Définition

Les listes font partie de ce qu'on appelle les *données composites* (nous verrons plus tard les *tuples* et les *dictionnaires*). Elles permettent de regrouper de manière structurée des ensembles de valeurs. On les appelle *listes* en Python, ou bien *tableaux* de manière plus générale.

Notation : dans une liste, les éléments sont séparés par des **virgules**, et l'ensemble est délimité par des **crochets**.

In [1]:
```
ma_premiere_liste = [1, "ok", True]
```

Même si cela n'a ici un grand intérêt, les éléments d'une liste peuvent donc être de types différents : ici, nous avons successivement un entier (`int`), une chaine de caractères (`str`), et un booléen (`bool`).

Accès aux éléments d'une liste

On accède à un élément d'une liste en mettant entre crochets l'indice de l'élémént (qui commence à **zéro**).

In [2]:
```
famille = ["Bart", "Lisa", "Maggie"]
```

In [3]:
```
famille[0]
```

Out[3]:
```
'Bart'
```

Un indice qui dépasse la valeur `longueur de la liste -1` provoquera une erreur `list index out of range` .

In [4]:
```
famille[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-6e7a4ba7ec75> in <module>
----> 1 famille[3]

IndexError: list index out of range
```

Il est par contre possible d'utiliser des indices **négatifs** :

In [5]:
```
famille[-1]
```

Out[5]:
```
'Maggie'
```

Longueur d'une liste

La longueur d'une liste sera donnée par la fonction `len()`

In [6]:
```
len(famille)
```

Out[6]:
```
3
```

Attention : le dernier élément de la liste a donc l'indice `len(liste)-1`

Parcours des éléments d'une liste

In [7]:
```
for k in range(len(famille)):
    print(famille[k])
```

```
Bart
Lisa
Maggie
```

Remarque : nous avons déjà vu une méthode plus directe de parcours d'une liste :

In [8]:
```
for k in famille :  # <- beaucoup plus efficace !
    print(k)
```

```
Bart
Lisa
Maggie
```

Les éléments d'une liste sont MODIFIABLES.

C'est une différence fondamentale avec une autre structure (les *tuples*) que nous verrons plus tard.

In [9]:
```
famille[0] = "Milhouse"
```

In [10]:
```
famille
```

Out[10]:
```
['Milhouse', 'Lisa', 'Maggie']
```

On dit que les listes sont des objets **mutables**.

Ajout d'un élement à une liste : méthode append()

In [11]:
```
famille.append("Lisa")
```

In [12]:
```
famille
```

Out[12]:
```
['Milhouse', 'Lisa', 'Maggie', 'Lisa']
```

La méthode `append()` rajoute donc un élément **à la fin** de la liste.

Liste vide

Très souvent, la méthode `append()` ci-dessus est utilisée à partir d'une liste vide `[]` , à laquelle on rajoute peu à peu des éléments.

Exemple

Filtrons la liste m ci-dessous pour n'en garder que les éléments supérieurs à 20.

In [13]:
```
m = [45, 12, 15, 20, 18, 19, 23, 17, 12, 18]
p = []

for k in m :
    if k > 20:
        p.append(k)
```

Suppression d'un élément d'une liste ...

... par la méthode remove() :

In [16]:
```
famille.remove("Lisa")
```

In [17]:
```
famille
```

Out[17]:
```
['Milhouse', 'Maggie']
```

La méthode `remove()` va supprimer le premier élément de la liste (ce qui est problématique s'il y en a plusieurs) qui correspond à l'élément passé en argument.

In [18]:
```
maliste = [8, 4, 2, 4, 7]
maliste.remove(4)
```

In [19]:
```
maliste
```

Out[19]:
```
[8, 2, 4, 7]
```

In [20]:
```
maliste.remove(5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-a72d2de18dbe> in <module>
----> 1 maliste.remove(5)

ValueError: list.remove(x): x not in list
```

... par la fonction del() :

La fonction `del()` permet de supprimer un élément en donnant son indice.

In [21]:
```
maliste = [8, 4, 2, 5, 7]
del maliste[3]
```

In [22]:
```
maliste
```

Out[22]:
```
[8, 4, 2, 7]
```

In [ ]:
```

```

Exercice :

Trouvez le nombre qui est exactement à la même place dans la liste `a` et dans la liste `b` (les deux listes ont la même taille)

In [23]:
```
a = [8468, 4560, 3941, 3328, 7, 9910, 9208, 8400, 6502, 1076, 5921, 6720, 948, 956
1, 7391, 7745, 9007, 9707, 4370, 9636, 5265, 2638, 8919, 7814, 5142, 1060, 6971, 40
65, 4629, 4490, 2480, 9180, 5623, 6600, 1764, 9846, 7605, 8271, 4681, 2818, 832, 52
80, 3170, 8965, 4332, 3198, 9454, 2025, 2373, 4067]
b = [9093, 2559, 9664, 8075, 4525, 5847, 67, 8932, 5049, 5241, 5886, 1393, 9413, 88
72, 2560, 4636, 9004, 7586, 1461, 350, 2627, 2187, 7778, 8933, 351, 7097, 356, 411
0, 1393, 4864, 1088, 3904, 5623, 8040, 7273, 1114, 4394, 4108, 7123, 8001, 5715, 72
15, 7460, 5829, 9513, 1256, 4052, 1585, 1608, 3941]


for i in range(len(a)):
    if a[i] == b[i]:
        print(a[i], " à l'indice ", i)
```

```
5623  à l'indice  32
```

In [3]:
```
i
```

Out[3]:
```
2
```

Construction d'une liste d'éléments identiques

In [24]:
```
p = [0]*12
```

In [9]:
```
p
```

Out[9]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Cette instruction est très utile pour initialiser des listes de taille donnée.

Exemple

le code ci-dessous compte l'occurence de chaque lettre de l'alphabet dans un texte.

In [25]:
```
texte = "cet texte est prodigieusement ennuyeux"

def rang(lettre):
    return(ord(lettre)-97)

compt = [0]*26
for k in texte :
    if k != " " :
        compt[rang(k)] += 1
```

In [19]:
```
compt
```

Out[19]:
```
[0, 0, 1, 1, 9, 0, 1, 0, 2, 0, 0, 0, 1, 3, 1, 1, 0, 1, 2, 5, 3, 0, 0, 2, 1, 0]
```

Construction d'une liste (méthode dite *par compréhension*)

In [20]:
```
nombres = [k for k in range(10)]
```

In [21]:
```
nombres
```

Out[21]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Il est bien sûr possible d'agir sur le paramètre :

In [22]:
```
carres_parfaits = [k**2 for k in range(10)]
```

In [23]:
```
carres_parfaits
```

Out[23]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Plus subtil, on peut filtrer "à la volée" les éléments pour n'en garder que certains.

In [24]:
```
c = [n for n in carres_parfaits if n % 3 == 0]
```

In [25]:
```
c
```

Out[25]:
```
[0, 9, 36, 81]
```

Exercice

En une ligne, créez la liste `p` qui contient tous les éléments positifs de `m`.

In [26]:
```
m = [-3, -6, 4, 7, -2, 1, -3, 5]
p = [k for k in m if k > 0]
```

In [27]:
```
p
```

Out[27]:
```
[4, 7, 1, 5]
```

Un phénomène inquiétant : la copie de liste

In [31]:
```
a = 3
b = a
a = 5
```

In [37]:
```
id(a)
```

Out[37]:
```
1527898960
```

In [38]:
```
id(b)
```

Out[38]:
```
1527898928
```

```
In [33]: b
```

```
Out[33]: 3
```

```
In [40]: a = [4, 5, 7]
         b = a
         a[0] = 12
```

Selon toute vraisemblance, la liste `a` est maintenant égale à `[12, 5, 7]`. Vérifions-le :

```
In [41]: id(a)
```

```
Out[41]: 81446152
```

```
In [42]: id(b)
```

```
Out[42]: 81446152
```

```
In [29]: a
```

```
Out[29]: [12, 5, 7]
```

Tout est donc normal. Mais :

```
In [30]: b
```

```
Out[30]: [12, 5, 7]
```

`b` est lui **aussi** devenu égal à `[12, 5, 7]`, alors que la modification sur l'élément `a[0]` n'a été faite qu'**après** l'affectation `b = a`.
Les listes `a` et `b` sont en fait strictement et définitivement identiques, elles sont simplement deux dénominations différentes d'un même objet. On peut le vérifier en regardant l'emplacement-mémoire vers lequel pointent la variable `a` et la variable `b` :

```
In [ ]: id(a)
```

```
In [ ]: id(b)
```

Comment copier le contenu d'une liste vers une autre

Parmi plusieurs solutions, celle-ci est simple et efficace : la méthode `copy()`

```
In [1]: a = [3, 4, 9]
        b = a.copy()
        a[0] = 12
```

```
In [2]: a
```

```
Out[2]: [12, 4, 9]
```

In [3]:
```
b
```

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

On pourrait aussi écrire `b = list(a)` mais la méthode `copy()` a l'avantage d'exister aussi pour les *dictionnaires*, que nous verrons plus tard.

Tableaux à plusieurs dimensions : listes de listes

In [47]:
```
a = [[3, 5, 2],
     [7, 1, 4],
     [8, 6, 9]]
```

La liste `a` est composée de 3 éléments qui sont eux-même des listes de 3 éléments.

In [48]:
```
a[0]
```

Out[48]:
```
[3, 5, 2]
```

In [49]:
```
a[0][1]
```

Out[49]:
```
5
```

In [50]:
```
a[2][2]
```

Out[50]:
```
9
```

Exercice

Résolvez ce pydéfi (https://callicode.fr/pydefis/AlgoMat/txt)

Solution possible :

In [14]:
```
M=[[36, 19, 27, 36, 7, 10], [2, 18, 3, 33, 2, 21], [26, 27, 4, 22, 30, 31], [29, 3
6, 7, 20, 6, 30], [30, 6, 14, 23, 15, 13], [22, 10, 10, 35, 15, 22]]

def modif(k):
    return ( 11*k + 4 ) % 37

for k in range(23): #pour répéter 23 fois
    for i in range(6) :
        for j in range(6):
            M[i][j] = modif(M[i][j])
            # ou bien M[i][j] = ( 11*M[i][j] + 4 ) % 37

s = 0
for i in range(6) :
    for j in range(6):
        s = s + M[i][j]
```

In [13]:
```
s
```

Out[13]:
```
575
```

Pydéfi : Le sanglier d'Erymanthe

https://callicode.fr/pydefis/Herculito04Sanglier/txt (https://callicode.fr/pydefis/Herculito04Sanglier/txt)

In [11]:
```
M = [0, 50, 40, 100, 70, 90, 0]
s = 0
for k in range(7-1):
    if M[k] > M[k+1]:
        pierreLancees = (M[k]-M[k+1])//10+1
        s += pierreLancees
print(s)

# penser à faire le total des pierres lancées depuis le début
```

```
16
```

En une ligne (aucun intérêt à part pour épater la galerie...) :

In [5]:
```
M = [0, 50, 40, 100, 70, 90, 0]
k = sum([(M[i]-M[i+1])//10+1 for i in range(len(M)-1) if M[i]>M[i+1]])
```

In [6]:
```
k
```

Out[6]:
```
16
```