Travaux pratiques d'introduction 8

Listes : adjonctions

Informatique tronc commun MPSI et PCSI

I Présentation

I.1 Les méthodes

Nous avons défini le type `list` de python et les fonctions de base. Jusqu'à présent nous utilisions les listes avec une longueur fixée : on pouvait changer les valeurs de chacune des positions mais on ne pouvait pas ajouter un élément supplémentaire.
On peut créer une nouvelle liste avec un élément de plus :

```
>>> l = [4, 3, 7, 6, 0]
>>> l1 = l + [5]
>>> l1
[4, 3, 7, 6, 0, 5]
>>> l
[4, 3, 7, 6, 0]
```

Cette opération ne modifie pas la liste initiale mais crée une nouvelle liste : cela nécessite de copier tous les éléments et a donc un coût que l'on souhaite éviter. Python a prévu cette possibilité sous la forme d'une méthode.

Définition 1 : méthodes

Une méthode est une fonction d'un type spécial qui est invoquée en ajoutant son nom **après** l'objet auquel elle s'applique avec un point de séparation. Une méthode peut avoir des paramètres, ils seront placés classiquement entre les parenthèses qui suivent le nom de la méthode.

I.2 La méthode append

I.2.a Présentation

Définition 2 : `append`

L'instruction `liste.append(x)` attache l'élément `x` au bout de la liste `liste`. C'est une instruction élémentaire qui procède sans affectation, elle ne fait que modifier sans créer.

```
>>> a = [1, 2, 3]
>>> a.append(5)
>>> a
[1, 2, 3, 5]
```

est équivalent à :

```
>>> a = [1, 2, 3]
>>> a = a + [5]
>>> a
[1, 2, 3, 5]
```

À retenir

Préférez la version avec `.append()` qui est plus compacte, plus facile à lire et est plus rapide à exécuter.

Il est toujours possible de mélanger dans une même liste des variables de type différent. On peut d'ailleurs mettre une liste dans une liste.

```
>>> liste = []
>>> liste
[]
>>> liste.append(1)
>>> liste
[1]
>>> liste.append("ok")
>>> liste
[1, 'ok']
```

À retenir : résultat de append

La fonction `append` modifie la liste mais ne renvoie rien, c'est-à-dire qu'elle ne renvoie pas d'objet récupérable dans une variable. Il s'agit d'un exemple d'utilisation de méthode (donc de fonction particulière) qui fait une action mais qui ne renvoie rien.

I.2.b Application d'une fonction aux éléments d'une liste

Nous avons vu qu'une liste de taille n pourra être construite selon le motif suivant :

```
def carre(liste):
    n = len(liste)
    l2 = [0]*n
    for i in range(n):
        l2[i] = liste[i]**2
   return l2
```

On peut construire une liste sans connaître à l'avance la taille de la liste. Une première conséquence est qu'il est possible de construire une liste de taille n en partant d'une liste vide et y adjoindre n éléments.

```
def carre(liste):
    l2 = []
    for x in liste:
        l2.append(x**2)
    return l2
```

Autre méthode :

```
def carre(liste):
    return [x**2 for x in liste]
```

On a maintenant deux méthodes possibles pour construire une liste de taille donnée, la construction avec `append()` étant légèrement plus lente.

I.3 La fonction pop

L'opération inverse de `append` est notée `pop` ; l'instruction `liste.pop()` a deux effets.

Définition 3 : pop

L'instruction `liste.pop(i)` enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, `liste.pop()` enlève et renvoie le dernier élément de la liste.

```
>>> a = [1, 2, 3]
>>> a.pop()
3
>>> a
[1, 2]
```

II Exercices

II.1 Compréhension des méthodes

Exercice 1 - append
Donner ce qui est affiché en fin d'exécution de programme suivant (sans le taper sur l'ordinateur)

```
L = [0,1,58,50,74,65,1,2,9,86,5,45,1,71,23,25,74,65,19,37,50]
Resultat1 =[]
Resultat2 = []
Resultat3 = []
Med = 50
for i in range (len(L)):
    if  L[i] < Med:
        Resultat1.append(L[i])
    elif L[i] > Med:
        Resultat2.append(L[i])
    else:
        Resultat3.append(L[i])
print(Resultat1, "-", len(Resultat1))
print(Resultat2, "-", len(Resultat2))
print(Resultat3, "-", len(Resultat3))
```

Exercice 2 - pop
Donner ce qui est affiché en fin d'exécution de programme suivant (sans le taper sur l'ordinateur)

```
L =[0,1,58,50,74,65,1,2,9,86,5,45,1,71,23,25,74,65,19,37,50]
T =5
while len(L)>T:
    L.pop()
Resultat = L
print(Resultat)
```

II.2 Programmation

Dans la suite des exercices, on pourra tester les fonctions avec la liste :

```
Ex = [ 0, -2,  1, -2, -2, -1,  2,  3,  3,  0,  0,  0,  1,  2,  0,
      -2, -1, -1,  1,  3,  3,  1,  3,  0,  2, -1, -1,  1,  1,  1]
```

Exercice 3 - Termes positifs
Écrire une fonction `positifs(L)` qui renvoie la liste des termes strictement positifs de `L`, dans le même ordre.
`positifs(Ex)` renvoie `[1, 2, 3, 3, 1, 2, 1, 3, 3, 1, 3, 2, 1, 1, 1]`

Exercice 4 - Recherche
Écrire une fonction `positions(x, L)` qui renvoie la liste des indices i en lesquels `L[i]` vaut x, la liste renvoyée sera vide si x n'apparaît pas dans la liste `L`.
`positions(-1, Ex)` renvoie `[5, 16, 17, 25, 26]`

Exercice 5 - Positions du maximum
Écrire une fonction `indices_max(L)` qui renvoie la liste des indices i en lesquels `L[i]` vaut le maximum de la liste.
On pourra, dans un premier temps, utiliser l'exercice 4 après avoir calculé la valeur du maximum. Il existe cependant une méthode qui permet de trouver le résultat en ne parcourant qu'une fois la liste.
`indices_max(Ex)` renvoie `[7, 8, 19, 20, 22]`

Exercice 6 - Doublons
Écrire une fonction `sans_doublon(L)` qui renvoie la liste des valeurs de `L` dans le même ordre mais sans répétitions dans des termes consécutifs. `sans_doublon(Ex)` renvoie `[0, -2, 1, -2, -1, 2, 3, 0, 1, 2, 0, -2, -1, 1, 3, 1, 3, 0, 2, -1, 1]`

Exercice 7 - Création de deux groupes d'élèves dans une classe
Soit la liste suivante des prénoms des élèves d'une classe, sans majuscules :

```
Liste = ['victor', 'alexandre', 'maxime', 'chloé', 'zoé',
         'mickael', 'louis', 'paul', 'antoine', 'etienne']
```

Sous Python, on obtient la place d'une lettre dans l'alphabet entre 1 et 26 en écrivant `Place = ord(Lettre)- ord('a')+ 1`. où `Lettre` est une variable contenant la lettre étudiée sous forme de chaîne de caractères (`str`).
Proposer un code permettant de créer les listes `Liste1` et `Liste2` des élèves respectivement dans la première moitié et la seconde moitié de l'alphabet.

II.3 Étude du parenthésage d'expressions arithmétiques

On veut étudier le parenthésage d'expressions arithmétiques ou du code source d'un programme. Il est utile de déterminer la parenthèse ouvrante associée à une parenthèse fermante, les éditeurs montrent souvent les appariements de parenthèses.
Par exemple, dans $1+2*(7-(4-3)*((2-5)+2*((12/4-8)+2*3))$, la parenthèse ouvrante après $(4-3)*$ est associée à l'avant dernière parenthèse fermante.
On considère une chaîne de caractère représentant l'expression et on s'intéresse uniquement aux parenthèses classiques "(" et ")". On rappelle qu'une chaîne de caractères se manipule comme si elle était une liste de caractères, avec la particularité qu'elle n'est pas modifiable.

Exercice 8 - Test de parenthésage
Écrire une fonction `testPar(ch)` pour qu'elle renvoie `True` ou `False` selon que la chaîne de caractères est bien parenthésée ou non.

Pour associer une parenthèse fermante à la parenthèse ouvrante associée, on peut remarquer que celle-ci est la dernière parenthèse ouvrante non encore associée. Pour la calculer on peut appliquer l'algorithme suivant :

- On crée une liste vide pour les couples de parenthèses, `par`.
- On crée une liste vide pour les indices de parenthèses ouvrantes, `ouv`.
- On lit les caractères un par un,
  - quand on voit une parenthèse ouvrante on ajoute (`ouv.append`) sa position,
  - quand on lit une parenthèse fermante à la position j, on enlève (`ouv.pop`) la dernière valeur depuis la liste `ouv`, i, ce sera bien la dernière parenthèse ouvrante non encore associée ; on ajoute alors (`par.append`) le couple (i, j) à la liste `par`.

Dans la chaîne `"1+2*(7-(4-3)*((2-5)+2*((12/4-8)+2*3)))"` les opérations seront

Chaine lue	i	`ouv`	`par`
"1"	0	`[]`	`[]`
"$1+2*($"	4	`[4]`	`[]`
"$2*(7-($"	7	`[4,7]`	`[]`
"$-(4-3)$"	11	`[4]`	`[(7, 11)]`
"...$)*($"	13	`[4,13]`	`[(7, 11)]`
"...$*(($"	14	`[4,13,14]`	`[(7, 11)]`
"...$-5)$"	18	`[4,13]`	`[(7, 11), (14, 18)]`
"...$2*($"	22	`[4,13,22]`	`[(7, 11), (14, 18)]`
"...$*(($"	23	`[4,13,22,23]`	`[(7, 11), (14, 18)]`
"...$-8)$"	30	`[4,13,22]`	`[(7, 11), (14, 18), (23, 30)]`
"...$*3)$"	35	`[4,13]`	`[(7, 11), (14, 18), (23, 30), (22, 35)]`
"...3)"	36	`[4]`	`[(7, 11), (14, 18), (23, 30), (22, 35), (13, 36)]`
"...)))"	37	`[]`	`[(7,11),(14,18),(23,30),(22,35),(13,36),(4,37)]`

Exercice 9 - Liste des parenthèses associées
Écrire une fonction `listePar(ch)` qui reçoit une chaîne de caractères supposée bien parenthésée et qui retourne la liste des couples d'indices de parenthèses associées.

Exercice 10 - Test de parenthésage bis
Écrire une fonction `mauvaisePar(ch)` pour qu'elle renvoie -1 si la chaîne est bien parenthésée ou bien un indice i indiquant la position d'une parenthèse ouvrante non fermée ou la position d'une parenthèse fermante sans parenthèse ouvrante associée.
Par exemple `listePar("(1+2))*(5-3)")` renvoie 5.

III Solutions

Solution de l'exercice 1 - append

```
[0, 1, 1, 2, 9, 5, 45, 1, 23, 25, 19, 37]-12
[58, 74, 65, 86, 71, 74, 65]-7
[50, 50]-2
```

On pouvait obtenir le même résulytat avec, par exemple,

```
Resultat1 = [x for x in L if x > 50]
```

Solution de l'exercice 2 - pop

```
[0, 1, 58, 50, 74]
```

On pouvait obtenir le même résulytat avec, par exemple,

```
Resultat = L[ : 5]
```

Solution de l'exercice 3 - Termes positifs

```
def positifs(L) :
      pos = []
      for x in L :
          if x > 0:
          pos.append(x)
    return pos
```

Python propose aussi la construction `[x for x in L if x >= 0]`.

Solution de l'exercice 4 - Recherche

```
def positions (x, L):
      n = len(L)
      pos = []
      for i in range (n):
        if L[i] == x:
        pos.append(i)
      return pos
```

Python propose aussi la construction `[i for i in range(len(L))if L[i]] == x]`.

Solution de l'exercice 5 - Positions du maximum

```
def indices_max(L):
    maxi = L[0]
    for x in L :
        if x  > maxi:
            maxi = x
    return positions(maxi, L)
```

En un seul passage :

```
def indices_max(L):
    n = len(L)
    maxi = L[0]
    pos = [0]
    for i in range (1, n):
        if L[i] > maxi:
            maxi = L[i]
            pos = [i]
        elif L[i] == maxi:
            pos.append(i)
    return pos
```

Solution de l'exercice 6 - Doublons

On n'ajoute un élément que s'il est distinct de son prédécesseur, le premier terme n'a pas à être comparé.

```
def sans_doublon(L):
    n = len(L)
    L1 = [L[0]]
    for i in range(1, n):
        if L[i] != L[i -1]:
            L1.append(L[i])
    return L1
```

Solution de l'exercice 7 - Création de deux groupes d'élèves dans une classe

```
Taille =len(Liste)
Liste1 = []
Liste2 = []
for i in range(Taille):
    Prenom = Liste[i]
    Lettre = Prenom[0]
    Place = ord(Lettre)- ord('a') + 1
    if Place <= 13:
        Liste1.append(Prenom)
    else:
        Liste2.append(Prenom)
print(Liste1)
print(Liste2)
```

Solution de l'exercice 8 - Test de parenthésage

Il suffit de compter le nombre de parenthèses ouvrantes non encore fermées.
Un manque de parenthèses ouvrantes se voit quand ce nombre est nul alors qu'on lit une parenthèse fermante, un excès de parenthèses ouvrantes se voit à la fin si le nombre est non nul.

```
def testPar(ch):
    ouv = 0
    for car in ch:
        if car == "(":
            ouv = ouv + 1
        if car == ")":
            if ouv == 0:
                return False
            else:
                ouv = ouv - 1
    return ouv == 0
```

Solution de l'exercice 9 - Liste des parenthèses associées

```
def listePar(ch):
   n = len(ch)
   par = []
   ouv = []
   for i in range(n):
       car = ch[i]
       if car == "(":
           ouv.append(i)
       if car == ")":
           j = ouv.pop()
           par.append((j, i))
    return par
```

Solution de l'exercice 10 - Test de parenthésage bis

```
def mauvaisePar(ch):
    n = len(ch)
    ouv = []
    for i in range(n):
        car = ch[i]
        if car == "(":
          ouv.append(i)
        if car == ")":
            if ouv == []}:
                return i
            else:
                j = ouv.pop()
   if ouv == []}:
       return -1
   else:
       return ouv.pop()
```

Si on veut détecter tous les problèmes possibles, on peut aussi renvoyer toutes les paires de parenthèses, y comprises celles qui ne sont pas appariées, en indiquant avec `-1` la parenthèses manquante.

```
def parentheses(ch):
    n = len(ch)
    par = []
    ouv = []
    for i in range(n):
        car = ch[i]
        if car == "(":
            ouv.append(i)
        if car == ")":
            if ouv == []}:
                par.append((-1, i))
            else:
                j = ouv.pop()
                par.append((j, i))
    while ouv != []}:
        i = ouv.pop()
        par.append((i, -1))
    return par
```