IngeSUP - Cours 11 - Structures de données avancées II

Sommaire

Introduction

Dans le chapitre précédent nous avons vu trois nouveaux types qui s'avèrent extrêmement utiles : les dictionnaires, les tuples et les ensembles. Comme les listes ou les chaînes de caractères, ces trois nouveaux types sont aussi appelés des **structures de données** ou des **containers**.

Containers

Définition

> 📝 **Définition**
>
> Un container est un nom générique pour définir un objet Python qui contient une collection d'autres objets

Les containers que nous connaissons depuis le début de ce cours sont les listes et les chaînes de caractères. Dans la section suivante, nous allons examiner les différentes propriétés des containers.

Propriétés

Examinons d'abord les propriétés qui caractérisent tous les types de container.

- Capacité à supporter le test d'appartenance.

Entrée [ ]:

```
l = [1, 2, 3]
1 in l
```

Entrée [ ]:

```
"to" in "toto"
```

- Capacité à supporter la fonction `len()` renvoyant la longueur du container.

Voici d'autres propriétés générales que nous avons déjà croisées. Un container peut être :

Ordonné (ordered en anglais) : il y a un ordre précis des éléments ; cet ordre correspond à celui utilisé lors de la création ou de la modification.

Indexable (subscriptable en anglais) : on peut retrouver un élément par son indice.

itérable (iterable en anglais) : on peut faire une boucle dessus.

Certains containers sont appelés objets séquentiels ou **séquence**.

> **Définition**
>
> Un **objet séquentiel** ou **séquence** est un container itérable, ordonné et indexable. Les objets séquentiels sont les listes, les chaînes de caractères, les objets de type range, ainsi que les tuples.

Une autre propriété importante que l'on a déjà croisée et qui nous servira dans ce chapitre concerne la possiblité ou non de modifier un objet.

Un objet est dit **non modifiable** lorsqu'on ne peut pas le modifier, ou lorsqu'on ne peut pas modifier un de ses éléments si c'est un container.

On parle aussi d'objet **immutable** (immutable object en anglais). Cela signifie qu'une fois créé, Python ne permet plus de le modifier par la suite.

> **Note**
>
> Qu'en est-il des objets que nous connaissons ? Les listes sont modifiables, on peut modifier un ou plusieurs de ses éléments. Tous les autres types que nous avons vus précédemment sont quant à eux non modifiables : les chaînes de caractères ou strings, les objets de type range, mais également des objets qui ne sont pas des containers comme les entiers, les floats et les booléens.

Containers de type range

Revenons rapidement sur les objets de type *range*. Jusqu'à maintenant, on s'en est servi pour faire des boucles ou générer des listes de nombres. Toutefois, on a vu ci-dessus qu'ils étaient aussi des containers.

Ils sont ordonnés, indexables, itérables, et non modifiables.

Entrée [ ]:

```
# Indexables
r = range(3)
r[0]
```

Entrée [ ]:

```
r[0:1]
```

Entrée [ ]:

```
# Iterables
for i in r:
    print(i)
```

Entrée [ ]:

```
# Non modifiables -> ERREUR
r[2] = 10
```

Rappel sur les dictionnaires

Les dictionnaires se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets (ceci est vrai jusqu'à la version 3.6 de Python, voir remarque ci-dessous).

Il ne s'agit pas d'objets séquentiels comme les listes ou chaînes de caractères, mais plutôt d'objets dits de correspondance (*mapping objects* en anglais) ou *tableaux associatifs*. En effet, on accède aux valeurs d'un dictionnaire par des clés.

Entrée [ ]:

```
animal1 = {}
animal1["nom"] = "girafe"
animal1["taille"] = 5.0
animal1["poids"] = 1100
print(animal1)
```

> **Note**
>
> Jusqu'à la version 3.6 de Python, un dictionnaire était affiché sans ordre particulier. L'ordre d'affichage des éléments n'était pas forcément le même que celui dans lequel il avait été rempli. De même lorsqu'on itérait dessus, l'ordre n'était pas garanti.
> Depuis Python 3.7 (inclus), ce comportement a changé, un dictionnaire est toujours affiché dans le même ordre que celui utilisé pour le remplir. De même, si on itère sur un dictionnaire, cet ordre est respecté.

On peut aussi initialiser toutes les clés et les valeurs d'un dictionnaire en une seule opération :

Entrée [ ]:

```
animal2 = {"nom": "singe", "taille": 1.75, "poids": 70}
```

Mais rien ne nous empêche d'ajouter une clé et une valeur supplémentaire :

Entrée [ ]:

```
animal2["age"] = 15
```

Pour récupérer la valeur associée à une clé donnée, il suffit d'utiliser la syntaxe suivante `dictionnaire["cle"]` . Par exemple :

Entrée [ ]:

```
print(animal1["taille"])
```

> **Information**
>
> Après ce premier tour d'horizon, on voit tout de suite l'avantage des dictionnaires. Pouvoir retrouver des éléments par des noms (clés) plutôt que par des indices. Les humains retiennent mieux les noms que les chiffres. Ainsi, l'usage des dictionnaires rend en général le code plus lisible.

> **Note**
>
> Par exemple, si nous souhaitions stocker les coordonnées d'un point dans l'espace :
>
> `coors = [0, 1, 2]` pour la version liste, `coors = {"x": 0, "y": 1, "z": 2}` pour la version dictionnaire.
>
> Un lecteur comprendra tout de suite que `coors["z"]` contient la coordonnée , ce sera moins intuitif avec `coors[2]` .

Liste de dictionnaires

En créant une liste de dictionnaires qui possèdent les mêmes clés, on obtient une structure qui ressemble à une base de données :

Entrée [ ]:

```
animaux = [animal1, animal2]
print(animaux)
```

Entrée [ ]:

```
for animal in animaux:
    print(animal["nom"])
```

Entrée [ ]:

```
for i in range(len(animaux)):
    print(animaux[i]["nom"])
```

Vous constatez ainsi que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes.

Exercices de TD

Vous pouvez maintenant vous exercer à partir du notebook [TD 11 (../TD/TD%2011%20-%20Structures%20de%20donn%C3%A9es%20avanc%C3%A9es%20II.ipynb)](../TD/TD%2011%20-%20Structures%20de%20donn%C3%A9es%20avanc%C3%A9es%20II.ipynb).