NSI - Première

Python - 5 - listes et tuples

qkzk

2021/04/22

Les listes et les tuples

Python propose deux types : `list` et `tuple` qui partagent beaucoup de points communs et qui sont généralement considérés comme des *tableaux*.

Ce sont des collections d'objets, de nature diverse et qu'on peut manipuler.

En première on se contente d'étudier ce type, d'apprendre à le parcourir, à en créer, à les modifier lorsque c'est possible.

En terminale on étudiera précisémment les types abstraits liste et tableau.

Le type `list`

Une `list` est une structure de données *indexées* (numérotés) à partir de 0.

Le premier élément d'une `list` porte donc l'indice 0.

On crée une `list` avec la syntaxe `[ ]` et les élément sont inclus dans les crochets.

```
>>> notes = [12, 20, 8, 14]          # notes est une liste
>>> type(notes)                      # son type est `list`
<class 'list'>
>>> len(notes)                       # les listes ont une longueur
4
>>> notes[0]                         # le premier élément
12
>>> notes[-1]                        # le dernier élément
14
```

- On accède aux éléments par leur indice.
- Comme pour les chaînes, le mot clé `in` permet de tester l'appartenance

```
>>> 3 in notes
False
>>> 12 in notes
True
```

Longueur d'une `list`

Comme les `str`, les `list` ont une longueur à laquelle on peut accéder avec `len`

```
>>> tab = [3, 2, 4]
>>> len(tab)
3
```

Exercice 0

1. Créer une `list` vide affectée à la variable `vide`.
2. Vérifier que sa longueur est nulle.
3. Vérifier qu'elle ne contient pas les entiers entre 1 et 10 à l'aide d'une boucle.
4. Créer à la main le tableau des entiers pairs de 0 à 10.
5. Mesurer sa longueur.
6. Vérifier, toujours à l'aide d'une boucle quels sont les entiers inférieurs à 10 qu'il contient.

Type des objets dans les `list`

Contrairement à beaucoup de langages, Python n'impose pas que les éléments d'une `list` soient tous du même type.

Ainsi le code suivant est parfaitement valide.

```
>>> l = [3, "bonjour", 2.32, True]
```

`range` et `list`

Lorsqu'on crée un `range`, le type n'est pas une `list` mais un *générateur*.

Pour les plus curieux, cela signifie que les objets ne sont pas crées immédiatement, mais de manière *paresseuse* lorsqu'on en a besoin.

Cela évite d'encombrer la mémoire avec des éléments inutiles.

Pour consulter tous les éléments d'un range, on peut le convertir en `list`

```
>>> mon_range = range(2, 20, 3)
>>> mon_range
range(2, 20, 3)
>>> type(mon_range)
<class 'range'>
>>> mon_range[2]
8
>>> len(mon_range)
6
>>> list(mon_range)
[2, 5, 8, 11, 14, 17]
```

Mutabilité des `list`

Les `list` sont des objets *mutables*, cela signifie qu'ils contiennent des éléments qu'on peut **modifier** au besoin.

On peut ajouter, supprimer, modifier des valeurs.

Modifier un élément dans une liste

On **modifie** un élément existant avec la notation `ma_liste[incide] = valeur`

```
>>> ma_liste = ["a", "b", "c"]
>>> ma_liste[0] = "z"
>>> ma_liste
["z", "b", "c"]
```

Attention : lorsque `indice` est supérieur ou égal à la taille de la `list` cela engendre une érreur `IndexError`.

```
>>> ma_liste[4] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Ajouter des valeurs

On ajoute une valeur à la fin d'une liste avec la méthode `append` :

```
>>> ma_liste = []                    # créer une liste vide
>>> ma_liste
[]
>>> ma_liste.append(5)               # ajouter 5 à la fin
>>> ma_liste
[5]
>>> for i in range(3):               # pour les entiers 0, 1, 2
...     ma_liste.append(i)           #       les ajouter à la fin
...
>>> ma_liste                         # ma liste contient bien les éléments
[5, 0, 1, 2]
```

Comparer des listes

En Python, deux listes sont *égales* si elles ont les mêmes éléments.

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2]
>>> l1 == l2
False
>>> l2.append(3)
>>> l2
[1, 2, 3]
>>> l1 == l2
True
```

Attention : cela ne signifie pas que ce sont les mêmes objets en mémoire...

Lorsque ce sont les mêmes objets en mémoire on dit qu'elles sont *identiques*.

Exercice 1

1. Créer les `list` : `[4, 3, 2, 1]` et `[1, 2, 3, 4]` et affectez les à des variables
2. Vérifier qu'elles contiennent les mêmes éléments.
3. Sont-elles pour autant égales ?

Dans une `list`, l'ordre des éléments compte

Exercice 2

Pour chaque question on créera la liste de deux manières :

- directement en l'écrivant à la main `l1 = [1, 2, 3]`
- avec `append` et *éventuellement* une boucle.

```
l2 = []
for i in range(1, 4):
    l2.append(i)
```

- ensuite on vérifie qu'elles sont égales.

```
>>> l1 == l2
True
```

1. Créer la liste des carrés des 10 premiers entiers (de 0 à 9).
2. Créer la liste des prenoms de vos 4 voisins les plus proches.
3. Créer la liste des entiers entre 0 et 100 qui se terminent par 7.
4. Créer la liste de vos cinq séries ou films préférés (remplacez par jeux-vidéos si vous préférez).

Concaténer des listes

Contaténer des listes se fait exactement de la même manière que pour les châines de caractères avec le symbole `+`

On obtient *une copie* des listes données.

```
>>> l1 = [1, 2, 3]
>>> l2 = [3, 4, 5]
>>> l1 + l2                          # concaténation
[1, 2, 3, 3, 4, 5]
>>> l1                               # l1 n'a pas changé
[1, 2, 3]
>>> l2
[3, 4, 5]
```

Retirer des valeurs

On retire une valeur avec le mot clé `del` ou la méthode `remove`.

```
>>> del ma_liste[1]                  # retire le SECOND élément
>>> ma_liste
[5, 1, 2]
>>> ma_liste.remove(5)               # retire la première occurence d'un 5
>>> ma_liste
[1, 2]
```

- `del` efface l'élément depuis son *indice*,
- `.remove` retire l'élément depuis sa *valeur*

Retirer et renvoyer

Les deux manières précédentes de retirer des objets utilisent des effets de bord et ne renvoient rien.

Il existe une autre méthode, `pop` qui, contrairement à `append`, retire le dernier élément et le renvoie.

```
>>> ma_liste.pop()
2
>>> ma_liste
[1]
>>> ma_liste.pop()
1
>>> ma_liste
[]
>>> ma_liste.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

Exercice 3

Les amis de JP.

JP a une mauvaise mémoire, il utilise des listes Python pour enregistrer les noms et les ages de ses amis.

Ami	Age
Raymond	93
Léonce	77
Jacqueline	44
Léonie	41
Hicham	28

1. Créer la liste des prénoms et la liste des ages des amis de JP.
2. C'est l'anniversaire de Hicham, augmenter son age de 1.
3. Léonce a malheureusement péri en essayant sa mobylette. Supprimez le des amis et n'oubliez pas de supprimer son age.
4. JP s'est marié avec Jacqueline. Ils se sont bien trouvés ! Jacqueline a aussi ses amis qui deviennent les amis du couple.

Créer la liste des amis et Jacqueline et celle des ages :

Ami	Age
JP	45
Léonie	41
Hicham	29
Fernande	32

5. Regrouper dans une seule liste des amis du couple. Supprimez ensuite 'JP' et 'Jacqueline' (ils sont maintenant époux et plus seulement amis...)

   Faire la même chose avec les ages. On n'utilisera que les méthodes présentées ci-dessus. Interdit de recréer les listes à la main.

Comme on peut le voir il n'est pas évident d'associer un nom à un age. La structure `list` *n'est clairement pas la plus adaptée à ce travail... Nous verrons les* `dict` *qui permettent de faire ça beaucoup plus aisément*

Copier une liste

Lorsqu'on souhaite *faire une copie* d'une liste il faut être très prudent.

```
>>> l1 = ["Pierre", "Paul", "Jacques"]
>>> l2 = l1          # l1 et l2 pointent vers le même objet
```

Ce n'est pas **UNE COPIE** mais la même liste !!!

```
>>> l1[0] = "Amandine"
>>> l2
["Amandine", "Paul", "Jacques"]
```

Lorsqu'on modifie `l1`, `l2` est modifiée aussi !

Pour faire une copie il y a plusieurs approches, la plus simple est d'utiliser la méthode `.copy`

```
>>> l1 = ["Pierre", "Paul", "Jacques"]
>>> l2 = l1.copy()                          # l2 est une copie de l1
>>> l1[0] = "Amandine"
>>> l2
["Pierre", "Paul", "Jacques"]
>>> l1
["Amandine", "Paul", "Jacques"]
```

Le type `tuple`

Un `tuple`, c'est comme une `list` mais ce n'est pas mutable et c'est plus rapide.

En pratique, on crée un tuple avec la notation `(1, 2, 3)`, des parenthèses plutôt que des crochets.

Les tuples ne sont pas mutables, on ne peut rien leur ajouter, retirer, modifier. Pour modifier un tuple, on en crée une copie et voila.

L'intérêt des tuples est qu'ils sont plus rapides. Lorsqu'on n'a pas besoin de modifier les éléments qu'ils contiennent, on privilégie les tuples.

Détupler

Une opération courante qu'on rencontre dans des programmes python consister à détupler des valeurs.

Par exemple :

```
>>> mon_tuple = (1, 2)
>>> x, y = mon_tuple            # détuplage
>>> x
1
>>> y
2
```

C'est commode mais source d'erreurs. Il faut s'assurer d'avoir à gauche du signe `=` autant de variables (séparées par des virgules) qu'il y en a à droite.

Fonction renvoyant un tuple

On rencontre aussi des fonctions qui renvoient plusieurs valeurs. Dans ce cas, Python renvoie toujours un tuple.

```
def foncion_qui_renvoie_deux_valeurs():
    return (20, 33)

mes_valeurs = foncion_qui_renvoie_deux_valeurs()
mes_valeurs[0]          # 20
mes_valeurs[1]          # 33
```

On peut faire exactement la même chose en l'écrivant ainsi :

```
def foncion_qui_renvoie_deux_valeurs():
    return 20, 33           # sans parenthèses !
```

Itérer sur les listes et les tuples

La boucle `for` permet d'itérer sur n'importe quelle collection et donc sur les listes et les tuples.

```python
semaine = ["lundi", "mardi", "mercredi", "jeudi"] # la flemme
for jour in semaine:
    print(jour)
```

Va afficher :

```
lundi
mardi
mercredi
jeudi
```

Exercice 4

Écrire un programme python qui créé une liste `mois` qui comprend les mois de l'année, puis à l'aide de parcours successifs de la liste effectuer les actions suivantes :

1. Afficher un à un les éléments de la liste `mois`
2. Afficher la valeur de `mois[4]`
3. Echanger les valeurs de la première et de la dernière case de cette liste
4. Afficher 12 fois la valeur du dernier élément de la liste
5. Peut-on réaliser toutes ces étapes avec un tuple ?

Exercice 5

1. Créer la liste des cubes des entiers entre 1 et 20. On utilisera un `range`.
2. Comment savoir à l'aide de votre liste si 245 est le cube d'un entier ?
3. Ajouter le cube de 0 à sa place.
4. Retirer 11^3 de la liste en utilisant `remove`.
5. Retirer 8^3 de la liste en utilisant `del` (comptez soigneusement avant sa position).
6. À l'aide d'une boucle `while`, retirer un par un les cubes de la liste en partant de la fin et les afficher. À la fin de votre boucle, la liste est vide.

   Voici ce qu'on doit voir :

   ```
   8000
   6859
   5832
   ...
   8
   1
   0
   ```

Fonctions, `list` et mutabilité

On aborde ici une particularité de Python qu'il faut garder en tête lorsqu'on manipule des listes.

En Python, il est possible de passer une liste à une fonction. Jusque là rien d'original, c'est faisaible dans tous les langages que je connais.

Par contre, attention, *dans la fonction* il est possible de modifier la liste.

Par exemple.

```python
def ajoute_un_trois(liste: list) -> None:
    '''ajoute l'élément 3 à la fin de la liste'''
    liste.append(3)

ma_liste = []                   # ici la liste est vide

ajoute_un_trois(ma_liste)       # ma_liste == [3]

ajoute_un_trois(ma_liste)       # ma_liste == [3, 3]

ajoute_un_trois(ma_liste)       # ma_liste == [3, 3, 3]

ajoute_un_trois(ma_liste)       # ma_liste == [3, 3, 3, 3]

ajoute_un_trois(ma_liste)       # ma_liste == [3, 3, 3, 3, 3]
```

Cette propriété est très pratique. On s'en servira en particulier pour trier des listes.

Visualisez le comportement dans Python Tutor

Exercice 6

Échanger deux variables. Python permet d'aller plus vite, mais déjà les bases !

Pour échanger les valeurs des variables `x` et `y` on utilise une troisième variable `z`.

```python
x = 2
y = 3
# on veut que x = 3 et y = 2 ... mais sans le taper à la main !
z = x
x = y
y = z
```

1. Que se passe-t-il si on exécute le code suivant ?

   ```python
   x = 2
   y = 3
   x = y
   y = x
   ```

   Vérifiez l'état des variables `x` et `y`.

2. Modifier l'exemple précédent pour partir de `x = 2, y= 3` et arriver au contraire.

3. Créer une fonction `echange_tete_queue` qui prend un paramètres du type `list` et échange le premier et le dernier élément de la liste.

4. Créer une fonction `renverser` qui prend une liste et renvoie *une copie* de celle-ci mais dans l'ordre contraire.

   ```
   >>> l1 = [1, 2, 3]
   >>> renverser(l1)
   >>> [3, 2, 1]
   >>> l1
   >>> [1, 2, 3]
   ```

5. Faire la même chose mais en modifiant la liste de départ. Cette fois `renverser` ne renvoie rien.

```
>>> l1 = [1, 2, 3]
>>> renverser(l1)
>>> l1
>>> [3, 2, 1]
```