CHAPITRE 2

Manipuler les données

1. **Liste**
2. Collections
3. Fichiers

Listes

- Une **liste** est une **collection** qui est **ordonnée, modifiable et** qui peuvent contenir plusieurs fois la même valeur ;
- Une liste est déclarée comme suit :

Liste= [el1, elt2,…eln]

- **Liste[i]:** permet d'accéder à l'élément de la liste se trouvant à la i^{ème} position

```
cetteListe= ["apple","banana","cherry“]        #déclarer la liste
print(cetteListe)                              #afficher la liste
print(cetteListe[1])                           #afficher le premier élément de la liste
print(cetteListe[-1])                          #afficher la valeur de la position -1 (cycle)
cetteListe1= ["apple","banana","cherry","orange","kiwi","melon","mango"]
print(cetteListe1[2:5])                        #afficher les valeurs de la position 2 jusqu'à 5 (5 non inclus)
```

Affichage:
['apple', 'banana', 'cherry']
banana
cherry
['cherry', 'orange', 'kiwi']

- **Liste[i]= val:** permet de changer la valeur de l'élément de la liste se trouvant à la position i

```
cetteListe = ["apple","banana","cherry"]
cetteListe[1] ="blackcurrant          #modifier la valeur de l'élément à la 1ère position
print(cetteListe) #affiche  ['apple', 'blackcurrant', 'cherry']
```

Affichage:
['apple', 'blackcurrant', 'cherry']

PARTIE 3

Listes

- Parcours une liste:

```
cetteListe = ["apple", "banana", "cherry"]
for x in cetteListe :    #parcourir des éléments de la liste
   print(x)              #afficher la valeur de x qui correspond à un élément de la liste
```

- Recherche d'un élément dans une liste:

```
cetteListe= ["apple", "banana", "cherry"]
if "apple" cetteListe :       #vérifier si une chaine est un élément de la liste
    print("Yes, 'apple' appartient à la liste")
```

- **Fonction len(Liste):** est une fonction permet de retourner la Longueur de la liste Liste

```
cetteListe = ["apple", "banana", "cherry"]
print(len(cetteListe))    # afficher la longueur d'une liste
```

Listes

- **Fonction append(élem):** est une fonction qui permet d'ajouter un élément élem à la fin de liste

```
cetteListe = ["apple", "banana", "cherry"]
cetteListe.append("orange")    # ajouter de l'élément ''orange'' à la fin de la liste
print(cetteListe)              # afficher ['apple', 'banana', 'cherry', 'orange']
```

- **Fonction insert(pos, élem)** est une fonction qui permet d'ajouter un élément élem à une position i de la liste

```
cetteListe = ["apple", "banana", "cherry"]
cetteListe.insert(1, "orange")    # insérer l'élément ''orange'' à la deuxième position
print(cetteListe)                 # afficher les éléments de la liste  ["apple", "orange« , "banana", "cherry " ]
```

- **Fonction pop()** est une fonction qui assure la suppression du dernier élément de la liste

```
cetteListe = ["apple", "banana", "cherry"]
cetteListe.pop()  #supprimer le dernier de liste qui est cherry
print(cetteListe) ) # afficher les éléments de la liste  ["apple", "banana "  ]
```

Listes

- **Fonction del(élem)** est une fonction qui permet de supprimer un élément particulier élem dans une liste

```
cetteListe= ["apple", "banana", "cherry"]
Del(cetteListe[0])    #supprimer l'élément se trouvant à la première position
print(cetteListe)  # affiche ["banana", "cherry"]
```

- **Fonction extend(Liste):** est une fonction qui permet de fusionner une liste

```
listA = ["a", "b" , "c"]
listB = [1, 8, 9]
listA.extend(listB)        #fusionner le deux Listes Liste1 et Liste2
print(listA)                #affiche ['a', 'b', 'c', 1,8, 9]
```

- **Fonction copy():** est une fonction qui permet de copier le contenu d'une liste dans une autre

```
cetteListe = ["apple", "banana", "cherry"]
malist= cetteListe.copy()
print(malist)    #affiche  ["apple", "banana", "cherry"]
```

Listes

- **Fonction clear()** est une fonction qui permet de supprimer tous les éléments d'une liste

```
cetteListe = ["apple", "banana", "cherry"]
cetteListe.clear()    #vider la liste
print(cetteListe)     #afficher une liste vide []
```

- **Fonction reverse()** est une fonction qui permet d'inverser une liste

```
cetteListe = ["apple", "banana", "cherry"]
cetteListe.reverse()  #inverser la liste thislist
print(cetteListe)       #afficher la liste inversée ['cherry', 'banana', 'apple']
```

CHAPITRE 2

Manipuler les données

1. Liste
2. **Collections**
3. Fichiers

- Le module collections contient des conteneurs de données spécialisés
- La liste des conteneurs est la suivante

Conteneur	Utilité
namedtuple	Une fonction permettant de créer une sous-classe de tuple avec des champs nommés
desque	Un conteneur ressemblant à une liste mais avec ajout et suppression rapide à chacun des bouts
ChainMap	Permet de linker entre eux plusieurs mappings ensemble pour les gérer comme tout
Counter	Permet de compter les occurrences d'objet hachable
OrderedDict	Une sous classe de dictionnaire permettant de savoir l'ordre des entrées
defaultdict	Une sous classe de dictionnaire permettant de spécifier une valeur par défaut dans le constructeur

Collections - namedtuple

- un tuple est une collection immuable de données souvent hétérogène.

```
t=(11,22)
print(t)   #affiche (11, 22)
print(t[0]) #affiche 11
print(t[1]) #affiche22
```

- **La classe namedtuple** du module collections permet d'ajouter des noms explicites à chaque élément d'un tuple pour rendre ces significations claires dans un programme Python

```
from collections import  namedtuple #importation de la classe namedtuple du module collections
Point = namedtuple('Point', ['x', 'y'])
p = Point(11, y=22)        # instanciation par position ou en utilisant le nom du champs
print(p[0] + p[1])         # les champs sont accessibles par leurs indexes affiche :33
print(p.x + p.y)           # les champs sont accessibles par nom affiche 33
print(p)                   # lisible dans un style nom=valeur affiche Point(x=11,y=22)
```

Collections - namedtuple

- La méthode **_asdict()** permet de convertir une instance en un dictionnaire.
- Appeler **p._asdict()** renvoie un dictionnaire mettant en correspondance les noms de chacun des deux champs de p avec leurs valeurs correspondantes.

```
from collections import namedtuple
Point = namedtuple ('Point',['x','y'])
p=Point(11,y=22)
print(p._asdict()) # renvoie un dictionnaire, affiche {'x': 11, 'y': 22}
```

- La fonction **_replace(key=args)** permet de retourner une nouvelle instance de notre tuple avec une valeur modifiée.

```
from collections import namedtuple
Point = namedtuple ('Point',['x','y'])
p=Point(11,y=22)
print(p._replace(x=4)) #changer la valeur de x x=4, affiche: Point(x=4, y=22)
```

Collections - namedtuple

- La fonction **_mytuple._fields** permet de récupérer les noms des champs de notre tuple. Elle est utile si on veut créer un nouveau tuple avec les champs d'un tuple existant.

```
from collections import namedtuple
Point = namedtuple ('Point',['x','y'])
print(Point._fields)  #retourne les noms de champs affiche: ('x','y')
Color =namedtuple ('Color', 'red green, blue')
Pixel= namedtuple ('Pixel', Point._fields + Color._fields)   #on crée un nouveau tuple avec les champs de point et de color
print(Pixel(11,22,128,266,0))    #affiche: Pixel(x=11, y=22, red=128, green=266, blue=0)
```

Collections - deque

- Les listes Python sont une séquence d'éléments ordonnés, mutable ou modifiable.
- Python peut ajouter des listes en temps constant mais l'insertion au début d'une liste peut être plus lente (le temps nécessaire augmente à mesure que la liste s'agrandit).

Exemple :

```
favorite_list = ["Sammy", "Jamie", "Mary"]
favorite_list.insert(0, "Alice") # insérer "Alice" au début de favorite_fish_list
print(favorite_list) #affiche ['Alice', 'Sammy', 'Jamie', 'Mary']
```

- **La classe deque** du module collections est un objet de type liste qui permet d'insérer des éléments au début ou à la fin d'une séquence avec une performance à temps constant (O(1)).
- O(1) performance signifie que le temps nécessaire pour ajouter un élément au début de de liste n'augmentera pas, même si cette liste a des milliers ou des millions d'éléments.

Collections - deque

- Les fonctions **append(x), appendleft(x): append** ajoute une seule valeur du côté droit du deque et **appendleft** du côté gauche

```
from collections import deque
favorite_deque = deque(["Sammy", "Jamie", "Mary"])
favorite_deque.appendleft("Alice") #ajout au début de favorite_fish_deque
favorite_deque.append("Bob")  #ajout à la fin de favorite_fish_deque
print(favorite_deque)  #affiche deque(['Alice', 'Sammy', 'Jamie', 'Mary', 'Bob'])
```

- Les fonctions **pop(), popleft() et clear(): pop()** et **pop(), popleft()** permettent de faire sortir un objet d'un deque et **clear()** le vide.

```
from collections import deque
favorite_deque = deque(['Alice', 'Sammy', 'Jamie', 'Mary', 'Bob'])
favorite_deque.pop()  #affiche Bob
favorite_deque.popleft()  #affiche: Alice
print(favorite_deque.clear())  #affiche None
```

Collections - ChainMap

- **La classe collections.ChainMap** permet de linker plusieurs mappings pour qu'ils soient gérés comme un seul.
- **class collections.ChainMap(*maps)** cette fonction retourne une nouvelle ChainMap. Si il n'y a pas de maps spécifiés en paramètres la ChainMap sera vide.

Exemple :

```
from collections import ChainMap
x = {'a': 1, 'b': 2}
y = {'b': 10, 'c': 11}
z = ChainMap(y, x)
 for k, v in z.items(): #parcourir les éléments de z
     print(k, v)
#affiche:
#a 1
#b 10
C# 11
```

- Dans l'exemple précédent on remarque que la clé b a pris la valeur 10 et pas 2 car y est passé avant x dans le constructeur de ChainMap.

Collections - Counter

- La classe **collections.Counter** est une sous-classe de dict. qui permet de compter des objets hachable.
- En fait c'est un dictionnaire avec comme clé les éléments et comme valeurs leur nombre.
- class collections.Counter([iterable-or-mapping]) ceci retourne un Counter. L'argument permet de spécifier ce que l'on veut mettre dedans et qui doit être compté.

Exemple :

```
from collections import Counter
c = Counter()        # compteur vide
print(c)             # affiche: Counter()
c = Counter('gallahad')  #compteur avec un iterable
print(c)   #affiche Counter({'a': 3, 'l': 2, 'g': 1, 'h': 1, 'd': 1})
c = Counter({'red': 4, 'blue': 2})     # un compteur avec un mapping
print(c)   #affcihe: Counter({'red': 4, 'blue': 2})
 c = Counter(cats=4, dogs=8)          #un compteur avec key=valeur
print(c)        #affcihe: Counter({'dogs': 8, 'cats': 4})
```

Collections - Counter

- Si on demande une valeur n'étant pas dans notre liste il retourne 0 et non pas KeyError

```
from collections import Counter
c = Counter(['eggs', 'ham'])
 c['bacon’]  # clé inconnue, affiche: 0
```

- La fonction **elements():** retourne une liste de tous les éléments du compteur :

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
sorted(c.elements())  #affiche ['a', 'a', 'a', 'a', 'b', 'b']
```

- La fonction **most_common([n]):** retourne les n éléments les plus présents dans le compteur

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
print(Counter('abracadabra').most_common(3))   #affiche [('a', 5), ('b', 2), ('r', 2)]
```

Collections - Counter

- La fonction **substract([iterable or mapping])** : permet de soustraire des éléments d'un compteur (mais pas de les supprimer)

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
print(c.subtract(d)) #affcihe Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Collections - OrdredDict

- Les **collections.OrderedDict** sont comme les dict. mais ils se rappellent l'ordre d'entrée des valeurs. Si on itère dessus les données seront retournées dans l'ordre d'ajout dans notre dict.
- La fonction **popitem(last=True) :** fait sortir une paire clé valeur de notre dictionnaire et si l'argument last est a 'True' alors les pairs seront retournées en LIFO sinon ce sera en FIFO.
- La fonction **move_to_end(key, last=True) :** permet de déplacer une clé à la fin de notre dictionnaire si last est à True sinon au début de notre dict.

Exemple :

```
from collections import OrderedDict
d=OrderedDict()
d['a']='1' #remplir d
d['b']='2'
d['c']='3'
d['d']='4'

d.move_to_end('b')
print(d) #affiche OrderedDict([('a', '1'), ('c', '3'), ('d', '4'), ('b', '2')])
d.move_to_end('b',last=False)
print(d) #affiche OrderedDict([('b', '2'), ('a', '1'), ('c', '3'), ('d', '4')])
print(d.popitem(True)) #affiche ('d', '4')
print(d) #affiche OrderedDict([('b', '2'), ('a', '1'), ('c', '3')])
```

Collections - defaultdict

- **defaultdict** du module de collections qui permet de rassembler les informations dans les dictionnaires de manière rapide et concise.
- **defaultdict** ne soulève jamais une **KeyError**.
- Si une clé n'est pas présente, **defaultdict** se contente d'insérer et de renvoyer une valeur de remplacement à la place.
- **defaultdict** se comporte différemment d'un dictionnaire ordinaire. Au lieu de soulever une KeyError sur une clé manquante, defaultdict appelle la valeur de remplacement sans argument pour créer un nouvel objet. Dans l'exemple ci-dessous, list() pour créer une liste vide.

```
from collections import defaultdict
my_defaultdict =defaultdict(list)
print(my_defaultdict["missing"]) #affcihe []
```

Collections - defaultdict

```
from collections import defaultdict
def default_message():
   return "key is not there"

defaultdcit_obj= defaultdict(default_message)
defaultdcit_obj["key1"]="value1"
defaultdcit_obj["key2"]="value2 »
print(defaultdcit_obj["key1"]) #affiche: value1
print(defaultdcit_obj["key2"]) #affiche: value2
print(defaultdcit_obj["key3"]) #affiche: key is not there
```

Remplissage d'un defaultdict

Affichage des éléments d'un defaultdict

Si la clé n'existe pas alors appeler la fonction default_message

Utilisation de la fonction lambda

```
from collections import defaultdict
defaultdcit_obj= defaultdict(lambda: "key is missing") #déclaration d'une fonction lambda pour afficher un message
defaultdcit_obj["key1"]="value1"
defaultdcit_obj["key2"]="value2"
print(defaultdcit_obj["key1"]) #affiche: value1
print(defaultdcit_obj["key2"]) #affiche: value2
print(defaultdcit_obj["key3"]) #appel de la fonction lambda, affiche: key is missing
```

PARTIE 3

Collections - defaultdict

- Lorsque la classe int est fournie comme fonction par défaut, la valeur par défaut retournée est zéro.

```
from collections import defaultdict
defaultdcit_obj =defaultdict(int)
defaultdcit_obj["key1"]="value1"
defaultdcit_obj["key2"]="value2"
print(defaultdcit_obj["key1"]) #affiche: value1
print(defaultdcit_obj["key2"]) #affiche: value2
print(defaultdcit_obj["key3"]) #affiche: 0
```

- Nous pouvons également utiliser l'objet Set comme objet par défaut

```
from collections import defaultdict
defaultdcit_obj =defaultdict(set)
defaultdcit_obj["key1"]="value1"
defaultdcit_obj["key2"]="value2"
print(defaultdcit_obj["key1"]) #affiche: value1
print(defaultdcit_obj["key2"]) #affiche: value2
print(defaultdcit_obj["key3"]) #affiche: set()
```

CHAPITRE 2

Manipuler les données

1. Liste
2. Collections
3. **Fichiers**

Utilisation des fichiers

- **Python a plusieurs fonctions pour créer, lire, mettre à jour et supprimer des fichiers texte.**
- La fonction clé pour travailler avec des fichiers en Python est open().
  - Elle prend deux paramètres; **nom de fichier et mode.**
  - Il existe quatre méthodes (modes) différentes pour ouvrir un fichier:
    - "r" -Lecture -Par défaut. Ouvre un fichier en lecture, erreur si le fichier n'existe pas
    - "a" -Ajouter -Ouvre un fichier à ajouter, crée le fichier s'il n'existe pas
    - "w" -Écrire -Ouvre un fichier pour l'écriture, crée le fichier s'il n'existe pas
    - "x" -Créer -Crée le fichier spécifié, renvoie une erreur si le fichier existe

- **Ouvrir et lire un fichier**
  - Pour ouvrir le fichier, utilisez la fonction open() intégrée
  - La fonction **open()** renvoie un objet fichier, qui a une méthode **read()** pour lire le contenu du fichier

```
f =open("demofile.txt","r")
print(f.read())
```

- **Renvoyez les 5 premiers caractères du fichier :**

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Utilisation des fichiers

- **Vous pouvez renvoyer une ligne en utilisant la méthode readline ():**

```
f =open("demofile.txt","r")
print(f.readline())
```

- **Ecrire dans un fichier**
  - Pour écrire dans un fichier existant, vous devez ajouter un paramètre à la fonction **open():**

```
f =open("demofile2.txt","a")   # ouvrir le fichier en mode ajout c'est à dire il est possible d'ajouter du texte dedans
f.write("Le fichier contient plus de texte")  #ajouter la phrase "Le fichier contient plus de texte"
f.close()   #fermer le fichier

f =open("demofile3.txt","w")    #ouvrir le fichier en mode écriture en effaçant son ancien contenu
f.write("domage!! l'ancient contenu est supprimé")   #écrire la phrase domage!! l'ancient contenu est supprimé"
f.close() #fermer le fichier

f =open("demofile3.txt","r")
print(f.read()) # ouvrir et lire le fichier après l'ajout
```

Utilisation des fichiers

- **Fermer un fichier**
  - Il est recommandé de toujours fermer le fichier lorsque vous en avez terminé.

```
f =open("demofile.txt","r")   #ouvrir le fichier en mode lecture
print(f.readline())    #lire une line du fichier
f.close()  #fermer le fichier
```

- **Supprimer d'un fichier**
  - Pour supprimer un fichier, vous devez importer le module OS et exécuter sa fonction **os.remove ():**

```
import os  #importer la bibliothèque os
os.remove("demofile.txt")  #supprimer un fichier
```

Format CSV

- Il existe différents formats standards de stockage de données. Il est recommandé de favoriser ces formats car il existe déjà des modules Python permettant de simplifier leur utilisation.
- Le fichier **Comma-separated values (CSV)** est un format permettant de stocker des tableaux dans un fichier texte. Chaque ligne est représentée par une ligne de texte et chaque colonne est séparée par un **séparateur** (virgule, point-virgule …).
- Les champs texte peuvent également être délimités par des guillemets.
- Lorsqu'un champ contient lui-même des guillemets, ils sont doublés afin de ne pas être considérés comme début ou fin du champ.
- Si un champ contient un signe pouvant être utilisé comme séparateur de colonne (virgule, point-virgule …) ou comme séparateur de ligne, les guillemets sont donc obligatoires afin que ce signe ne soit pas confondu avec un séparateur.

- **Données sous la forme d'un tableau**

Nom	Prénom	Age
Dubois	Marie	29
Duval	Julien "Paul"	47
Jacquet	Bernard	51
Martin	Lucie;Clara	14

- **Données sous la forme d'un fichier CSV**

```
Nom;Prénom;Age
"Dubois";" Marie "; 29
"Duval";"Julien ""Paul""";47
Jacquet;Bernard;51
Martin;"Lucie;Clara";14
```

Format CSV

- Le module csv de Python permet de simplifier l'utilisation des fichiers CSV
- **Lecture d'un fichier CSV**
  - Pour lire un fichier CSV, il faut ouvrir un flux de lecture de fichier et ouvrir à partir de ce flux un lecteur CSV.

```
import csv #importer la bibliothèque csv
fichier = open("exempleCSV.csv", "r")
lecteurCSV = csv.reader(fichier,delimiter=";")   # Ouverture du lecteur CSV en lui fournissant le caractère séparateur (ici ";")
for ligne in lecteurCSV:          # parcourir  les lignes du fichier CSV
                print(ligne)      # afficher chaque ligne du fichier CSV
fichier.close()   #fermer le fichier CSV
```

Affichage:
['Nom ', 'Prenom', 'Age']
['Dubois', 'Marie', '29']
['Duval', 'Julien ', '47']
['Jacquet', 'Bernard', '51']
['Martin', 'Lucie', '14']

  - Il est également possible de lire les données et obtenir un dictionnaire par ligne contenant les données en utilisant **DictReader** au lieu de **reader**

```
import csv #importer la bibliothèque csv
fichier = open("noms.csv", "r")
lecteurCSV = csv.DictReader(fichier,delimiter=";")
for ligne in lecteurCSV:
                print(ligne)
fichier.close()
```

Affichage:
{'Nom ': 'Dubois', 'Prenom': 'Marie', 'Age': '29'}
{'Nom ': 'Duval', 'Prenom': 'Julien', 'Age': '47'}
{'Nom ': 'Jacquet', 'Prenom': 'Bernard', 'Age': '51'}
{'Nom ': 'Martin', 'Prenom': 'Lucie', 'Age': '14'}

Format CSV

- **Écriture dans un fichier CSV**
  - À l'instar de la lecture, on ouvre un flux d'écriture (fonction **writer()**) et on ouvre un écrivain CSV à partir de ce flux (fonction **writerow()**) :

```
import csv #importer la bibliothèque csv
fichier = open("annuaire.csv", "w") #ouvrir un fichier en mode écriture
ecrivainCSV = csv.writer(fichier,delimiter=";")   #ouvrie un flux d'écriture
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"])       #écrire une 1ère ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Dubois","Marie","0198546372"])   #écrire une 2ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Duval","Julien","0399741052"]) #écrire une 3ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"]) #écrire une 4ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Martin","Julie","0399731590"]) #écrire une 5ème ligne dans le fichier annuaire.csv
fichier.close() #Fermer le fichier
```

Génération du fichier annuaire.csv

Nom	Prénom	Téléphone
Dubois	Marie	0198546372
Duval	Julien	0399741052
Jacquet	Bernard	0200749685
Martin	Julie	0399731590

Format CSV

- Il est également possible d'écrire le fichier en fournissant un dictionnaire par ligne à condition que chaque dictionnaire possède les mêmes clés.
- La fonction DictWriter() produit les lignes de sortie depuis des dictionnaires . Il faut également fournir la liste des clés des dictionnaires avec l'argument **fieldnames**

```
import csv #importer la bibliothèque csv
bonCommande = [{"produit":"cahier", "reference":"F452CP", "quantite":41, "prixUnitaire":1.6},
                {"produit":"stylo bleu", "reference":"D857BL", "quantite":18, "prixUnitaire":0.95},
                {"produit":"stylo noir", "reference":"D857NO", "quantite":18, "prixUnitaire":0.95},
                ]  #déclarer des dictionnaires
fichier = open("bon-commande.csv", "w") #ouvrir un fichier en écrire
ecrivainCSV = csv.DictWriter(fichier,delimiter=";",fieldnames=bonCommande[0].keys())
#produire des lignes de sortie depuis les dictionnaires,
# fieldnames contient les clés des dictionnaires
ecrivainCSV.writeheader()      # Ecrire la ligne d'en-tête avec le titre des colonnes
for ligne in bonCommande:    # Parcourir les dictionnaires
                ecrivainCSV.writerow(ligne)  #Ecrire une ligne dans le fichier
fichier.close()
```

Génération du fichier bon-commande.csv

```
reference;quantite;produit;prixUnitaire
F452CP;41;cahier;1.6
D857BL;18;stylo bleu;0.95
D857NO;18;stylo noir;0.95
```

Format JSON

- Le format JavaScript Object Notation (JSON) est issu de la notation des objets dans le langage JavaScript.
- Il s'agit aujourd'hui d'un format de données très répandu permettant de stocker des données sous une forme structurée.
- Il ne comporte que des associations **clés → valeurs** (à l'instar des dictionnaires), ainsi que des listes ordonnées de valeurs (comme les listes en Python).
- Une valeur peut être une autre association clés → valeurs, une liste de valeurs, un entier, un nombre réel, une chaîne de caractères, un booléen ou une valeur nulle.
- Sa syntaxe est similaire à celle des dictionnaires Python.

Exemple de fichier JSON : Définition d'un menu
il s'agit d'un objet composé de membres qui sont un attribut (Fichier) et un tableau (commandes)
qui contient d'autres objets: les lignes du menu. Une ligne de menu est identifiée par son titre et son action

Exemple de fichier JSON

```
{
 "menu": "Fichier",
 "commandes": [
    {
       "titre": "Nouveau",
       "action":"CreateDoc"
    },
    {
       "titre": "Ouvrir",
       "action": "OpenDoc"
    },
    {
       "titre": "Fermer",
       "action": "CloseDoc"
    }
 ]
}
```

Format JSON

- **Lire un fichier JSON**
  - La fonction loads (texteJSON) permet de décoder le texte JSON passé en argument et de le transformer en dictionnaire ou une liste.

```
import json                                  #importer la bibliothèque json
fichier = open ( "exemple.json", "r" ) #ouvrir le fichier exemple.json en lecture
x=json.loads(fichier.read( ) )             #décoder le texte et le transformer en dictionnaire.
print(x)
```

Affichage:
{'menu': 'Fichier', 'commandes': [{'titre': 'Nouveau', 'action': 'CreateDoc'}, {'titre': 'Ouvrir', 'action': 'OpenDoc'}, {'titre': 'Fermer', 'action': 'CloseDoc'}]}

- **Écrire un fichier JSON**
  - la fonction **dumps(variable, sort_keys=False)** transforme un dictionnaire ou une liste en texte JSON en fournissant en argument la variable à transformer.

```
import json          #importer la bibliothèque json
quantiteFournitures= {" cahiers":134, "stylos":{"rouge":41,"bleu":74},"gommes":85}   #déclarer un dictionnaire
fichier=open ("quantiteFournitures.json","w")  #ouvrir un fichier en écriture
fichier.write(json.dumps(quantiteFournitures))  #transformer le dictionnaire en texte json
fichier.close() #fermer le fichier
```

PARTIE 3