# Listes et chaînes de caractères

GCC ! – Prologin

2020

## Introduction aux listes

Les listes permettent de stocker plusieurs éléments.

En Python, on crée une liste vide (une liste sans élément) avec :

```
1 empty_list = []
```

Nommer ses variables en anglais est une bonne habitude à prendre, les mots-clés des langages de programmation étant le plus souvent en anglais. En plus, ça ne peut que faire progresser ton niveau de langue ! Ici, *empty_list* correspond à *liste_vide*.

Ça suffit pour le blabla, il est temps d'apprendre ! Créons une liste qu'on appelle L : `L = []`

**len(L)** retourne la longueur - *length* en anglais - de la liste L, correspondant au nombre d'éléments dans la liste.

Si on fait `len(L)` sur notre liste vide, on obtient 0 puisque notre liste ne contient pas d'éléments.

Ensuite, **L.append(x)** ajoute l'élément x à la fin de la liste L. On ajoute alors le caractère 'x' à la liste L avec `L.append('x')`. Maintenant, **len(L)** affiche 1 et `L = ['x']`.

Également, `L[i]` accède à l'élément à la place i dans la liste L. Ainsi, si on veut accéder à l'élément 'x', on écrit `L[0]` car 'x' se trouve à l'indice 0. En effet, la numérotation des indices dans une liste en Python commence à 0.

Donc si on essaye d'accéder à `L[1]`, on aura une erreur. Par contre, si on rajoute 'y' avec `L.append('y')`, la liste devient `['x', 'y']`. Il est alors possible d'accéder à 'y' en écrivant `L[1]`.

Par exemple :

```
1 L = ['a', 'b']
2 L.append('c')
3 print(L)
```

Le code ci-dessus affiche `['a', 'b', 'c']`. On a alors : `L[0] = 'a'`, `L[1] = 'b'`, et `L[2] = 'c'`. Aussi, `len(L) = 3`.

Il est également possible de créer des listes de nombres :

```
list_of_numbers = [1, 5, 9, 42, 13]
```

Ainsi que des listes avec des nombres et des caractères en même temps. Par exemple :

```
list_mix = ['a', 1, 26, 'z']
```

En Python, on peut aussi *concaténer* deux listes, c'est-à-dire les mettre l'une à la suite de l'autre pour n'en créer qu'une.

Pour ce faire, on utilise le signe `+`.

Par exemple, on crée deux listes L1 et L2, on les fusionne dans une troisième liste L3, puis on affiche L3 :

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = L1 + L2
print(L3)
```

Ce code affiche `[1, 2, 3, 4, 5, 6]`.

Imaginez maintenant que vous souhaitiez créer une liste remplie de 42, disons 100 fois le nombre 42 :

```
L = [42] * 100
print(L)
```

Cela affiche `[42, 42, 42, ..., 42, 42]`.

Une fois votre liste de nombres créée, vous aimeriez sûrement la trier. Pour cela, on utilise la méthode **sort()**. Par exemple :

```
L = [2, 9, 6, 4, 42, 13]
L.sort()
print(L)
```

Ce qui affichera `[2, 4, 6, 9, 13, 42]`.

### Testez votre code !

Testez votre code le plus régulièrement possible. C'est important pour savoir où se trouve le bug, car bug il y aura !



Avec ce sujet vous seront fournis plusieurs fichiers tests vous permettant de vérifier que votre code fonctionne.

Pour être certains que tout cela se passe dans les règles de l'art, il vous faudra respecter quelques règles :

— Créer un fichier dans lequel vous mettrez vos fonctions (*tp_listes.py* par exemple)
— Respecter les noms de fonctions précisés dans les énoncés suivants
— Exécuter le fichier test avec `./tests.py` suivi du nom de votre fichier dans le terminal ( `./tests.py tp_listes.py` par exemple)

### Exercice 1 : commençons par le commencement !

**But :** écrire la fonction `init_list(n)` qui stocke dans une liste les entiers de 0 inclus à n inclus, et la retourne.

**Exemple :** `init_list(5)` renvoie `[0, 1, 2, 3, 4, 5]` .

### Exercice 2 : À la recherche de k

**But :** écrire la fonction `is_in(k, L)` qui traverse la liste non triée L et retourne `True` si k est dans la liste, ou `False` autrement.

**Exemple :** `is_in(2, [1, 2, 3])` renvoie `True` (2 est dans la liste) et `is_in(5, [1, 2, 3])` renvoie `False` (5 n'est pas dans la liste).

### Exercice 3 : Où se trouve k ?

**But :** écrire la fonction `pos(k, L)` qui retourne la position de k dans la liste si k est présent, ou -1 si k n'est pas dans la liste.

**Exemple :** `pos(2, [3, 1, 2])` renvoie `2` et `pos(2, [4, 5, 6])` renvoie -1. N'oublie pas que les indices commencent à 0.

### Exercice 4 : Maximum

**But :** écrire la fonction `maximum(L)` qui retourne le plus grand des éléments de la liste non triée L.

**Exemple :** `maximum([1, 2, 5, 3])` renvoie `5` , `maximum([12, 3, 8, 1])` renvoie `12` .

**Bonus :** écrire la fonction `pos_max(L)` qui retourne la position du maximum dans la liste non triée L.

**Exemple :** `pos_max([1, 2, 5, 3])` renvoie `2` , `pos_max([12, 3, 8, 1])` renvoie `0` .



### Exercice 5 : Somme

**But :** écrire la fonction `sum_list(L)` qui retourne la valeur de la somme de tous les éléments de la liste L.

**Exemple :** `sum_list([1, 2, 3])` renvoie `6`, `sum_list([0, 1, 1, 0, 0])` renvoie `2`.

### Exercice 6 : ²

**But :** écrire la fonction `square(L)` qui retourne une liste LS contenant les éléments de la liste L au carré (multipliés par eux-mêmes).

**Exemple :** $2^2 = 2 * 2 = 4$ et le code suivant :

```
1 L = [1, 2, 3]
2 LS = square(L)
3 print(LS)
```

Affiche `[1, 4, 9]`.

Autre exemple, `square([2, 5, 6])` renvoie `[4, 25, 36]`.

### Exercice 7 : facto-quoi ?

**But :** écrire la fonction `fact(n)` qui remplit une liste avec les valeurs factorielles de 0 à n et qui la retourne.

En maths, la factorielle d'un entier $n$ est le produit des nombres entiers strictement positifs inférieurs ou égaux à $n$. Cette opération est notée avec un point d'exclamation : $n!$.

Par convention : $0! = 1$.

Par exemple :

- $3! = 3 * 2 * 1 = 6$
- $42! = 42 * 41 * 40 * ... * 3 * 2 * 1 =$??? (à toi de nous dire !)

**Exemple :** `fact(4)` renvoie `[1, 1, 2, 6]`.

### Exercice 8 : double trouble

**But :** écrire la fonction `double(L)` qui renvoie `True` si on peut trouver deux doublons **consécutifs** dans la liste L, ou `False` autrement.

**Exemple :** `double([1, 2, 2, 3])` renvoie `True` car il y a deux fois le chiffre 2 à la suite (en position 1 et 2) tandis que `double([1, 2, 3])` et `double([1, 2, 1])` renvoient `False` car il n'y a jamais deux nombres identiques consécutifs.

## Introduction aux chaînes de caractères

Les chaînes de caractères sont tout simplement des suites de caractères, de lettres : des mots, des phrases, et vous les avez en fait déjà manipulées dans le TP précédent, notamment à l'aide de la fonction `print`.

En anglais, on appelle les chaînes de caractères des "*strings*", et caractère devient "*character*" abrégé "*char*".

En Python, les strings se matérialisent sous la forme de listes de caractères, et on retrouve les mêmes propriétés !

Par exemple :

```
1 s = "Hello World!"
2 print(s[0])
```

Nous renverra 'H'.

Comme vous avez pu le voir, les strings s'écrivent entre guillemets `"` mais peuvent aussi s'écrire entre apostrophes `'`.

On peut aussi accéder à des morceaux de strings avec l'aide des crochets et deux points :

```
1 s = "girls can code"
2 print(s[0:5])
```

Nous renverra "girls".

On récupère donc le segment de chaîne entre l'indice 0 compris et l'indice 5 exclu !

Les strings peuvent contenir des *caractères spéciaux*. Ces caractères sont similaires aux caractères normaux, mais ne peuvent pas s'écrire directement.

Par exemple :

Le caractère *sauter une ligne*, il ne peut pas vraiment s'écrire dans le code.

```
1 print("Ceci est une
2 nouvelle ligne")
3
4 # Erreur !
5 # Python considère, la plupart du temps, qu'une nouvelle ligne est
6 # une nouvelle instruction.
```



Le caractère `"` ou `'`, qui est interprété comme le caractère de fin d'une string.

```
print("Le canard m'a dit "Bonjour !"")

# Erreur !
# Python considère que si vous écrivez ", c'est pour indiquer que la
# string est terminée.
```

Pour pallier ce problème il existe une façon d'écrire les caractères spéciaux. Cette façon consiste à écrire le caractère `\` suivi d'un autre caractère.

```
print("Ceci est une\n nouvelle ligne")
# \n permet de sauter une ligne

print("Le canard m'a dit \"Bonjour !\"")
# \" permet d'écrire " sans terminer la string
```

Il existe d'autres caractères spéciaux comme *la tabulation* (qui permet d'insérer un grand espace avec `\t`) ou encore les emojis (`\u265E`).

### Exercice 1 : les palindromes !

Les palindromes sont des mots qui peuvent se lire dans les deux sens. Par exemple, les mots "kayak", "radar" et "php" (le langage informatique !) sont des palindromes.

On considère que certaines phrases peuvent être des palindromes si l'on en retire les espaces et les possibles accents.

La phrase "Engage le jeu que je le gagne." de l'écrivain Alain Damasio est un palindrome.

**But :** écrire la fonction `palindrome(s)` qui renvoie `True` si la chaîne de caractères `s` est un palindrome, et `False` autrement.

### Exercice 2 : occurrences

**But :** écrire la fonction `count(c, s)` qui retourne le nombre de fois qu'on trouve le caractère `c` dans la chaîne `s`.

## Bonus : un peu d'algorithmique !

### Exercice 1 : faire le tri !

Nous chez Prologin, on aime bien que nos affaires soient rangées, classées, bien ordonnées ! Pour ça, on utilise des algorithmes de tri.



**But :** écrire la fonction `trier(L)` qui trie une liste L en ordre croissant (sans utiliser la méthode *sort()* bien sûr !).

Tu peux t'intéresser aux algorithmes de tri à bulles, tri par sélection, tri rapide, …

### Exercice 2 : Introduction à la Récursivité

La récursivité est la propriété de pouvoir appliquer une même règle plusieurs fois en elle-même ; dans notre cas, écrire un algorithme qui va s'appeler lui-même.

Par exemple, pour une liste, on va d'abord appliquer un calcul sur le premier élément de la liste, puis rappeler notre fonction sur le **reste** de la liste.

Si ce n'est toujours pas clair, n'hésite pas à appeler un organisateur, qui pourra t'expliquer ce que c'est, avec des schémas en plus !

**Factorielle** Rappelez vous la fonction factorielle, on peut la recoder de façon récursive avec :

```
1  def facto(n):
2      if n == 0:
3          return 1
4      else:
5          return facto(n - 1) * n
```

**Tic tac tic tac ...** **But :** écrire la fonction `tictac(n)` qui affiche le décompte de n jusqu'à 0, puis affiche "Boum !", de façon récursive ! C'est à dire que la fonction s'appelle elle-même.

```
1  tictac(10)
2  >>>10
3  >>>9
4  >>>8
5  >>>7
6  >>>6
7  >>>5
8  >>>4
9  >>>3
10 >>>2
11 >>>1
12 >>>0
13 >>>Boum!
```

**Fibonacci !** **But :** écrire la fonction récursive `fibo(n)` qui calcule et retourne le n-ième nombre de la suite de Fibonacci.



Qu'est-ce que la suite de Fibonacci ? Tu peux aller sur cette page pour découvrir ce que c'est : https://fr.wikipedia.org/wiki/Suite_de_Fibonacci.

Algorithme récursif :

— fibo(0) = 0 et fibo(1) = 1,
— Si n <= 1 : renvoyer n
— Sinon : renvoyer fibo(n - 1) + fibo(n - 2)

Attention, ne testez pas votre fonction `fibo(n)` avec des nombres trop grands, cela prendrait beaucoup trop de temps !

### Exercice 3 : cherche Médor, cherche !

Alors c'est bien d'avoir des listes bien remplies, mais parfois, on cherche à retrouver un élément stocké quelque part dans la liste.

Un algorithme de recherche efficace sur listes **déjà triées** est l'algorithme de recherche dichotomique, *binary search* en anglais. Le principe est le même que quand on cherche un mot dans le dictionnaire : imaginons je cherche "pomme" et je vois à un endroit "poire" et plus loin "porte", je sais alors que "pomme" est entre les deux !

**Principe :**

```
recherche_dichotomique_récursive(élément, liste_triée):
   m = longueur de liste triée / 2 ;
   si liste triée[m] = élément :
       renvoyer m ;
   si liste triée[m] > élément :
       renvoyer recherche_dichotomique_récursive(élément, liste_triée[1:m]) ;
   sinon :
       renvoyer recherche_dichotomique_récursive(élément, liste_triée[m:l]) ;
```

Pour en savoir plus, tu peux aller sur cette page : https://fr.wikipedia.org/wiki/Recherche_dichotomique

**But :** implémenter la fonction `bin_search(k, L)` qui prend en paramètre une liste *triée par ordre croissant* `L` et retourne la position de `e` dans la liste s'il y est présent, -1 sinon. Cette fonction devra implémenter une recherche dichotomique.

**Exemple** : `bin_search(2, [1, 2, 3])` retourne `1`