de données « *essai* » qui devrait à présent exister dans l'un de vos répertoires, d'y ouvrir un curseur et d'utiliser celui-ci pour effectuer des requêtes. Notez encore une fois que rien n'est transcrit sur le disque tant que la méthode *commit()* n'a pas été invoquée.

```
# Utilisation d'une petite base de données acceptant les requêtes SQL
import sqlite3
baseDonn = sqlite3.connect("C:/CoursUniv/Python/Exemples/BDD/essai.sq3")
cur = baseDonn.cursor()
while 1:
    print("Veuillez entrer votre requête SQL (ou <Enter> pour terminer) :")
    requete = input()
    if requete =="":
        break
    try:
        cur.execute(requete) # exécution de la requête SQL
    except:
        print('*** Requête SQL incorrecte ***')
    else:
        for enreg in cur: # Affichage du résultat
            print(enreg)
    print()

choix = input("Confirmez-vous l'enregistrement de l'état actuel (o/n) ? ")
if choix[0] == "o" or choix[0] == "O":
    baseDonn.commit()
else:
    baseDonn.close()
```

Il ne nous est pas possible de développer davantage le langage de requêtes dans le cadre restreint de ce cours. Nous allons cependant examiner encore un exemple de réalisation Python

faisant appel à un système de bases de données, mais en supposant cette fois qu'il s'agit de dialoguer avec un système serveur indépendant (lequel pourrait être par exemple un gros serveur de bases de données d'entreprise, un serveur de documentation dans une école, etc.). Comme il existe d'excellents logiciels libres et gratuits, vous pouvez aisément mettre en service vous-même un serveur extrêmement performant tel que MySQL. L'exercice sera particulièrement intéressant si vous prenez la peine d'installer le logiciel serveur sur une machine distincte de votre poste de travail habituel, et de relier les deux par l'intermédiaire d'une connexion réseau de type TCP/IP.

### 4.3. Usage de Python avec MySQL

Dans les lignes qui suivent, nous supposons que vous avez déjà accès à un serveur MySQL, sur lequel une base de données « *discotheque* » aura été créée pour l'utilisateur «*coel*», lequel s'identifie à l'aide du mot de passe « *abcde* ». Ce serveur peut être situé sur une machine distante accessible via un réseau, ou localement sur votre ordinateur personnel.

#### 4.3.1. Modules d'interaction avec une base de données MySQL

Python est un langage de programmation qui peut être utilisé pour communiquer et interagir avec MySQL et d'autres bases de données. Il existe plusieurs modules disponibles en Python qui vous permettent de vous connecter et d'interagir avec une base de données MySQL, notamment :

1. *mysql-connector-python***:** c'est un pilote Python pour MySQL qui est écrit en pur Python et qui ne nécessite aucune bibliothèque tiers (l'installation en ligne se fait en tapant sur la ligne de commandes : *pip install mysql-connector-python*).
2. *PyMySQL***:** c'est une bibliothèque Python qui implémente un client MySQL pur Python et qui peut être utilisée pour se connecter et interagir avec une base de données MySQL.
3. *MySQLdb***:** c'est une bibliothèque Python qui fournit une interface Python à la base de données MySQL. Elle nécessite l'installation de la bibliothèque client MySQL C.

Pour utiliser l'un de ces modules pour vous connecter et travailler avec une base de données MySQL en Python, vous devrez d'abord installer le module et l'utiliser pour établir une connexion à la base de données. Une fois que vous avez établi une connexion, vous pouvez exécuter des instructions SQL pour manipuler les données dans la base de données.

### 4.3.2. Connexion et sélection de données MySQL avec Python

Par exemple, pour vous connecter à une base de données MySQL en utilisant le module *mysql-connector-python*, vous pouvez utiliser le code suivant :

**Exemple**

```
import mysql.connector

# Connexion à la base de données
con = mysql.connector.connect(user='< nom utilisateur >', password=' mot de passe>', host='<nom hôte>', database='<nom base de données>')

# Créer un curseur
cur = con.cursor()

# Executer une réquete SQL
cur.execute("SELECT * FROM <table>")

# Accèder aux résulatts
resultats = cur.fetchall()

# Parcourir les résultats et les afficher
for res in resultats:
   print(res)

# Fermer le curseur et la connexion
cur.close()
con.close()
```

Ce code se connectera à une base de données MySQL, exécutera une instruction SELECT pour récupérer toutes les lignes d'une table et affichera ensuite les résultats.

### 4.3.3. Insertion de données MySQL avec Python

Pour insérer des données dans une base de données MySQL à partir d'un programme Python, vous pouvez utiliser l'instruction *SQL INSERT*.

Voici un exemple de comment insérer une nouvelle ligne dans une table MySQL en utilisant le module *mysql-connector-python* :

**Exemple**

```
# Insérer une nouvelle ligne
```

*cur.execute("INSERT INTO <table> (column1, column2, column3) VALUES (%s, %s, %s)", (value1, value2, value3))*

Dans cet exemple, l'instruction *INSERT* est utilisée pour ajouter une nouvelle ligne à la table, avec les valeurs *value1*, *value2* et *value3* pour les colonnes *column1*, *column2* et *column3*, respectivement. Les marqueurs de place *%s* sont utilisés pour indiquer que les valeurs seront fournies en tant que paramètres à la méthode *execute()*.

Vous pouvez également utiliser l'instruction *INSERT* pour insérer plusieurs lignes en une seule fois. Voici un exemple comment le faire :

**Exemple**

*# Insérer plusieurs lignes*
*values = [(value1_1, value2_1, value3_1), (value1_2, value2_2, value3_2), (value1_3, value2_3, value3_3)]*
*cur.executemany("INSERT INTO <table> (column1, column2, column3) VALUES (%s, %s, %s)", values*

Ce code insérera trois lignes dans la table, avec les valeurs *(value1_1, value2_1, value3_1), (value1_2, value2_2, value3_2) et (value1_3, value2_3, value3_3)* pour les colonnes *column1*, *column2* et *column3*, respectivement. La méthode *executemany()* est utilisée pour exécuter l'instruction *INSERT* plusieurs fois avec des valeurs différentes.

Il est important de se rappeler de valider les modifications apportées à la base de données à l'aide de la méthode *con.commit()*, sinon elles seront perdues lorsque la connexion sera fermée.

### 4.3.4. Mettre à jour des données MySQL avec Python

Pour *mettre à jour (update)* des données dans une *base de données MySQL* à partir d'un programme *Python*, vous pouvez utiliser l'instruction *SQL UPDATE*.

Voici un exemple de comment *mettre à jour une ligne* dans une *table MySQL* en utilisant le module *mysql-connector-python* :

**Exemple**

*# Mettre à jour une ligne*

*cur.execute("UPDATE <table> SET column1 = %s, column2 = %s WHERE id = %s", (new_value1, new_value2, id))*

Dans cet exemple, l'*instruction UPDATE* est utilisée pour modifier les valeurs des colonnes *column1* et *column2* pour la ligne avec la valeur *id spécifiée*. Les espaces réservés *%s* sont utilisés pour indiquer que les valeurs seront fournies en tant que paramètres à la méthode *execute()*.

Vous pouvez également utiliser l'*instruction UPDATE* pour mettre à jour plusieurs lignes à la fois en utilisant une *clause WHERE* pour spécifier une condition que les lignes doivent remplir.