LISTES ET AUTRES OBJETS SÉQUENTIELS EN PYTHON

2BCPST1 - LYCÉE THIERS

Table des matières

1. Les listes en `python`

1.1. **Les listes.** Dès que l'on commence à manipuler en python un grand nombre de données l'usage de variable numérique devient insuffisant.

En `python` la structure de donnée qui va nous aider est la *liste*. c'est un **objet** de type `list` qui permet de collecter des éléments : données de type quelconque : `int`, `float`, `str`, `bool`, ou d'autres listes, etc...

• On peut définir une liste en explicitant ses éléments.
Une liste est créée à l'aide d'une affectation. Ses éléments sont entre crochets `[.]`.
Exemple :

```
>>> liste = [1, 2, 3, 'toto']
```

```
  >>> print(liste)
[1, 2, 3, 'toto']
>>> type(liste)
<class 'list'>
```

• La fonction `len(.)` prend en argument une liste et retourne son nombre d'éléments.

```
>>> len(liste)
4
```

• Les éléments d'une liste s'obtiennent grâce à leur **indice** entre crochets. Attention le premier élément a pour indice 0 !

```
>>> liste[0], liste[1], liste[len(liste) - 1]
(1, 2, 'toto')
```

Exemple : calcul de la moyenne de notes.

• La fonction `sum(.)` prend en argument une liste et retourne, lorsque c'est possible, le résultat de l'opération `'+'` sur ses éléments.

Saisissons la liste des notes.

```
>>> l = [10, 12, 14, 6, 8, 15, 3, 17]    # la liste de notes
```

Définition de la fonction `moyenne(.)` qui s'applique à toute liste non vide de nombres.

```
>>> def moyenne(liste):                  # fonction moyenne(.)
...     return sum(liste) / len(liste)
...
```

```
>>> moyenne(l)
10.625
```

On obtient le résultat attendu, la moyenne est de 10.625.

On peut aussi procéder sans la fonction `sum()`, à l'aide d'une boucle `for` :

```
>>> def moyenne(liste):                  # fonction moyenne(.)
...     somme = 0.0
...     for x in liste:
...         somme = somme + x
...     return somme / len(liste)
...
```

• Les listes sont des objets **modifiables** (on dit aussi **mutables**) : on peut modifier leurs éléments.

```
>>> liste = [1, 2, 3, 'toto']    # une liste
```

On modifie un élément de la liste en lui affectant une nouvelle valeur (de n'importe quel type, simple : `int, float,...` ou complexe : `list, str,...`).

```
>>> liste[0] = 'le début'
>>> liste[2] = [-1, -2, -3]
>>> print(liste)
['le début', 2, [-1,- 2, -3], 'toto']
```

• L'indice -1 permet d'obtenir le dernier élément. C'est plus simple que `liste[len(liste) - 1]`. L'indice -2 l'avant-dernier, etc...

```
>>> print(liste[-1], liste[-2])
'toto' [-1, -2, -3]
```

Un indice qui n'est pas compris entre `-len(liste)` et `len(liste)-1` produit une erreur `'IndexError'`.

```
>>> print(liste[-5], liste[4])
... IndexError: list index out of range
```

• La commande `in` permet de déterminer si un élément appartient ou non à une liste.

```
>>> liste = [1, -3, 5, 17.0, 2]
>>> 1 in liste
True
>>> -1 in liste
False
>>> -3.0 in liste
True
>>> 17 in liste
False
>>> 'toto' in liste
False
```

• Avec en plus la commande `for` on peut faire parcourir à une variable les éléments d'une liste :

```
>>> for i in liste:
...     print(i)
...
1
-3
5
17.0
2
```

• Par exemple pour répéter 3 fois une instruction :

```
>>> liste = [0, 1, 2]
>>> for i in liste:
...    print('Bonjour')
...
Bonjour
Bonjour
Bonjour
```

1.2. **Création d'un itérateur avec `range()`.** La dernière application du parcours d'une liste pour répéter une séquence d'instructions est intéressante. Mais couteuse en mémoire puisqu'une liste stocke en mémoire toutes les valeurs qu'elle contient.

• Pour cette raison (depuis la version 3 de `python`), la fonction `range(n)` retourne un *itérateur* sur les `n` premiers entiers naturels (de 0 à `n-1` inclus). Les valeurs intermédiaires ne sont plus stockées en mémoire. Cependant la commande `in` permet encore de déterminer l'appartenance d'un élément :

```
>>> print(range(10))
range(0,10)
>>> 2 in range(10)
True
>>> 10 in range(10)
False
```

• On peut convertir le résultat retourné en une liste grâce à la fonction de conversion `list()` :

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

• Avec deux arguments entiers `m,n`, `range(m,n)` retourne un itérateur sur les $\max(0, n-m)$ entiers consécutifs qui sont $m \leqslant . < n$.

```
>>> print(list(range(0,10))); print(list(range(-1,11)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

• Avec trois arguments entiers `m, n, k`, `range(m, n, k)` retourne un itérateur sur la liste des entiers de la forme `m+k*p` avec `p` entier naturel qui sont compris entre `m` (inclu) et `n` (exclu).

```
>>> print(list(range(0,10,2))); print(list(range(10,0,-2)))
[0, 2, 4, 6, 8]
[10, 8, 6, 4, 2]
```

Ainsi : `range(n)` est équivalent à `range(0,n)` et à `range(0,n,1)`.

```
>>> list(range(0,10,-1))    # produit la liste vide
[]
```

Par contre un argument non entier provoque un message d'erreur '`TypeError`'.

Attention : Dans `python 2`, la fonction `range()` retourrne non pas un itérateur mais une liste ; c'est la fonction `xrange()` qui retourne un intégrateur (cf. annexe).

1.3. **Exemples.**

1.3.1. *Exemple 1 : la liste des carrés d'entiers compris entre 0 et 20.* Nous souhaiterions créer la liste des carrés des entiers compris entre 0 et 20.

Pour cela on utilisera la **méthode** list.append() appliquée aux objets de type liste qui permet d'ajouter un élément en queue de liste :

```
>>> liste=[ ] ; print(liste)
[]
>>> liste.append('toto') ; print(liste)
['toto']
>>> liste.append('le héros') ; print(liste)
['toto', 'le héros']
```

Solution :

```
>>> listeCarrés = [ ]        # initialisation
>>> for i in range(21):
...    listeCarrés.append(i ** 2)    # actualisation
...
```

```
>>> print listeCarrés
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361, 400]
```

1.3.2. *Exemple 2 : premiers termes de la suite Fibonacci.* Ecrire une fonction `fibonacci()` qui prend en argument un entier `N` et retourne une liste contenant les N premiers termes de la suite de Fibonacci : $u_0 = 0$, $u_1 = 1$, $\forall\, n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n$.

Solution :

```
 >>> def fibonacci(N):
...    result = [0, 1]        # Initialisation
...    for k in range(2, N)
...        result.append(result[k - 1] + result[k - 2])
...    return result
```

La définition de la fonction est proche de la définition par récurrence de la suite, la boucle `for` jouant le rôle de la relation de récurrence.

```
 >>> fibonacci(16)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

1.4. **Les méthodes de la classe `list`.**

Pour appilquer une méthode à une liste il faut faire suivre le nom de la liste d'un point puis du nom de la méthode avec entre parenthèses ses paramètres.

Méthode :	Action :
`liste.append(x)`	Pour ajouter `x` à la fin de la liste `liste`
`liste.extend(liste2)`	Pour ajouter la liste `liste2` à la suite de la liste `liste`
`liste.insert(i,x)`	Pour insérer l'élément `x` en position `i` dans `liste`
`liste.pop()`	Pour retirer et renvoyer le dernier élément dans `liste`
`liste.pop(i)`	Pour retirer et renvoyer l'élément en position `i` dans `liste`
`liste.remove(x)`	Pour retirer la première occurence de `x` dans `liste`
`liste.index(x)`	Renvoie la 1^{ere} position de `x` dans `liste`. Message d'erreur si aucune.
`liste.count(x)`	Renvoie le nombre d'occurrences de `x` dans `liste`.
`liste.sort()`	Trie la liste par ordre croissant.
`liste.reverse()`	Renverse l'ordre des éléments de la liste.

- Méthodes `append(), extend(), insert(), pop()` : pour ajouter ou retirer un élément d'une liste via son indice.
- Méthode `remove()` : pour retirer le premier élément d'une liste d'une valeur donnée.
- Méthode `index()` : pour chercher un élément dans une liste.
- Méthode `count()` : pour compter les éléments d'une liste.
- Méthodes `sort(), reverse()` : manipulation de liste : triage et renversement.

1.5. **Saucissonnage ou slicing.**

```
>>> liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
>>> liste1 = liste[2:5]    # tranche (slice) de la liste
>>> print(liste1)
['c', 'd', 'e']
```

L'instruction `>>> liste1 = liste[2:5]` crée une nouvelle liste `liste1` dont les éléments sont ceux de `liste` allant de l'indice 2 (inclus : le 3ème élément) à l'indice 5 (exclus : jusqu'au 5ème élément, celui d'indice 4). On l'appelle une *tranche* (slice en anglais) de la liste.

Les indices entre crochets peuvent sortir de la plage d'indice de la liste :

```
>>> liste2 = liste[0:100]    # tranche des indices de 0 à 100
>>> print(liste2)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

Un troisième paramètre définit un pas :

```
>>> liste3 = liste[6:2:-1]    # tranche des indices de 6 à 2 par pas de -1
>>> print(liste3)
['g', 'f', 'e', 'd']
```

Les ':' définissent les champs, les valeurs de ces derniers étant optionnelles.

LISTE[Indice départ (inclus) : Indice arrivée (exclus) : Pas]

`python` crée la tranche en copiant l'élément de `LISTE` d'indice `Indice de départ`, puis tous les éléments obtenus en ajoutant successivement `Pas` à l'indice, tant qu'on ne dépasse pas `Indice d'arrivée`. **Par défaut :** Pas = 1

- si Pas > 0 : **par défaut** indice départ = 0 ; indice arrivée = `len`(LISTE)
- si Pas < 0 : **par défaut** indice départ = `len(LISTE)-1` : indice arrivée = -1

```
>>> liste4 = liste[::2]    # 2 slicing par pas de 2
>>> print(liste4)
['a', 'c', 'e', 'g', 'i']
liste5 = liste[::-1]    # 2 slicing par pas de -1
['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

```
>>> liste5 = liste[3::2]    # départ de l'indice 3, par pas de 2
>>> print(liste5)
>>> ['d', 'f', 'h']
liste6 = liste[:3:-1]  # départ de la fin par pas de -1, arrêt avant
l'indice 3,
>>> print(liste6)
['i', 'h', 'g', 'f', 'e']
```

```
>>> liste7 = liste[:]    # Pour copier une liste
>>> print(liste7)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

1.6. **Copie de liste.** Il faut prendre garde à la façon dont `python` copie une liste... Illustrons cela sur un exemple :

```
>>> liste = ['a', 'b', 'c']    # définition d'une liste
>>> copie = liste            # puis copie de la liste
>>> print(copie)
['a', 'b', 'c']
```

Copie d'une liste : jusqu'ici tout va bien.

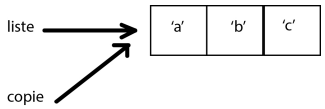
```
>>> liste[0] = 'modifié'    # Modifons un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
```

Modifions un élément de la liste originelle. Tout se passe comme prévu.

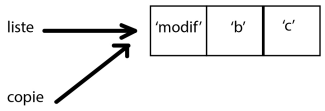
```
>>> print(copie)    # regardons ce qu'est devenue la copie
['modifié', 'b', 'c']
```

Mais, ô surprise, la copie aussi a été modifiée... Contrairement à ce qu'on aurait pu attendre.

Copie d'une liste : explication En fait une liste en `python` ne contient que l'adresse mémoire où sont stockés ses éléments ; c'est ce que l'on appelle un pointeur.



Quand on copie `liste` dans `copie` c'est cette adresse mémoire qui est copiée. C'est un alias qui est créé. On parle de copie par référence.



Quand on modifie un élément d'une liste, il est alors aussi modifié dans la copie.
L'avantage étant qu'on encombre moins la mémoire centrale puisque les éléments de la liste ne figurent qu'en un seul emplacement mémoire.

On peut contourner ce problème grâce à : `copie = liste[:]` qui fait une 'copie superficielle' :

```
>>> liste = ['a', 'b', 'c']    # définition d'une liste
>>> copie = liste[:]            # puis copie superficielle de la liste
>>> print(copie)
['a', 'b', 'c']
>>> liste[0] = 'modifié'    # Modifons un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
>>> print(copie)    # regardons ce qu'est devenue la copie
['a', 'b', 'c']
```

`python` fait une copie des éléments de la liste.. mais lorsqu'un élément est une liste, c'est l'adresse mémoire des objets de la liste qui est copiée. Aussi si l'un des éléments est une liste on retombe sur le même problème :

```
>>> liste = ['a', 'b', [0, 1]]    # définition d'une liste
>>> copie = liste[:]             # puis copie non-superficielle de la liste
>>> liste[0] = 'modifié'         # Modifons un élément dans liste
>>> liste[2][0] = 'MODIF'         # et un élément dans liste[2]
>>> print(liste); print(copie)    # regardons ce qu'est devenue la copie
['modifié', 'b', ['MODIF',1]]
['a', 'b', ['MODIF',1]]
```

Pour contourner totalement le problème utiliser la méthode `liste.deepcopy()` qui effectue une 'copie profonde'. Il faut avoir auparavant importé la bibliothèque `copy` :

```
>>> from copy import deepcopy; copie = deepcopy(liste).
```

1.7. **Liste définie par compréhension.** On peut définir une liste à l'aide des mots-clés `for`, `in` et `if` comme on définit un ensemble en mathématiques 'par compréhension' :

`[f(x) for x in liste]` correspond à $\{f(x) | x \in liste\}$

Exemple :

```
>>> liste = range(10)        # définition de la liste des entiers de 0 à 9
>>> LISTE = [x**2 for x in liste] # LISTE des carrés des éléments de liste
>>> print(LISTE)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`[f(x) for x in liste if Condition(x)]` correspond à $\{f(x) | x \in$ `liste` tel que `Condition(x)`$\}$

Exemple :

```
>>> Liste = [x**2 for x in liste if (x%2==0)] # Carrés des éléments pairs
>>> print(Liste)
[0, 4, 16, 36, 64]
```

Exemple : Liste des multiplies de 12 entre 0 et 100 :

```
>>> list = [x for x in range(101) if x % 12 == 0]    # Multiples de 12
>>> print(list)
[0, 12, 24, 36, 48, 60, 72, 84, 96]
```

Exemple : Liste des diviseurs d'un entier naturel N :

```
>>> N=120
>>> diviseurs = [x for x in range(1,N+1) if (N % x == 0)] # Diviseurs de N
>>> print(diviseurs)
[1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
```

• **Exemple : application au calcul de la variance d'une série statistique.**

Une série statistique pourra être donnée par une liste de nombres.

• Fonction calculant l'espérance d'une série statistique :

```
def esperance(X):    # X est une liste numérique non vide
    return sum(X) / len(X)
```

On l'utilisera pour le calcul de la variance d'une série statistique :

• Calcul de la variance à l'aide de la définition : $V(X) = E((X - E(X))^2)$:

```
def variance(X):    # X est une liste numérique non vide
    m = esperance(X)
    Y = [ (x - m) ** 2 for x in X ]
    return esperance(Y)
```

• Calcul de la variance à l'aide de la formule de Koenig-Huygens : $V(X) = E(X^2) - E(X)^2$:

```
def variance(X):    # X est une liste numérique non vide
    X2 = [ x ** 2 for x in X ]
    return esperance(X2) - esperance(X)**2
```

2. Structures de données séquentielles

2.1. Opérations communes aux types séquentiels.

Les types `int`, `float`, `bool` sont des *types scalaires*.

Les types `list` (listes) et `str` (chaînes de caractère) sont des structures de données *séquentielles*.

Tous les objets de type séquentiel ont en commun les opérations suivantes :

Opération	Résultat
`s[i]`	élément d'indice `i` de `s`
`s[i:j]`	Tranche de `i` (inclus) à `j` (exclus)
`s[i:j:k]`	Tranche de `i` à `j` par pas de `k`
`len(s)`	Longueur de `s`
`max(s), min(s)`	Plus grand et plus petit élément de `s`
`x in s`	`True` si `x` est dans `s`, `False` sinon
`x not in s`	`True` si `x` n'est pas dans `s`, `False` sinon
`s+t`	Concaténation de `s` et `t`
`s*n, n*s`	Concaténation de `n` copies de `s`
`s.index(x)`	Indice de la 1^{ere} occurrence de `x` dans `s`
`s.count(x)`	Nombre d'occurrences de `x` dans `s`

où `s` et `t` sont des objets séquentiels de même type, et `i, j, k, n` sont des entiers.

2.2. Les chaînes de caractère.

2.2.1. *les chaînes de caractères.* Les objets de type `str`, chaînes de caractère, sont des structures de données séquentielles :

```
>>> chaine = 'Amanda'
>>> len(chaine)
6
>>> print(chaine[::-1])
adnamA
>>> chaine.count('a')
2
>>> print(chaine*2)
AmandaAmanda
>>> max(chaine)
'n'    # minuscules sont > majuscules puis ordre alphabétique
```

Ce ne sont pas des objets modifiables : changer un élément produit une erreur `TypeError` :

```
>>> chaine[0] = 'Z'
TypeError: 'str' object does not support item assignment
```

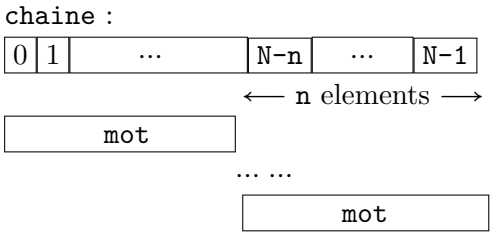
Exemple : Ecrire une fonction qui teste si une chaine est un palindrome :

```
>>> def palindrome(c):
...     if (c == c[::-1]): return True
...     else: return False
```

Exemple : Recherche d'un mot dans une chaîne. La recherche d'un mot dans une chaîne de caractère est un problème essentiel en informatique qui admet des algorithmes élaborés et efficients (temps d'exécution, utilisation de la mémoire).

Donner une solution naïve, pas très efficiente n'est pas difficile :

```
def contient(chaine,mot):
    N = len(chaine)
    n = len(mot)
    for i in range(N-n+1):
        if (mot == chaine[i:i+n]):  # comparaison du mot et d'une tranche
            return True
    return False
```



Le résultat est lent est couteux en mémoire car la fonction doit effectuer jusqu'à `(N-n)` copies d'une liste de longueur `n`, et pour chacune `n` comparaison, soit de l'ordre de `(N-n)*2n` opérations et `(N-n)*n` fois l'emplacement nécessaire au stockage d'un caractère utilisé.

2.3. **les `tuples`.** En français `t-uplets`. Ce sont des objets séquentiels.

On les définit par la donnée de leurs éléments entre parenthèses `(.)`. Ils diffèrent des listes surtout en ce que ce sont des objets non-modifiables (non-mutable) :

```
>>> seq = (0,1,2,3)
>>> print(seq)
(0, 1, 2, 3)
>>> print(seq[0])
0
>>> seq[0] = 1
[...]
TypeError: 'tuple' object does not support item assignment
```

`seq[0] = 1` produit une erreur de type :`TypeError`.