Cours_liste

March 18, 2021

1 Type de base et type construit

En python, on connaît les types de base : `int`, `float`, `bool`, `str`

Rappel : le type `str` (*chaîne de caractères*) est un type déjà *plus complexe* puisqu'il s'agit d'une **séquence ORDONNÉE de caractères**.

[1]:
```
# L'ordre des caractères est important
"trie" == "tire"
```

[1]:
```
False
```

[2]:
```
# On accède à un caractère grâce à son INDICE, en utilisant la notation␣
↪crochets []
chaine = "Python"
chaine[3]
```

[2]:
```
'h'
```

A partir de ces types de base, on construit des structures de données plus complexes : les listes `list`, les tuples `tuple`, les dictionnaires `dict` (pour ne parler que des types construits *natifs*)

2 Les listes

2.1 Définition

Une liste est une **séquence ORDONNÉE** d'objet divers :

- une liste est une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets
- les objets d'une liste sont repérés par leur **INDICE** (notation crochets `[ ]`)
- les objets d'une liste peuvent être de type différent même si en pratique on utilise le plus souvent des listes d'objets de type identique

Vous pouvez consulter documentation officielle de python sur les listes

[3]:
```
# Définition dune liste (notation crochets) de 6 objets de type différents
ma_liste = ["lundi", 45, True, 6.7, "python", 22]
print(ma_liste)
```

```
['lundi', 45, True, 6.7, 'python', 22]
```

[4]:
```
# une liste est de type `list`
type(ma_liste)
```

[4]:
```
list
```

Attention, l'ordre des éléments d'une liste est important. Ainsi les 2 listes ci-dessous ne sont pas les mêmes !

[5]:
```
liste1 = [5, 7, 3]
liste2 = [3, 7, 5]

liste1 == liste2  # Utilisation de == (opérateur booléen "comparateur␣
↪d'égalité")
```

[5]:
```
False
```

Attention il existe beaucoup de similitude entre liste et chaîne de caractères mais on verra par la suite qu'il y a aussi des différences importantes !

3 Manipulation de listes

3.1 Accéder à un élément d'une liste

- Il suffit de mettre l'**indice** de l'élément souhaité entre crochet
  Syntaxe : `nomListe[indiceElement]`
- **Attention** la numérotation des indices commence à 0
- On peut aussi utiliser des indices négatifs qui correspondent à une numérotation en commençant par la fin

[6]:
```
ma_liste = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",␣
↪"dimanche"]
```

éléments de la liste	"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"
indices des éléments de la liste	0	1	2	3	4	5	6
indices des éléments de la liste	-7	-6	-5	-4	-3	-2	-1

[7]:
```
jour = ma_liste[0]
print(jour)
```

```
lundi
```

[8]:
```
print(ma_liste[6])
```

```
dimanche
```

[9]:
```
print(ma_liste[-2])
```

```
samedi
```

Tenter d'accéder à un indice qui n'existe pas donne lieu à une erreur `IndexError`

[10]:
```
print(ma_liste[7])
```

```
␣
↪---------------------------------------------------------------------------

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-10-9c722c1aed40> in <module>
----> 1 print(ma_liste[7])


IndexError: list index out of range
```

Attention aux erreurs de "débutants": Au début, lorsqu'on manipule des listes contenant des entiers, il est fréquent de confondre les entiers contenus dans la liste avec leurs indices…

[11]:
```
ma_liste = [1,5,2,4,3,7]
```

éléments de la liste	1	5	2	4	3	7
indices des éléments de la liste	0	1	2	3	4	5
indices des éléments de la liste	-6	-5	-4	-3	-2	-1

[12]:
```
ma_liste[0]
```

[12]:
```
1
```

[13]:
```
ma_liste[1]
```

[13]:
```
5
```

[14]:
```
ma_liste[2]
```

[14]: 2

[15]:
```
ma_liste[3]
```

[15]: 4

[16]:
```
ma_liste[4]
```

[16]: 3

3.2 Longueur d'une liste

La longueur d'une liste correspond au nombre d'éléments qu'elle contient. On accède à la longueur d'une liste grâce à la **fonction** `len()`

[17]:
```
ma_liste = [1, 6, 7, 5, 6 ,12, 14]

len(ma_liste)
```

[17]: 7

3.3 Suppression d'un élément d'une liste

On supprime un objet de la liste grâce à la fonction `del()`

[18]:
```
ma_liste = [1, 6, 7, 5, 6 ,12, 14]
print(ma_liste)

del(ma_liste[2])
print(ma_liste)
```

```
[1, 6, 7, 5, 6, 12, 14]
[1, 6, 5, 6, 12, 14]
```

Remarque importante : Quand on supprime un élément, **celui-ci ne laisse pas *"une place vide"***. Les éléments suivants viennent directement se mettre à la suite... et donc leur indice change !!

3.4 Modification d'une liste

[19]:
```
ma_liste = ['Alice', 'Bob', 'Tom']
print(ma_liste)

ma_liste[1] = 'Python'
print(ma_liste)
```

```
['Alice', 'Bob', 'Tom']
['Alice', 'Python', 'Tom']
```

Remarque : **on a remplacé** l'élément `'Bob` par `'Python'`. **on n'a pas inséré** `'Bob'` à la place d'indice 2 !!

3.5 Concaténation de listes

Concaténer des listes, c'est les mettre "bout à bout" pour en faire une seule.

[20]:
```
ma_liste1 = [1 ,2 ,3]
ma_liste2 = [4, 5, 6]

ma_liste3 = ma_liste1 + ma_liste2
print(ma_liste3)
```

```
[1, 2, 3, 4, 5, 6]
```

3.6 Slice de liste

Le slice correspond à un "morceau" d'une liste. la première borne est inclus, la deuxième exclue !

[21]:
```
ma_liste = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",␣
 ↪"dimanche"]
ma_liste[1:4]
```

[21]:
```
['mardi', 'mercredi', 'jeudi']
```

3.7 Test d'appartenance `in`

Syntaxe : `element in liste`

L'opérateur `in` renvoie un booléen :

- `True` si element est dans la liste
- `False` si element n'est pas dans la liste

[22]:
```
ma_liste = [1,5,2,4,3,7]
print(ma_liste)
6 in ma_liste
```

```
[1, 5, 2, 4, 3, 7]
```

[22]:
```
False
```

[23]:
```
5 in ma_liste
```

[23]:
```
True
```

3.8 Quelques usages courants

- Liste vide

[24]:
```
liste_vide = [ ]
```

[25]:
```
len(liste_vide)
```

[25]:
```
0
```

- Créer une liste à partir d'une chaîne de caractères grâce à la fonction `list`

[26]:
```
chaine =  "python"

liste=list(chaine)
print(liste)
```

```
['p', 'y', 't', 'h', 'o', 'n']
```

- Créer une liste des entiers successifs grâce aux fonctions `list` et `range`

[27]:
```
liste_entier = list(range(10))
print(liste_entier)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Créer une liste de plusieurs éléments tous identiques grâce à l'opérateur `*` (peut être utile pour initialiser une liste)

Je déconseille cet usage car il cache un effet de bord, visible sur les listes de listes (que l'on verra plus tard)...

[28]:
```
liste = [0]*10  # liste contenant dix 0
print(liste)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

[29]:
```
liste = ["Alice"]*5 # liste contenant cinq fois la chaîne de caractères 'Alice'
print(liste)
```

```
['Alice', 'Alice', 'Alice', 'Alice', 'Alice']
```

4 Parcourir une liste

Une liste, étant une **séquence ordonnée d'éléments**, se parcourt facilement avec une boucle `for`. On pourra soit :

- parcourir les éléments de la liste
- parcourir les indices des éléments de la liste

4.1 Parcourir les éléments d'une liste

[30]:
```
ma_liste = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",␣
↪"dimanche"]

for element in ma_liste:
    print(element)
```

```
lundi
mardi
mercredi
jeudi
vendredi
samedi
dimanche
```

4.2 Parcourir les indices des éléments d'une liste

[31]:
```
ma_liste = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",␣
↪"dimanche"]

for i in range(len(ma_liste)):
    print(i)
```

```
0
1
2
3
4
5
6
```

Arrêtons-nous sur la syntaxe `range(len(mot))` qui est très courante et qui pose parfois problèmes car on a 2 appels de fonctions imbriqués l'une dans l'autre. Quand cela se présente, il faut commencer par regarder l'appel "le plus à l'intérieur" :

1. `len(ma_liste)` = `len(["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"])` = 7
2. donc `range(len(ma_liste))` = `range(7)` = 0, 1, 2, 3, 4, 5, 6 $\Longrightarrow$ Il s'agit bien de la séquence composée des indices des éléments

Remarque : Une bonne façon d'assimiler cela est d'utiliser Thonny en mode debug : la décomposition en 2 étapes comme ci-dessus est parfaitement visible !

5 Les listes sont des objets

Il existe donc des **fonctions "réservées" aux listes appelées méthodes** qui s'utilisent avec la **notation pointée**

syntaxe : `ma_liste.methode(paramètres_éventuels)`

On obtient la liste des méthodes disponibles pour les objets de type `list` grâce à la fonction `dir()`

[32]:
```
dir(list)
```

[32]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
```

```
 'remove',
 'reverse',
 'sort']
```

Toutes les méthodes notées ainsi `__nomMethode__` sont dites spéciales et dépassent largement les notions de programmation utilisées en lycée. On peut donc conssidérer qu'il y a 11 méthodes disponibles pour les listes : `append`, `clear`, `copy`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` et `sort`

5.1 A quoi servent les méthodes définies pour les listes?

Les méthodes sont des fonctions, donc des programmes, permettant de réaliser des manipulations courantes mais non disponibles par les opérations de base.

Exemple : On a vu que le code suivant mettait l'élément `'Python'` en position d'indice 1 et de ce fait ecrase l'élément `'Bob'`

[33]:
```
ma_liste = ['Alice', 'Bob', 'Tom']
print(ma_liste)

ma_liste[1] = 'Python'
print(ma_liste)
```

```
['Alice', 'Bob', 'Tom']
['Alice', 'Python', 'Tom']
```

Comment faire pour **insérer** l'élément `'Python'` en position d'indice 1 entre l'élément `'Alice'` et `'Bob'` ?

Deux solutions :

- On écrit un bout de programme python qui réalise cette actions
- Plus malin : On utilise la méthode `insert`. (qui justement a été créée car **insérer est une manipulation de base** sur les listes !)

5.2 Comment utiliser une méthode ?

Parmi ces méthodes, `append` est celle qui vous sera le plus utile !! Pour savoir comment l'utiliser, il suffit de consulter la documentation :

[34]:
```
help(list.append)
```

```
Help on method_descriptor:

append(self, object, /)
    Append object to the end of the list.
```

On peut lire que `append` permet d'**ajouter un élément en fin de liste**.

Exemple :

[35]:
```
ma_liste = [] # liste vide

print(ma_liste)
```

```
[]
```

[36]:
```
ma_liste.append(5) # Ajout de l'entier 5 à la liste

print(ma_liste)
```

```
[5]
```

[37]:
```
ma_liste.append(8) # Ajout de l'entier 8 à la liste

print(ma_liste)
```

```
[5, 8]
```

6 Les listes sont mutables

Jusqu'ici, vous pouvez une grande similitude entre les listes et les chaînes de caractères. En effet toutes les deux :

- sont des séquences ordonnées
- utilisent la notion d'**indice**
- se par courent avec une boucle `for`
- etc ...

Exemples :

[38]:
```
liste = [2,6,3,8,9]
chaine = "python"
```

[39]:
```
liste[3]
```

[39]: 8

[40]:
```
chaine[3]
```

[40]: 'h'

[41]:
```
len(liste)
```

[41]: 5

[42]:
```
len(chaine)
```

[42]: 6

Néanmoins il existe une **différence fondamentale** illustrée par les exmples ci-dessous :

[43]:
```
liste = [2,6,3,8,9]

liste[3] = 12
print(liste)
```

```
[2, 6, 3, 12, 9]
```

[44]:
```
chaine = "python"

chaine[3] = "a"
print(chaine)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-44-09295e22ec6b> in <module>
  1 chaine = "python"
  2
----> 3 chaine[3] = "a"
  4 print(chaine)


TypeError: 'str' object does not support item assignment
```

[45]:
```
liste = [2,6,3,8,9]

del(liste[3])
print(liste)
```

```
[2, 6, 3, 9]
```

[46]:
```
chaine = "python"

del(chaine[3])
print(chaine)
```

```
␣
↪---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-46-9511f461e608> in <module>
  1 chaine = "python"
  2
----> 3 del(chaine[3])
  4 print(chaine)


TypeError: 'str' object doesn't support item deletion
```

En fait, les **listes sont mutables** et les **chaînes de caractères sont immuables**. Cela signifie qu'il est **possible de modifier une liste** mais il est **impossible de modifier une chaine de caractères**

Dans la pratique, cela aura de nombreuses conséquences sur lesquelles on reviendra plus tard…