Programmation en Python - Cours 4 : Listes et autres structures de données

MPSI - Lycée Thiers

2014/2015

Nous allons approfondir l'utilisation des listes en `python` et voir les autres structures de données : leurs opérations et méthodes.

Méthodes de la classe `list`

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.

- Pour ajouter ou retirer un élément d'une liste via son indice.
- Pour retirer le premier élément d'une liste d'une valeur donnée.
- Pour chercher un élément dans une liste

  Noter aussi que `x in liste` retourne `True` ou `False` selon si `x` apparaît ou non dans `liste`.
- Manipulation de liste : triage et renversement.

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']
```

Saucissonnage ou slicing

Les ' :' définissent les champs, qui sont optionnels.

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']
```

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']
```

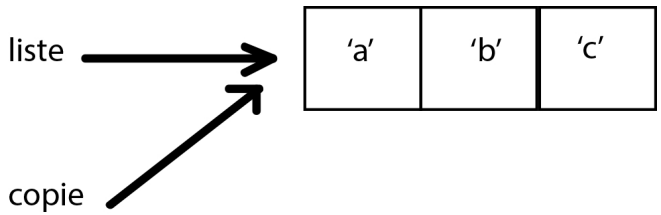
```
>>> liste[0] = 'modifié'    # Modifons un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
```

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

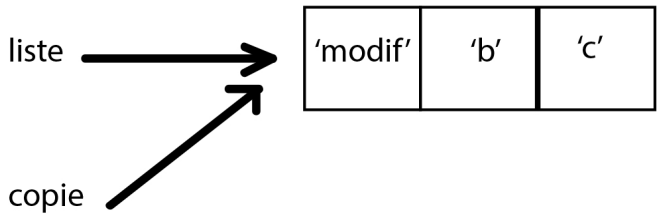
- Copie d'une liste : jusqu'ici tout va bien.
- Modifions un élément de la liste originelle. Tout se passe comme prévu.
- Mais, ô surprise, la copie aussi a été modifiée... Contrairement à ce qu'on aurait pu attendre.

Copie de 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.



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.

Copie de liste

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)
```

Compréhension de liste

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]
```

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 de *type 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.

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 :

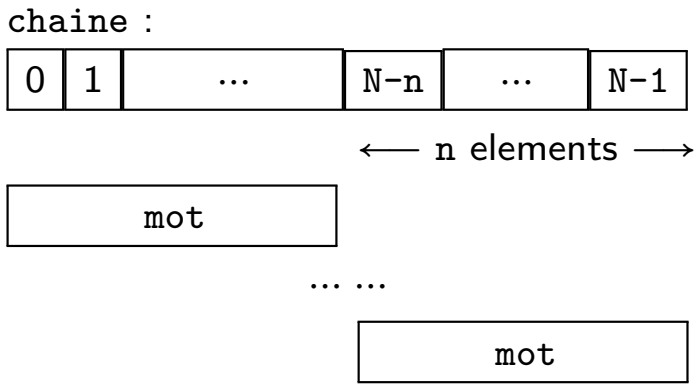
```
>>> def palindrome(c):
...     if (c == c[::-1]): return True
...     else: return False
```

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 de la tranche de lgr n
            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.

les structures de données tuple

En français t-uplet. Ce sont des objets séquentiels.

On les définit par la liste de leurs éléments (comme pour une liste mais) entre parenthèses (.).

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

Ils diffèrent des listes surtout en ce que ce sont des objets non-modifiables (non-mutable) :

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

Les parenthèses sont optionnelles ; nous les avons déjà utilisées par l'usage des virgules :

```
>>> a = 2
>>> 2*a, 3*a, 4*a    # retourne un t-uplet
(4, 6, 8)
```

Les dictionnaires

Ce sont des structures de données qui ne sont pas séquentielles : un élément n'est plus repéré à l'aide d'un indice mais à l'aide d'un nom (sa clef). C'est un *champ* : couple d'une clef et de sa valeur. La valeur peut être de type quelconque, la clef un nombre une chaine ou un t-uplet.

Cette structure de donnée s'appelle en `python` un *dictionnaire* et dans d'autres langages un *enregistrement*.

Un dictionnaire se définit entre accolades $\{.\}$ par la suite des champs (la clef suivie de la valeur, séparés de `:`).

France = {'Capitale' : 'Paris', 'Superficie' : 674800, 'Population' : 65000000, 'Langue' : 'Français'}

C'est identique à :

```
France = {}
France['Capitale'] = 'Paris'
France['Superficie'] = 674800
France['Population'] = 65000000
France['Langue'] = 'Français'
```

Les dictionnaires

L'accès à un élément se fait à l'aide de sa clef :

```
>>> France['Capitale']
'Paris'
```

Pour ajouter un champ affecter une valeur à une nouvelle clef. Pour supprimer un champ utiliser la fonction `del()`.

```
>>> France['Continent'] = 'Europe'     # ajout d'un champ
>>> del(France['Capitale'])          # suppression d'un champ
>>> print(France)
{'Continent' : 'Europe', 'Superficie' : 674800, 'Population' : 65000000,
'Langue' : 'Français'}
```

Les dictionnaires admettent les méthodes suivantes :

`dict.keys()`	retourne la liste des clefs du dictionnaire `dict`
`dict.values()`	retourne la liste des valeurs du dictionnaire `dict`
`dict.items()`	retourne la liste des champs `(key,value)` de `dict`
`dict.copy()`	retourne une copie du dictionnaire.

Exemple :

Pour enregistrer une liste constituée d'un dictionnaire pour chaque élève d'une classe d'effectif 47 élèves, chaque dictionnaire comportant des champs pour les nom, prénom, age, 2eme langue :

```
eleves = []
for i in range(47):
    dict = {}
    dict['Nom'] = raw_input('Nom ? ')
    dict['Prénom'] = raw_input('Prénom ? ')
    dict['Age'] = int(raw_input('Age ? ')
    dict['LV2'] = raw_input('Seconde Langue ?')
    eleves.append(dict)
```

Remarque : le dictionnaire `dict` est une variable locale, une nouvelle est créée à chaque passage dans la boucle `for`, qui n'a plus rien à voir avec les définitions de `dict` précédentes.
Ecrivons maintenant une fonction qui recherche si un élève de nom donné est dans la classe et si oui retourne son dictionnaire.

```
def search(nom):
    for e in eleves:
        if e['Nom'] == nom:
            return e
    return False
```

Exemples :

- Ecrire une fonction qui convertit une chaîne de caractère de minuscule à majuscule.
- On utilisera la table de caractère ASCII : une table des caractères, codés par un nombre entre 0 et 255 (0x00 et 0xFF, un octet) :
  – `chr(n)` retourne le caractère ASCII de code `n` ;
  – `ord('a')` renvoie le code ASCII du caractère `'a'`.
  – Les lettres minuscules (et majuscules) sont rangées dans l'ordre alphabétique.
- A partir du code ASCII `ord('x')` d'un caractère `'x'`, le code ASCII du caractère converti en majuscule est : `ord('x')+ord('A')-ord('a')`.

```
def upper(s):     # convertit une chaine en majuscule
    n = len(s)     # n= lgr de la chaine
    result = ''     # result initialement chaine vide
        for i in range(n):
            if ord('a') <= ord(s[i]) <= ord('z'):
                result += chr(ord(s[i]) + ord('A') - ord('a'))
            else : result += s[i]
    return result
```

- Exemple d'appel de la fonction `upper()` :

```
>>> upper("Bonjour l'Univers")
"BONJOUR L'UNIVERS"
```

- On obtient le même résultat avec l'appel de la méthode prédéfinie de la classe `str` :

```
>>> "Bonjour".upper()
"BONJOUR"
```

Exemples

• Ecrire une fonction qui prend en paramètre une chaîne de caractère et retourne la liste de ces mots. Les mots sont délimités par les limites de la liste et par les espaces.

• Réponse :

```
def tranche(s):
    n = len(s)    # n lgr de la chaine s
    result = []    # initialement result est la liste vide
    mot = ''    # initialement mot est la chaine vide
    for i in range(n):     # on parcourt la chaine s
        if s[i] != ' ':    # si le caractère lu n'est pas un espace
            mot += s[i]     # on l'ajoute à la fin de mot
            flag = True     # et on en garde le souvenir
        else:              # si le caractère lu est un espace
            if flag:    # si le caractère précédent n'était pas un espace
                result.append(mot)    # ajouter mot à la liste result
                mot = ''    # réinitialisation de mot
                flag = False    # on se souvient qu'il s'agissait d'un espace
    if flag: result.append(mot)    # A la sortie de for ajouter le dernier mot
    return result
```

• Exemple :

```
 >>> tranche('Quels sont les mots de cette longue phrase?')
['Quels', 'sont', 'les', 'mots', 'de', 'cette', 'longue', 'phrase?']
```

Exemples

• Ecrire un programme qui convertit une date saisie sous la forme "3 décembre 2013" en "3/12/2013". On utilisera les fonctions que l'on vient d'écrire.

• Réponse :

```
def conversion_date(s):
    dict = {'JANV': '01', 'FEVR':'02', 'MARS':'03', 'AVRI':'04', 'MAI':'05',\
    'JUIN':'06', 'JUIL':'07', 'AOUT':'08', 'SEPT':'09', 'OCTO':'10', 'NOVE':'11',\
    'DECE':'12'}
    mots = tranche(s)    # retourne la liste des mots
    mois = mots[1]    # mot contient le mois
    mois = upper(mois[0:4])    # slicing (4 lettres) puis conversion majuscule
    return mots[0] + '/' + dict[mois] + '/' + mots[2]
```

un backslash permet d'écrire une instruction sur plusieurs lignes.

• Exemple :

```
>>> conversion_date('3  janvier 2013')
'3/01/2013'
```

fonctionne même en présence de plusieurs espaces successifs (voire `tranche()`) ou de différentes casses.

Exemple : le crible d'Erathostène

• Ecrire une fonction qui prend en paramètre un entier positif `n` est retourne la liste des nombres premiers inférieurs ou égaux à `n` en appliquant le crible d'Erathostène.

• Erathostène, philosophe, géographe, astronome et mathématicien grec du IIIe siècle avant J.C. est célèbre pour son calcul de la circonférence de la terre, ainsi que pour son crible, un des premiers *tests de primalité*.

• Le crible d'Erathostène commence à barrer dans la liste des entiers de 2 à $\sqrt{n}$ tous les multiples stricts des nombres non barrés qui sont successivement lus. A la fin tous les nombres non barrés sont premiers.

```
def erathostene(n):
    liste = range(2,n+1)    # listes des entiers de 2 à n
    for i in range(n-1):     # parcours de la liste par indice (pour la modifier)
        if liste[i] != 0:     # si le nombre est premier
            k=2
            while k*liste[i] <= n:
                liste[i+(k-1)*liste[i]] = 0    # Mettre à 0 les multiples
                k += 1
    return [p for p in liste if p!=0]        # retourne les éléments non nuls
```

• Exemple :

```
>>> erathostene(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97]
```