

Chapitre 3

Listes, chaînes de caractères, dictionnaires

Nous allons voir dans ce chapitre nos premières structures de données un peu complexes : les listes, les chaînes de caractères et les dictionnaires.

3.1 Les listes

3.1.1 Généralités

Création et accès

Une liste est une collection ordonnée d'objets informatiques.

Pour créer une liste vide : `liste_vide = list()` ou `liste_vide = []`.

Pour créer une liste déjà remplie, on utilise les crochets en séparant les éléments de la liste par des virgules : `grosse_liste = [1, 3, "poire", 3, 0]`. Notez bien que les éléments ne sont pas nécessairement du même type et peuvent être répétés.

Pour accéder aux éléments, on utilise le nom de la liste, des crochets et les indices dans la liste. On utilise la même chose pour changer la valeur d'un élément.

Attention, comme d'habitude avec Python, la numérotation des indices commence à 0.

```
1 une_liste = [ 1, 5, -3, "poire", 12 ]
2
3 print(une_liste[1])  ## Affichera 5
4
5 une_liste[3] = 0
6 print(une_liste)  ## Affichera [1, 5, -3, 0, 12]
```

Pour connaître la taille de la liste, on utilise la fonction `len` :

```
1 une_liste = [ 1, 5, -3, "poire", 12 ]
2
3 taille = len(une_liste)
4 print(taille)  ## Affichera 5
```

Ajout, suppression et présence d'éléments

Pour ajouter un seul élément en fin de liste, on utilise la méthode `append`. Pour ajouter toute une liste d'un coup, on utilise la méthode `extend`, ou simplement l'opérateur `+` selon si on veut modifier la liste existante ou créer une nouvelle liste :

```
1 une_liste = [ 1, 5, -3 ]
2
3 une_liste.append(7)
4 print(une_liste)    ## Affichera [1, 5, -3, 7]
5
6 autre = [1, 0]
7 une_liste.extend(autre)
8 print(une_liste)    ## Affichera [1, 5, -3, 7, 1, 0]
9
10 nouveau = une_liste + autre
11 print(une_liste)    ## Affichera [1, 5, -3, 7, 1, 0]
12 print(nouveau)     ## Affichera [1, 5, -3, 7, 1, 0, 1, 0]
```

Attention, il y a une différence d'écriture entre `len(liste)` et `liste.append(2)`. Dans le premier cas, la fonction `len` prend une liste en argument. Dans le second cas, on agit directement sur la liste. Cette différence provient de deux paradigmes de programmation : le procédural et la POO (programmation orientée objet). Je ne rentrerai pas plus dans les détails car c'est hors programme (c'est bien dommage...).

Pour supprimer un élément d'indice donné, on utilise la méthode `pop` (par défaut, c'est le dernier de la liste). Pour supprimer la première occurrence d'une valeur donnée, on utilise la méthode `remove`.

Remarque : `pop` renvoie la valeur supprimée, `remove` ne renvoie rien.

```
1 une_liste = [ 8, "un", 5, 2, "un", -3, 7 ]
2
3 element = une_liste.pop()
4 print(une_liste)    ## Affichera [8, "un", 5, 2, "un", -3]
5 print(element)      ## Affichera 7
6
7 une_liste.pop(2)
8 print(une_liste)    ## Affichera [8, "un", 2, "un", -3]
9
10 element = une_liste.remove("un")
11 print(une_liste)    ## Affichera [8, 2, "un", -3]
12 print(element)      ## Affichera None: remove ne renvoie rien !
```

Pour vérifier qu'un élément est dans une liste, on utilise `in` :

```
1 une_liste = [ 8, 1, 5, 2, 1, -3, 7 ]
2
3 if "deux" in une_liste:
4     print("la chaine 'deux' est présente.")
5 else:
6     print("la chaine 'deux' est absente.")
7 ## Affichera "la chaine 'deux' est absente."
```

Parcours d'une liste

Pour parcourir une liste on utilise la structure `for in` :

```
1 une_liste = [ 8, 1, 5 ]
2
3 for element in une_liste:
4     print(element)
5 ## Affichera 8 puis 1 puis 5.
```

Notez la ressemblance avec la structure de boucle de d'habitude avec `range`. C'est normal, la fonction `range` produit une (sorte de) liste.

Si on souhaite aussi garder une trace de l'indice de l'élément, on utilise la fonction `enumerate`.

```
1 une_liste = [ 8, 1, 5 ]
2
3 for indice, element in enumerate(une_liste):
4     print(indice, ">=", element)
5 ## Affichera "0 >= 8" puis "1 >= 1" puis "2 >= 5".
```

Copie de liste

Un problème se pose :

```
1 une_liste = [ -1, 1, 5 ]
2 autre_liste = une_liste
3 une_liste[0] = 3
4
5 print(une_liste)      ## Affichera [3, 1, 5]
6 print(autre_liste)   ## Affichera [3, 1, 5]
```

L'affectation à la ligne 2 crée une nouvelle variable mais qui fait référence à la même liste. Les variables `une_liste` et `autre_liste` sont donc la même liste. Si on modifie l'une, l'autre est aussi modifiée. Pour obtenir deux listes distinctes, il faut utiliser la méthode `copy`.

```
1 une_liste = [ -1, 1, 5 ]
2 autre_liste = une_liste.copy() ## <-- copie de la liste : listes différentes
3 une_liste[0] = 3
4
5 print(une_liste)      ## Affichera [3, 1, 5]
6 print(autre_liste)   ## Affichera [-1, 1, 5]
```

3.1.2 Spécificité Python

Effet de bord

Un effet de bord est une modification d'un objet informatique dans une fonction sans que la fonction ne renvoie elle-même la modification.

```
1 def modif(liste, element):
2     liste.append(element)
3     print(element, "ajouté à la liste")
4
5 une_liste = [ 1, 5, 7 ]
6 modif(une_liste, -8)
7 ## Affichera "-8 ajouté à la liste"
8
9 print(une_liste)
10 ## Affichera [1, 5, 7, -8]
```

La liste est modifiée à l'intérieur de la fonction et la modification se propage à l'extérieur de celle-ci : c'est un effet de bord. Ces effets de bord ne fonctionnent pas avec des nombres car les nombres sont immutables, on est donc obligé d'affecter une nouvelle valeur, pas de changer ses caractéristiques.

Il faut être vigilant avec les effets de bord : ils peuvent être à l'origine de bugs difficiles à repérer.

Slicing

Pour obtenir le dernier élément de la liste, on peut utiliser `liste[-1]`. L'avant-dernier est donc `liste[-2]` et ainsi de suite.

Pour extraire une sous-liste, Python dispose du slicing (tronçonnage ?? en français) :

```
1 une_liste = [ -1, 1, 5, 7, 0, 2.6, 3 ]
2 sous_liste = une_liste[2:5]
3 print(sous_liste)
4 ## Affichera [5, 7, 0]
```

Dans l'exemple, on part de l'indice 2 et on s'arrête juste avant l'indice 5. On a donc les valeurs d'indice 2, 3 et 4. C'est le même comportement que `range(2,5)`.

On dispose aussi de raccourcis `liste[:5]` et `liste[3:]` pour prendre le début ou la fin de la liste.

```
1 une_liste = [ -1, 1, 5, 7, 0, 2.6, 3 ]
2
3 debut_liste = une_liste[:5]
4 print(debut_liste)
5 ## Affichera [-1, 1, 5, 7, 0]
6
7 fin_liste = une_liste[3:]
8 print(fin_liste)
9 ## Affichera [7, 0, 2.6, 3 ]
```

Ainsi, on a `liste == liste[:3] + liste[3:]`.

Compréhension de liste

C'est une des spécificités les plus puissantes de Python. Il est possible de définir une liste en intégrant une boucle `for` dans la définition. La définition se lit alors très simplement, quasiment comme en français.

```
1 une_liste = [1, 3, 4, 6, 9]
2
3 carres = [ x**2 for x in une_liste ]
4 print(carres)
5 ## Afficher [1, 9, 16, 36, 81]
```

On peut même sélectionner des valeurs en ajoutant une condition :

```
1 une_liste = [1, 3, 4, 6, 9]
2
3 carres = [ x**2 for x in une_liste if x%2==0 ]
4 print(carres)
5 ## Afficher [16, 36]
```

3.1.3 Listes de listes

Puisqu'on peut mettre n'importe quoi comme élément d'une liste, on peut fabriquer des listes de listes. Elles se manipulent aussi simplement que les listes normales.

```
1 coordonnees = [ [0,5] , [2,7] , [-1,2] ]
2
3 print( len(coordonnees) ) ## Afficher 3
4 print( coordonnees[1] ) ## Afficher [2,7]
5 print( coordonnees[0][1] ) ## Afficher 5
```

3.2 Les chaînes de caractères

On peut voir les chaînes de caractères comme des listes mais elles sont **immuables** : elles ne sont pas modifiables. Ce qui veut dire que l'utilisation des méthodes `append`, `extend`, `pop` et `remove` n'est pas possible. En revanche, le reste l'est.

```
1 chaine = "Hello world"
2
3 print( chaine[1] ) ## Afficher 'e'
4 print( len(chaine) ) ## Afficher 11
5
6 nouvelle = chaine + " !" ## <-- création d'une nouvelle chaîne
7 print(nouvelle) ## Afficher "Hello world !"
8
9 print( 'e' in chaine ) ## Afficher True
10
11 print( chaine[:5] ) ## Afficher "Hello"
12
13 chaine[4] = 'i' ## <-- ERREUR: tentative de modification
```

D'autre part, on peut vouloir répéter une chaîne de caractères :

```
1 beethoven = "Pom " * 4
2 print( beethoven )
3 ## Afficher 'Pom Pom Pom Pom'
```

3.3 Les Dictionnaires

Plutôt que d'indexer les valeurs d'une liste sur les entiers, on peut vouloir les indexer sur d'autres objets comme des chaînes de caractères. En informatique, ces objets sont appelés des **tableaux associatifs**, en Python, ce sont des **dictionnaires**.

Dans le cas de dictionnaires, les indices sont appelés des **clés**.

Un dictionnaire vide est créé avec `dico_vide = dict()` ou `dico_vide = {}`.

Pour la consultation, les dictionnaires s'utilisent exactement comme les listes. Pour l'affectation de nouvelles valeurs, c'est même plus simple.

```
1 personnage = { "nom": "Tintin", "age": 30, "métier": "journaliste" }
2
3 print( personnage )
4 ## Affichera { "métier": "journaliste", "age": 30, "nom": "Tintin" }
5
6 print( personnage["age"] )
7 ## Affichera 30
8
9 personnage["age"] = 32
10 personnage["père"] = "Hergé"
11 print( personnage )
12 ## Affichera { "métier": "journaliste", "père": "Hergé", "age": 32,
13 #             "nom": "Tintin" }
14
15 print( len(personnage) )
16 ## Affichera 4
```

Remarquez qu'une clé n'est pas obligatoirement une chaîne de caractères. On pourrait écrire `personnage[7] = 123`.

Remarquez aussi que l'ordre des éléments n'est pas nécessairement celui donné (ligne 1, 4 et 12). En fait, il n'y a pas d'ordre dans un dictionnaire. Au moment de l'affichage, c'est Python qui décide l'ordre des couples clé/valeur.

Pour observer le contenu du dictionnaire : on utilise les méthodes `keys` et `values` :

```
1 animal = { "espèce": "poule", "pattes": 2, "plumes": True }
2
3 print( animal )
4 ## Affichera { "plumes": True, "espèce": "poule", "pattes": 2 }
5
6 print( "plumes" in animal.keys() )
7 ## Affichera True
8 print( 2 in animal.keys() )
9 ## Affichera False
10
11 print( "plumes" in animal.values() )
12 ## Affichera False
13 print( 2 in animal.values() )
14 ## Affichera True
```

Notez bien que ces méthodes ne gardent pas l'association clé/valeur. Elles sont aussi utiles pour parcourir un dictionnaire dans une boucle, ainsi que `items` qui rassemble les deux :

```

1 animal = { "espèce": "poule", "pattes": 2, "plumes": True }
2
3 for cle in animal.keys():
4     print( cle, "=>", animal[cle] )
5     ## Affichera "'pattes' => 2" puis "'plumes': True" puis "'espèce': 'poule'"
6
7 for valeur in animal.values():
8     print( " valeur:", valeur )
9     ## Affichera "valeur: True" puis "valeur: 2" puis "valeur: 'poule'"
10
11 for cle,valeur in animal.items():
12     print( cle, "->", valeur )
13     ## Affichera "'espèce' -> 'poule'" puis "'pattes' -> 2" puis "'plumes' -> True"

```

Encore une fois, l'ordre n'a pas d'importance pour Python et ce n'est pas nécessairement le même d'une boucle à l'autre.

3.4 Quelques exercices

Exercice 11.

1. Écrire une fonction `plus_grand_dix` qui prend une liste d'entiers en entrée et qui renvoi le nombre d'éléments supérieurs ou égaux à 10 dans la liste.
2. Écrire une compréhension de liste qui donne la liste des 100 premiers carrés impairs.

Exercice 12.

1. Écrire une fonction `converti` qui prend une chaîne de caractères et qui retourne une nouvelle chaîne de caractères dans laquelle tous les caractères `'e'` ont été changés en `'o'`.
2. Écrire en moins d'une trentaine de caractères la chaîne

```
'<----->'
```

Exercice 13.

1. Écrire une fonction `inverse_dico` qui prend un dictionnaire en entrée et qui retourne un nouveau dictionnaire où le rôle des clés et des valeurs ont été échangés.
2. Que donne `inverse_dico(pattes)` où


```
pattes = {"vache": 4, "poule": 2, "mouton": 4, "serpent": 0} ?
```

