Informatique en CPGE (2018-2019)
Les types composés

S. B.

Lycée des EK

13 décembre 2018

Définition

Un objet de type composé est constitué de plusieurs objets ; nous étudions ici des suites d'objets qui sont indexés par des entiers indiquant leur position dans la suite. Nous avons déjà rencontré le type **str** pour les chaînes de caractères et nous allons étudier les types **tuple** et **list**.

Définition

Les **n-uplets**, (couples, triplets, quadruplets, . . .), sont des objets de type **tuple**, (uplet en français se dit tuple en anglais). Ces objets sont des suites ordonnées d'éléments, très semblables aux chaînes de caractères et avec des fonctions communes. Les éléments d'un n-uplet peuvent être de n'importe quel type et ne pas être tous du même type.

Pour construire un objet de type **tuple**, on utilise des parenthèses (facultatives) pour la lisibilité et les éléments sont séparés par des virgules, par exemple :

```
a = (3, 5.2) # ou a = 3, 5.2
b = ('date', 24, 'décembre')
c = (a, b)       # couple formé d'un couple et d'un triplet
x, y = c # x vaut a et y vaut b
```

Attention : pour un 1-uplet, contenant par exemple l'unique élément entier 3, on écrit `(3,)`. Il ne faut surtout pas oublier la virgule.

Affectation multiple : l'affectation `a = 3, 5.2` crée aussi le couple (3, 5.2). Inversement, on peut déconstruire le couple par `x,y = a` ; on obtient x = 3 et y = 5.2. Ceci permet d'expliquer ce qui se passe lorsqu'on écrit `x,y = y,x` : cette instruction crée d'abord un couple t = (y, x), (expression à droite du signe =), puis déconstruit t en affectant à x le premier élément du couple et à y le deuxième. C'est pourquoi on obtient ainsi l'échange des variables x et y.

L'accès à une composante ou une suite de composantes se fait comme pour les chaînes de caractères : `a[0]` donne le nombre entier 3, `b[2]` donne le mot 'décembre', `b[0:2]` donne ('date', 24).
Comme pour les chaînes de caractères, on peut concaténer, avec l'opérateur +, un n-uplet avec un p-uplet pour obtenir un (n+p)-uplet : a + b est le quintuplet (3, 5.2, 'date', 24, 'décembre').
On obtient la longueur d'un n-uplet avec la fonction `len` comme pour les chaînes de caractères : `len(a+b)` renvoie 5 et `len(c)` renvoie 2 (c'est un couple).

Appartenance

On teste l'appartenance à un n-uplet comme pour les chaînes de caractères avec l'opérateur `in` :

```
>>> 'j' in 'bonjour'
True
>>> 'décembre' in b
True
>>> 'e' in b
False
```

Note

Les objets de type **tuple** sont, comme les objets de type **str**, **immuables** : on ne peut pas changer un élément de l'objet par une affectation. L'instruction `b[1]=25` renvoie une erreur. Ceci est intéressant dans certains cas, mais on peut aussi avoir besoin de pouvoir modifier la valeur d'un élément et il existe un type d'objet pour cela, c'est le type **list**.

Définition

Une liste, objet de type **list**, est d'une certaine manière un n-uplet dont on peut changer les valeurs des éléments. C'est donc en particulier un ensemble ordonné d'éléments éventuellement hétérogènes. Une liste en Python correspond à ce que l'on appelle un tableau dans plusieurs autres langages.

Syntaxe

Les éléments d'une liste sont séparés par des virgules et entourés de crochets. Les opérateurs fonctionnent comme pour les n-uplets. Par exemple :

```
liste1 = [2, 5, 3, 8]
liste2 = ['vert', 'rouge']
liste1[2] = 'bleu'        # on change un élément de la liste
print(liste1)        # affiche [2, 5, 'bleu', 8]
liste3 = liste1 + liste2        # concaténation de deux listes
print(liste3)        # affiche [2, 5, 'bleu', 8, 'vert', 'rouge']
liste4 = [liste1, liste2]        # liste de listes
print(liste4)        # affiche [[2, 5, 'bleu', 8], ['vert', 'rouge']]
```

Attention, une liste composée d'un unique élément a s'écrit [a] sans virgule après le a contrairement à la règle pour les n-uplets.

On peut utiliser la fonction `range` pour initialiser une liste d'entiers ou l'opérateur * pour la répétition :

```
liste1 = list(range(4))        # [0, 1, 2, 3]
liste2 = list(range(3,6))        # [3, 4, 5]
liste3 = list(range(3,11,2))        # [3, 5, 7, 9]
liste4 = 3*liste2        # [3, 4, 5, 3, 4, 5, 3, 4, 5]
```

Construction par compréhension

C'est un procédé très utile de construction d'une liste.
L'instruction est `[expression(i) for i in range(...)]`, par exemple :

```
carre = [x**2 for x in range (1,10)]
# construit [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Une liste est "itérable". Ainsi si on a défini une fonction f qui s'applique au type des éléments d'une liste, on peut calculer la liste des images de chaque élément avec le simple code suivant :

```
maliste = ...
images = [f(x) for x in maliste]
```

Insertion et extraction

Pour ajouter un élément à la fin d'une liste, on utilise la méthode `append` :

```
liste = ['vert', 'rouge']
liste.append('jaune')        # on ajoute un élément à la fin de la liste
print(liste)        # affiche ['vert', 'rouge', 'jaune']
```

Pour insérer un objet dans une liste, on utilise la méthode `insert` ; pour extraire une sous-liste, on opère comme pour le type **str** :

```
liste = [1, 3, 7, 9]
liste.insert(2, 5)        # on insère à l'index 2 l'élément 5
print(liste)        #affiche [1,3,5,7,9], 7 et 9 sont décalés vers la droite
liste2 = liste[1 : 4]
print(liste2)        #affiche [3,5,7]
print(liste[-1])        #affiche 9, le dernier élément
```

Suppression d'un élément et tri

On utilise la méthode `remove` pour supprimer un élément et la méthode `sort` pour trier une liste :

```
liste = [1, 3, 7, 9, 3, 5]
liste.remove(3)       # on supprime l'élément 3 (seul le premier rencontré !)
print(liste)          # affiche [1, 7, 9, 3, 5], 7, 9, 3 et 5 sont décalés vers la gauche
liste.sort()          # trie la liste (ou liste2 = sorted(liste) )
print(liste)          # affiche [1, 3, 5, 7, 9]
```

Référence

Un objet de type **list** peut être modifié. C'est très pratique lorsque c'est voulu mais aussi dangereux par exemple avec le code suivant où on pourrait croire modifier seulement la liste `liste2` :

```
liste1 = [1, 3, 5, 7, 9]
liste2 = liste1
liste2.append(11)
print(liste1)       # affiche [1, 3, 5, 7, 9, 11]
```

On peut penser que `liste2` est une copie de `liste1`. Mais en fait `liste1` et `liste2` contiennent une **référence** vers le même objet. Si l'objet est modifié avec l'une des deux variables, la modification se voit aussi avec l'autre.

Donc pour modifier une liste sans toucher à l'autre, il faut faire une véritable copie, c'est-à-dire créer un nouvel objet. Le code est le suivant :

```
liste1 = [1, 3, 5, 7, 9]
liste2 = list(liste1)        # liste2 est un nouvel objet
liste2.append(11)        # seule liste2 est modifiée
print(liste1)        # affiche [1, 3, 5, 7, 9]
```

Une liste peut être composée d'objets de différents types et en particulier de listes. Une liste composée de n listes de longueurs p représente un tableau ou une matrice (n, p) (n lignes et p colonnes).

On commence par créer une liste vide `matrice`, puis avec une boucle `for`, on crée les listes `ligne` une par une en les ajoutant à la liste `matrice` :

```
matrice = [ ]
for n in range(6) :     # 6 lignes
        ligne = [ . . . ]     # une ligne de longueur p
        matrice.append(ligne)
```

La commande `print(matrice[2])` permet par exemple d'afficher la troisième ligne. Puisque `matrice[2]` est une liste, la commande `print(matrice[2][4])` permet par exemple d'afficher le cinquième élément de la troisième ligne. De manière générale `matrice[i][j]` désigne le (j+1)ème élément de la (i+1)ème ligne.

Par exemple :

```
matrice = [ ]
for n in range(3) :     # 3 lignes
        ligne = [j*j for j in range(3*n, 3*(n+1)]     # 3 colonnes
        matrice.append(ligne)

print(matrice[1])         # affiche [9, 16, 25]
print(matrice[1][2])          # affiche 25
```

Attention, pour faire une copie d'une liste de listes, il faut traiter chaque sous-liste. Par exemple :

```
m = [[1, 2], [3, 4]]
mc = []
for i in range(2) : # 2 lignes
        ligne = list(m[i])
        mc.append(ligne)
```

Et si nous avons une liste de listes de listes ... ? Le plus simple dans tous les cas est d'utiliser la fonction `deepcopy` du module `copy` avec la syntaxe :

```
from copy import deepcopy
m = [[1, 2], [3, 4]]
mc = deepcopy(m)
```

Une image est un tableau (n lignes et p colonnes) de pixels (picture elements) et chaque pixel, dans le système RVB, est constitué de trois couleurs (rouge, vert, bleu). Dans un fichier au format bmp les pixels sont écrits les uns à la suite des autres ; chaque couleur est codée par un nombre de 0 à 255 codé par un octet et il faut donc trois octets pour coder un pixel.

Afin d'écrire ou de lire un tel ficher on peut donc utiliser une liste de n listes de longueurs p si on s'intéresse à chaque pixel, ou bien une liste de n listes de longueur $3 \times p$ si on s'intéresse à chacun des trois octets formant un pixel.

Attention : il peut y avoir un problème au niveau de la mémoire qui ne peut stocker des listes trop grandes. La taille maximale est de l'ordre de quelques centaines de millions. Il est possible de faire un test, par exemple avec l'instruction `liste=[0]*n`, pour déterminer la valeur maximale de n.