Chapitre 8 - Structures de données

Objectifs

- presenter les structures *liste, tuple et dictionnaire*
- distinguer les données *homogènes et hétérogènes*
- utiliser des *listes de listes*
- représenter des tableaux et des *matrices*
- stocker des données dans des *fichiers*

Définition
Une **structure de données** est une valeur composée qui regroupe plusieurs valeurs. Elle dispose d'une organisation permettant d'accéder à chacune de ses valeurs.

Le liste en est un exemple. Elle

- est structuré par son **ordre**
- est accédé par ses **indices** (0, 1, ...)
- est modifiable (**mutable**)

Les listes

Une liste est un ensemble d'objets qui

- sont délimités par des crochets `[]`
- sont séparés par des virgules `,`
- peuvent être de type différents (entier, texte, logique, ...)

```
In [1]: a = [1, 'abc', True, [1, 2]]
```

Fonctions

Voici deux fonctions `len` et `del` qui s'appliquent à des listes.

La fonction `len` permet de trouver la longeur de la liste.

In [2]:
```
len(a)
```

Out[2]:
```
4
```

La fonction `del` permet de supprimer un élément d'une liste.

In [3]:
```
del(a[2])
```

L'élément avec l'indice 2 a été supprimé.

In [4]:
```
a
```

Out[4]:
```
[1, 'abc', [1, 2]]
```

Méthodes

Les méthodes sont comme des fonctions, mais au lieu de passer la liste comme paramètre, la fonction est attaché à la liste par un point `.`

La méthode `a.append(x)` ajoute un élément `x` à la fin de la liste `a`.

In [5]:
```
a = [1, 'abc']
a.append(2)
a
```

Out[5]:
```
[1, 'abc', 2]
```

C'est souvent utilisé pour construire une liste. Dans ce cas on commence avec une liste vide `[]` et on ajoute un élément `x` à chaque itération avec la methode `append(x)`.
L'exemple suivant demande une entrée et ajoute un nouveau texte à la liste à l'aide d'une boucle.

In [6]:
```
a = []
x = input()
while x != '':
    a.append(x)
    x = input()
a
```

```
123
456
789
```

Out[6]:
```
['123', '456', '789']
```

La méthode `insert(i, x)` permet d'insérer un élément `x` dans une liste à l'indice `i` . Toutes les éléments qui suivent sont décalés.

In [7]:
```
a = [1, 'abc']
a.insert(1, True)
a
```

Out[7]:
```
[1, True, 'abc']
```

Ceci est différent de l'affectation qui remplace un élément.

In [8]:
```
a[1] = 123
a
```

Out[8]:
```
[1, 123, 'abc']
```

La méthode `remove(x)` enlève un élément `x` d'une liste. Elle enlève seulement le premier élément trouvé, mais donne une erreur si l'élément ne fait pas partie de la liste.

In [9]:
```
a = [1, 2, 3, 4, 2, 1]
a.remove(1)
a
```

Out[9]:
```
[2, 3, 4, 2, 1]
```

In [10]:
```
a.remove(1)
a
```

Out[10]:
```
[2, 3, 4, 2]
```

In [11]:
```
#a.remove(1)
#ValueError: list.remove(x): x not in list
```

La méthode `pop(i)` enlève l'élément i de la liste. Si aucun argument n'est donné, c'est le dernier argument qui est enlevé.

In [12]:
```
a = [1, 12, 32, 'abc']
print(a.pop(2))
a
```

```
32
```

Out[12]:
```
[1, 12, 'abc']
```

Résumé

- `a.append(x)` ajoute l'élément `x` à la fin de la liste `a`.
- `a.insert(i, x)` ajoute l'élément `x` à position `i` de la liste `a`.
- `a.remove(x)` enlève l'élément `x` de la liste `a` (et donne une erreur s'il n'existe pas).
- `a.pop(i)` enlève l'élément à la position `i` de la liste `a` et le retourne comme valeur.

Les tuples

Le tuple est aussi un ensemble d'objets. Contrairement à la liste il n'est pas modifiable. Une fois construit, le tuple ne change plus.

Un tuple est un ensemble d'objets qui:

- sont délimités par des parenthèses `()`
- sont séparés par des virgules `,`
- peuvent être de type différent (entier, chaine, booléen, etc.)
- peuvent être accédés par un indices entre crochets `[i]`

In [13]:
```
tup = (1, 'abc', True)
tup
```

Out[13]:
```
(1, 'abc', True)
```

On peut accéder à une élément en utilisant son indice.

In [14]:
```
tup[1]
```

Out[14]:
```
'abc'
```

On peut transformer une liste en tuple avec la fonction `tuple`.

In [15]:
```
tuple([2, 34, 678])
```

Out[15]:
```
(2, 34, 678)
```

Par contre faire une affectation à un élément de tuple donne une erreur.

In [16]:
```
# tup[1] = 123
# TypeError: 'tuple' object does not support item assignment
```

Avec `for` on peut itérer sur un tuple.

In [17]:
```
for x in tup:
    print(x)
```

```
1
abc
True
```

La fonction `len` retourne la longueur d'un tuple.

In [18]:
```
len(tup)
```

Out[18]: 3

La fonction `list` transforme un tuple en list.

In [19]:
```
list(tup)
```

Out[19]: `[1, 'abc', True]`

Pourquoi on a besoin des tuples ? Ne pourrait-on pas juste utiliser des listes ?

Oui, partout ou on peut utiliser un tuple, on aurait pu utiliser une liste. Une liste est plus puissante et a plus de méthodes. Mais on utilise un tuple:

- pour s'assurer que les **données restent constantes** et ne puissent pas être modifiées
- pour rendre le programme **plus rapide et plus petit** (les listes étant plus complexes)

Les dictionnaires

Un dictionnaire est un ensemble de **paires clé-valeur** ou la clé et la valeur peuvent être n'importe quel type, ou presque.

Un dictionnaire:

- est une **ensemble de paires** `key:value`
- délimité par des accolades `{}`
- séparé par des virgules `,`
- dont les paires **clé:valeurs** sont séparés par un deux-point `:`

Par exemple on peut associer un entier à une chaine ou une chaîne à un entier:

In [20]:
```
en = {1:'one', 2:'two', 3:'three'}
it = {'uno:':1, 'due':1, 'tre':3}
```

```
In [21]: en[2]
```

```
Out[21]: 'two'
```

```
In [22]: it['tre']
```

```
Out[22]: 3
```

Comme dans un vrai dictionnaire, on peut aussi associer deux chaînes.

```
In [23]: en_it = {'one':'uno', 'two':'due', 'three':'tre'}
```

Pour accéder à une valeur, on utilisé sa clé comme indice.

```
In [24]: en_it['two']
```

```
Out[24]: 'due'
```

```
In [25]: auto = {'immatrication':'SL-98-FG', 'marque':'Peugot', 'model':'306
         '}
```

On peut ajouter des paires clé-valeur à un dictionnaire par une affectation utilisant une nouvelle clé comme indice.

```
In [26]: auto['annee'] = 2016
         auto
```

```
Out[26]: {'immatrication': 'SL-98-FG',
          'marque': 'Peugot',
          'model': '306',
          'annee': 2016}
```

Parcourir un dictionnaire

La méthode `keys()` permet d'itérer sur les clés.

```
In [27]: for k in auto.keys():
             print(k)
```

```
immatrication
marque
model
annee
```

La méthode `values()` permet d'itérer sur les valeurs.

In [28]:
```
for v in auto.values():
    print(v)
```

```
SL-98-FG
Peugot
306
2016
```

La méthode `items()` permet d'itérer sur des paires clé-valeur.

In [29]:
```
for (k, v) in auto.items():
    print(k, '<--->', v)
```

```
immatrication <---> SL-98-FG
marque <---> Peugot
model <---> 306
annee <---> 2016
```

Itérer sur un dictionnaire tout court donne ses clés.

In [30]:
```
for k in auto:
    print(k)
```

```
immatrication
marque
model
annee
```

Exemple d'utilisation

Un dictionnaire peut servir à calculer la fréquence des lettres. Un dictionnaire est idéale pour cette tâche, car on a pas besoin de créer une liste pour toutes les lettres au début. A chaque fois une lettre est rencontrée pour la première fois, une nouvelle entrée pour cette lettre est ajouté au dictionnaire et la fréquence pour cette lettre est 1. On crée alors une nouvelle entrée dans le dictionnaire. Si cette même lettre est de nouveau rencontré, on increment le compteur.
La même méthode peut être e pour compter la fréquence des mots dans un ouvrage.

In [31]:
```
s = 'hello beautiful world'
freq = {}
for c in s:
    if c not in freq:
        freq[c] = 1
    else:
        freq[c] += 1

freq
```

Out[31]:
```
{'h': 1,
 'e': 2,
 'l': 4,
 'o': 2,
 ' ': 2,
 'b': 1,
 'a': 1,
 'u': 2,
 't': 1,
 'i': 1,
 'f': 1,
 'w': 1,
 'r': 1,
 'd': 1}
```

La fonction `list` permet de transformer un dictionnaire en liste de clés:

In [32]:
```
list(freq.keys())
```

Out[32]:
```
['h', 'e', 'l', 'o', ' ', 'b', 'a', 'u', 't', 'i', 'f', 'w', 'r',
'd']
```

En liste de valeurs:

In [33]:
```
list(freq.values())
```

Out[33]:
```
[1, 2, 4, 2, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1]
```

Ou en liste de tuples `(key, value)` :

In [34]:
```
list(freq.items())
```

Out[34]:
```
[('h', 1),
 ('e', 2),
 ('l', 4),
 ('o', 2),
 (' ', 2),
 ('b', 1),
 ('a', 1),
 ('u', 2),
 ('t', 1),
 ('i', 1),
 ('f', 1),
 ('w', 1),
 ('r', 1),
 ('d', 1)]
```

Terminologie
Des structures qui permettent l'accès à des valeurs au moyen de clés existent dans de nombreux langages.

On les appelles aussi

- tableau **associatif**
- table de hachage
- et en anglais: **hash, map, hashmap**

Données homogènes, hétérogènes

Les listes suivantes contiennent des éléments homogènes (entiers, chaînes).

In [35]:
```
a = [123, 234, 345]
```

In [36]:
```
b = ['jim', 'tom', 'pat']
```

Une liste peux aussi contenir des données hétérogènes:

In [37]:
```
c = ['alan', 'rochat', 17, True, '1M3', 2003]
```

On peut parcourir les éléments dans une boucle et afficher leur type:

In [38]:
```
for x in c:
    print(type(x), x)
```

```
<class 'str'> alan
<class 'str'> rochat
<class 'int'> 17
<class 'bool'> True
<class 'str'> 1M3
<class 'int'> 2003
```

On peut créer une liste de listes. Une simple indexation retourne alors une liste. Une double indexation retourne alors une élément de la sous-liste.

In [39]:
```
a = [[1], [2, 3], [4, 5, 6]]
print(a[2])
print(a[2][1])
```

```
[4, 5, 6]
5
```

Liste de tuples.

On l'utilse typiquement pour une liste de coordonnées. On peut acceder par simple et double indexation.

In [40]:
```
coord = [(1, 1), (5, 1), (6, 7), (1, 5)]
print(coord[2])
print(coord[2][0])
```

```
(6, 7)
6
```

On peut également effacer un élément et en insérer un.

In [41]:
```
coord.insert(2, (3, 3))
coord.pop(4)
coord
```

Out[41]:
```
[(1, 1), (5, 1), (3, 3), (6, 7)]
```

Exemple de gestion de stock

Définissions un stock d'articles d'informatique à l'aide d'une liste de dictionnaires:

In [42]:
```
stock = [{'name': 'printer', 'nbr': 4},
         {'name': 'mouse', 'nbr': 12},
         {'name': 'keyboard', 'nbr': 34}]
```

Pour ajouter un une nouvelle catégorie, nous pouvons utiliser la fonction `append` .

In [43]:
```
stock.append({'name': 'memory', 'nbr':0})
stock
```

Out[43]:
```
[{'name': 'printer', 'nbr': 4},
 {'name': 'mouse', 'nbr': 12},
 {'name': 'keyboard', 'nbr': 34},
 {'name': 'memory', 'nbr': 0}]
```

Trouver un élément dans une liste

Le défi ici est de trouver dans une liste **stock**, un dictionnaire qui possède un élément spécifique **name**. Pour ce type de recherche la démarche typique est:

- initialiser une balise booléenne (found) à `False`
- initialiser un élément (item) à vide
- parcourir la boucle (`for x in stock`)
- verifier un critère (==)
- mettre la balise à `True` si l'objet et trouvé
- mémoriser cet objet (`item = x`)

In [44]:
```
name = 'mouse'
#name = 'memory'
found = False
item = {}
for x in stock:
    if x['name'] == name:
        found = True
        item = x

found, item
```

Out[44]:
```
(True, {'name': 'mouse', 'nbr': 12})
```

Nous pouvons aussi définir une fonction qui

- parcourt la liste
- retourne l'élement quand il le trouve (return x)
- retourne un dictionnaire vide s'il ne le retrouve pas (return {})

In [45]:
```
def find_category(stock, name):
    for x in stock:
        if x['name'] == name:
            return x
    return {}
```

In [46]:
```
find_category(stock, 'printer')
```

Out[46]:
```
{'name': 'printer', 'nbr': 4}
```

In [47]:
```
find_category(stock, 'cpu')
```

Out[47]:
```
{}
```

Supprimer une catégorie

Pour supprimer une catégorie, on veut s'assurer que

- la catégorie existe, ensuite
- son nombre est zéro

Si ces conditions ne sont pas remplis, un message d'erreur est affiché.

In [48]:
```
def remove_category(stock, name):
    cat = find_category(stock, name)

    if cat == {}:
        print('category', name, 'does not exist')
    elif cat['nbr'] > 0:
        print('category', name, 'has more than 0 items')
    else:
        stock.remove(cat)
        print('category', name, 'has been removed')
```

Testons maintenant les trois possibilités:

In [49]:
```
remove_category(stock, 'screen')
```

```
category screen does not exist
```

In [50]:
```
remove_category(stock, 'keyboard')
```

```
category keyboard has more than 0 items
```

In [51]:
```
remove_category(stock, 'memory')
```

```
category memory has been removed
```

Changer le stock

Faisons maintenant une fonction pour changer le stock. De nouveau nous vérifions que

- la catégorie existe
- la nouvelle quantité n'est pas négative

In [52]:
```
def variation(stock, name, qty):
    cat = find_category(stock, name)

    if cat == {}:
        print('category', name, 'does not exist')
    elif cat['nbr'] + qty < 0:
        print('quantity will be negative')
    else:
        cat['nbr'] += qty
        print('quantity changed')
```

Verifions maintenant les 3 différants cas:

In [53]:
```
variation(stock, 'screen', -10)
```

```
category screen does not exist
```

In [54]:
```
variation(stock, 'printer', -10)
```

```
quantity will be negative
```

In [55]:
```
variation(stock, 'mouse', -10)
```

```
quantity changed
```

Représentation des tableaux et des matrices

Dans la vie courante, on souvent besoin de structures de données régies par deux critères différents, représentées sous forme de tableau. Par exemple, on peut représenter de cette façon le chiffre des ventes des voitures électriques en France par modèle et par mois.

modèle	janvier	février	mars	avril
Renault Zoé	733	1186	1370	990
Nissan Leaf	173	286	717	491
Peugot iOn	161	109	103	86

Ajouter la deuxième table de la page 147 du livere en Markdown, en utilisant l'outil online sur le site https://www.tablesgenerator.com/markdown_tables# (https://www.tablesgenerator.com/markdown_tables#)

Listes et dictionnaires comme tableaux à une seule ligne

On peut interpréter une liste comme tableau à une seule ligne.

indice	0	1	2
élément	5	12	21

In [56]:
```
a = [5, 12, 2]
```

Exercice
Créer un script qui additionne deux vecteurs a et b.

In [57]:
```
a = [1, 2, 3]
b = [10, 10, 20]
n = len(a)
c = []
for i in range(n):
    c.append(a[i] + b[i])
c
```

Out[57]:
```
[11, 12, 23]
```

Créer une fonction qui additionne deux vecteurs. La fonction donne un message d'erreur si les deux vecteurs ont une taille différente. Testez la fonction avec deux cas différents.

In [58]:
```
def vect_add(a, b):
    if len(a) != len(b):
        print('error: vectors have different length')
    else:
        result = []
        for i in range(len(a)):
            c = a[i] + b[i]
            result.append(c)
        return result
```

In [59]:
```
vect_add(a, [10, 10])
```

```
error: vectors have different length
```

In [60]:
```
vect_add(a, b)
```

Out[60]:
```
[11, 12, 23]
```

Exercice
Créer un script qui retourne la norme d'un vecteur a.
Pour déboguer imprimez toutes les résultats intermédiaires. Importez la fonction `sqrt` depuis le module `math`

In [61]:
```
from math import sqrt
a = [2, 3, 5]
acc = 0
for x in a:
    print(x, x ** 2)
    acc += x ** 2
print(acc)
print(sqrt(acc))
```

```
2 4
3 9
5 25
38
6.164414002968976
```

Transformez le script en fonction. Testez-le avec deux cas différents.

In [62]:
```
def vect_norm(a):
    acc = 0
    for x in a:
        acc += x ** 2
    return sqrt(acc)
```

In [63]:
```
vect_norm([3, 4])
```

Out[63]:
```
5.0
```

In [64]:
```
vect_norm([3, 4, 5])
```

Out[64]:
```
7.0710678118654755
```

Le tableau à plusieurs lignes: la matrice 2D

Prenons tout d'abord l'exemple d'une matrice à deux dimensions.

16	3	2	13
5	10	11	8
9	6	7	12

Pour representer cette matrice il suffit de placer 3 listes à 4 éléments dans une nouvelle liste:

In [65]:
```
mat = [[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12]]
mat
```

Out[65]:
```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12]]
```

Pour accéder à une ligne nous utilisons 1 indice.
Pour accéder à un élément nous utilisons 2 indices.

In [66]:
```
print(mat[1])
print(mat[1][2])
```

```
[5, 10, 11, 8]
11
```

Afficher une matrice

Affiche les lignes de la matrice sur des lignes séparées.

In [67]:
```
for line in mat:
    print(line)
```

```
[16, 3, 2, 13]
[5, 10, 11, 8]
[9, 6, 7, 12]
```

Affiche les éléments de chaque ligne sans les crochets et les virgules (Utiliser un `*` devant une liste pour passer ses éléments séparément à la fonction).

In [68]:
```
for line in mat:
    print(*line)
```

```
16 3 2 13
5 10 11 8
9 6 7 12
```

Affiche les éléments de chaque ligne aligné avec un tabulateur. (Utiliser l'option `sep='\t'`)

In [69]:
```
for line in mat:
    print(*line, sep='\t')
```

```
16      3       2       13
5       10      11      8
9       6       7       12
```

Crée une fonction qui affiche les éléments d'une matrice séparées par des tabulateurs.

In [70]:
```
def mat_print(mat):
    for line in mat:
        print(*line, sep='\t')
```

In [71]:
```
mat_print(mat)
```

```
16	3	2	13
5	10	11	8
9	6	7	12
```

Transposer une matrice

Créer un script qui crée une nouvelle matrice avec les éléments transposé.

- Commencez avec une matrice vide.
- Ajoutez ensuite des matrice vides pour chaque colonnne.
- Affichez toutes les étapes intermédiaires.

In [72]:
```
mat2 = []
print(mat2)
n = len(mat[0])

for i in range(n):
    mat2.append([])
print(mat2)

for line in mat:
    for i in range(n):
        mat2[i].append(line[i])
    print(mat2)
```

```
[]
[[], [], [], []]
[[16], [3], [2], [13]]
[[16, 5], [3, 10], [2, 11], [13, 8]]
[[16, 5, 9], [3, 10, 6], [2, 11, 7], [13, 8, 12]]
```

In [73]:
```
mat_print(mat2)
```

```
16	5	9
3	10	6
2	11	7
13	8	12
```

Représentation interne

La mémoire de l'ordinateur est linéaire: c'est une succéssion de mots-mémoires. Chaque emplacement peut accepter une séqunce bineaire et il est identifié par une adresse. Il n'est pas possible de créer directement une structure de données 2D ou 3D.

Les matrices `mat` et `mat2` sont représenté par une séquence linéaire en mémoire.

In [74]:
```
print('addr:', end='\t')
for i in range(12):
    print(i, end='\t')

print('\ndata:', end='\t')
for line in mat:
    print(*line, sep='\t', end='\t')
```

```
addr:	0	1	2	3	4	5	6	7	
8	9	10	11	
data:	16	3	2	13	5	10	11	8	
9	6	7	12	
```

In [75]:
```
print('\ndata:', end='\t')
for line in mat2:
    print(*line, sep='\t', end='\t')
```

```
data:	16	5	9	3	10	6	2	11	
7	13	8	12	
```

Les files d'attente et les piles

Entrée et sortie libre

Ce premier exemple est celui d'un parking ou les usagers viennent et partent quand ils souhaitent.

In [76]:
```
parking = []
parking.append(123)
parking.append(956)
parking.append(643)
parking
```

Out[76]:
```
[123, 956, 643]
```

In [77]:
```
parking.remove(956)
parking
```

Out[77]:
```
[123, 643]
```

File d'attente: premier arrivé, premier parti (FIFO)

Dans la file d'attente d'un fast-food drive-in la situation est un peu différente. C'est le premier arrivé qui peut partir en premier. Une telle liste est désignée en anglais par l'expression *first in - first out* ou aprégée par l'acronyme **FIFO**.

In [78]:
```
parking = []
parking.append(123)
parking.append(956)
parking.append(643)
parking
```

Out[78]:
```
[123, 956, 643]
```

In [79]:
```
parking.pop(0)
```

Out[79]:
```
123
```

In [80]:
```
parking.pop(0)
```

Out[80]:
```
956
```

La pile: dernier arrivé, premier partie (LIFO)

Une autre structure de donnée classique est la pile **LIFO** (*last in - first out*) dernier arrivé - premier parti. L'image que représente bien cette structure est celle d'une pile d'assiettes.

Une application typique est une calculatrice qui utilise la notation polonaise inverse.
https://fr.wikipedia.org/wiki/Notation_polonaise_inverse
(https://fr.wikipedia.org/wiki/Notation_polonaise_inverse)

Dans cette notation les operandes sont mis sur une pile. Quand un opérateur (`+, *`) est rencontré, les deux derniers eléments de la piles sont enlevé de la pile. Le résultat de l'opération est de nouveau mis sur la pile.

La méthode `pop()` sans argument enlève automatiquement le dernier élément de la pile.

In [81]:
```
print('Polish calculator')

stack = []
x = input()

while x != '':
    if x in '+*':
        a = stack.pop()
        b = stack.pop()

        if x == '+':
            c = a + b
        elif x == '*':
            c = a * b
        print(c)
        stack.append(c)

    else:
        stack.append(int(x))
    x = input()
```

```
Polish calculator
12
12
12
*
144
*
1728
```

Les fichiers

Les structures de données jusqu'à présent sont des structures de données en mémoire, créées par le programme. Elle disparaissent à la fin de l'exécution du programme. Les fichiers sont un moyen de sauvegarder les données sur disque, pour pouvoir les retrouver plus tard.

Il existe deux sortes de fichiers:

- des fichiers texte et
- des fichies binaires

Dans un **fichier texte** le nombre 12345 peut être écrit comme *séquence de 5 caractères*:

In [82]:
```
'12345'
```

Out[82]:
```
'12345'
```

Dans un **fichier binaire** ce même nombre est écrit comme une *valeur numérique sous forme de séquence binaire*:

In [83]:
```
print('{0:b}'.format(12345))
```

```
11000000111001
```

La fin de ligne

Les fichiers texte sont organisés en lignes. Une fin de ligne est notée avec le caractère spéciale `\n` . On peut avoir l'impression que c'est deux caractères, mais ce n'est pas le cas. La barre oblique inverse est une balise (escape character) pour signaler un caractère spéciale.

In [84]:
```
texte = 'un\ndeux'
```

In [85]:
```
texte
```

Out[85]:
```
'un\ndeux'
```

In [86]:
```
print(texte)
```

```
un
deux
```

Lors de de l'affichage par `print` le `\n` se transforme en fin de ligne. Lorsqu'on compte le nombre des caractères, le caractère `\n` compte comme un.

In [87]:
```
len(texte)
```

Out[87]:
```
7
```

Un fichier peut être ouvert en mode

- lecture
- écriture

Il existent deux variantes d'écriture:

- remplacement du fichier précédent
- ajout à un fichier existant

Ecriture dans un fichier

Pour écrire dans un fichier on doit appeler la fonction `open(file_name, mode)`.

- Le nom de fichier est une chaîne et comporte souvent une extension telle que `.txt`.
- Le mode indique la lecture (`r` read) ou l'écriture (`w` write)
- La fonction retourne une référence vers le fichier, souvent appelé `f`

La méthode `write(str)` ajoute la chaîne au fichier. Elle retourne le nombre de caractères ajoutés au fichier.

La méthode `close()` ferme le fichier et termine l'écriture ou lecture.

In [88]:
```
f = open('demo.txt', 'w')
f.write('hello')
f.write('world')
f.close()
```

Pour écrire chaque mot sur une ligne il faut ajouter un `\n`.

In [89]:
```
f = open('demo2.txt', 'w')
f.write('hello\n')
f.write('world\n')
f.close()
```

Quelques commandes magiques

La commande magique `%pwd` (print working directory) permet d'afficher le dossier actuel.

In [90]:
```
%pwd
```

Out[90]:
```
'/Users/raphael/OneDrive/nb/inf/08'
```

La commande magique `%ls` (list) permet d'afficher la liste des fichiers dans le dossier actuel.

In [91]:
```
%ls
```

```
08_structures.ipynb  city.txt             input.txt
08_structures.pdf    demo.txt             nombres.txt
cities.txt           demo2.txt            numbers.txt
```

La commande magique `%cat filename` permet d'afficher le contenu d'un fichier.

In [92]:
```
%cat demo.txt
```

```
helloworld
```

In [93]:
```
%cat demo2.txt
```

```
hello
world
```

Ecrire une liste dans un fichier

Le programme suivant écrit une liste dans un fichier.

In [94]:
```
cities = ['Paris', 'London', 'Tokyo']

f = open('cities.txt', 'w')
for x in cities:
    f.write(x + '\n')
f.close()
```

In [95]:
```
%cat cities.txt
```

```
Paris
London
Tokyo
```

Ecrire des entrées du clavier vers un fichier

Le programme suivant ajoute des entrées du clavier dans un fichier texte. Pour terminer la boucle il faut entrer la chaîne vide.

In [96]:
```
f = open('input.txt', 'w')
x = input()
while x != '':
    f.write(x + '\n')
    x = input()

f.close()
```

```
jim
tom
pat
```

In [97]:
```
%cat input.txt
```

```
jim
tom
pat
```

Lire un fichier existant

La lecture d'un fichier se fait en deux temps. Il faut d'abord ouvrir le fichier avec la fonction `open()`. Cette fonction comporte comme argument le nom de fichier, ainsi comme mode `r` (read). Une fois ouvert, la fonction `readline()` permet de lire une ligne. Une fois toutes les lignes lues, elle renvoie la chaîne vide. On configure ici la fonction `print()` pour ne pas ajouter un `\n` en fin de ligne.

In [98]:
```
f = open('cities.txt', 'r')
print(f.readline(), end='')
print(f.readline(), end='')
print(f.readline(), end='')
f.close()
```

```
Paris
London
Tokyo
```

On peut également utiliser une boucle pour parcourir un fichier ligne par ligne, avec la commande `for line in f:`

In [99]:
```
f = open('cities.txt', 'r')
for line in f:
    print(line, end='')
f.close()
```

```
Paris
London
Tokyo
```

Ecrire et lire une liste de nombres

Ecrivons d'abord une liste de nombres dans un fichier et vérifions le résultat.

In [100]:
```
f = open('numbers.txt', 'w')
f.write('1\n23\n456\n7890')
f.close()
%cat numbers.txt
```

```
1
23
456
7890
```

Nous lisons ligne par ligne, transformons la chaîne en entier avec `int` et ajoutons le nouveau élément à la liste avec `append`.

In [101]:
```
a = []
f = open('numbers.txt', 'r')
for line in f:
    a.append(int(line))
f.close()
a
```

Out[101]:
```
[1, 23, 456, 7890]
```

Lecture de toutes les lignes en une fois

On peut lire également tout le contenu du fichier d'un seul coup avec la méthoded `read()` .

In [102]:
```
f = open('cities.txt', 'r')
s = f.read()
print(s)
```

```
Paris
London
Tokyo
```