Initiation à Python - leçon 4.2

---
s1

Dans cette leçon nous allons découvrir une façon élégante de créer des listes ainsi que la notion d'itérateur.

---
s2

Les listes Python autorisent un mode de création élégant et concis : la compréhension. La compréhension signifie que les éléments de la liste sont le résultat d'un calcul.

Nous pouvons créer la liste des 10 premiers carrés de la façon suivante.

Ce qui se lit de la façon suivante : créer le nombre i au carré pour chaque i allant de 0 à 9

La même liste aurait pu être créée de cette façon moins concise.

Dans ce deuxième exemple, on voit que l'on peut introduire un test permettant de ne retenir qu'une partie des valeurs produites, comme ici uniquement les valeurs paires.

Ou encore cet exemple, où l'on produit une liste de tuples (x, y) dont les valeurs x et y doivent être différentes.

Je vous propose de reproduire ces exemples.

---
s3

Le même mécanisme peut être mis en oeuvre avec des sets et des tables, comme sur les deux exemples présentés que je vous propose de reproduire.

---
s4

Introduisons à présent les itérateurs.

On connaît la boucle for, qui permet de parcourir les différents éléments d’une liste, d’un tuple, d’une chaîne, ou même d’un dictionnaire ou d’un ensemble. Derrière cette syntaxe se cache en réalité un mécanisme un peu plus complexe.
Tout objet de type conteneur (une liste, une chaîne de caractères…) est capable (grâce à une des méthodes ou fonctions associées à cet objet) de créer un autre objet qu’on appelle itérateur et dont la seule fonction est de le parcourir.

Voyons cela sur un exemple.
On crée une liste
On crée l'objet itérateur associé à cette liste qui va permettre de la parcourir. On peut aussi appeler la fonction intrinsèque iter().
Au passage, on vérifie son type, ici un itérateur de liste
On commence à parcourir la liste grâce à la méthode next()
On continue à parcourir la liste ; on constate que l'itérateur garde en mémoire l'endroit où on s'est arrêté.
A la fin du parcours, il n'y a plus de valeur à afficher, next() provoque une exception

C'est précisément ce qui se passe - l'exception en moins - lorsque Python exécute une boucle for. On utilise ici le mot-clé pass qui signifie que dans le cas d'une exception de type StopIteration, l'exécution continue sans rien faire.

A vous de jouer.

---
s5

Quel est l'intérêt d'introduire les itérateurs ? On pourrait avoir besoin, par exemple, de parcourir le conteneur de droite à gauche (comme dans cet exemple, de la dernière lettre à la première). Or, la boucle for est un mécanisme de haut niveau qui ne peut être modifié.

Pour modifier la façon dont s'effectue le parcours, il faudrait redéfinir la méthode __iter__ de l'objet conteneur qui nous intéresse, l'objet string par exemple ; puis redéfinir l'itérateur associé à cet objet string. C'est assez lourd et cela dépasse le cadre de ce module.

Il existe un autre mécanisme, plus simple et plus intuitif permettant de modifier un itérateur : les générateurs.

---
s6

Les générateurs sont un moyen pratique de créer et manipuler des itérateurs. Voyons comment ils fonctionnent.

Un générateur est une fonction un peu particulière, capable de retourner une valeur différente à chaque appel grâce à une succession de mots-clés yield.

Voyons cela sur un premier exemple très simple.

On définit une fonction mon_generateur1 grâce au mot clé def. ici, au lieu d'utiliser le mot-clé return, on voit que l'on utilise plusieurs fois le mot-clé yield. C'est ce mot-clé qui fait la différence et qui fait que la fonction est un générateur. Ce générateur va simplement renvoyer 1, puis 2, puis 3.

L'objet mon_generateur1 est bien une fonction : une fonction n'est pas un objet itérable. Le résultat de cette fonction - à cause du mot-clé yield - est bien un générateur. Et ce générateur est itérable.

Comment l'utiliser ?
On crée l'objet itérateur associé à ce générateur. Cet itérateur va permettre de parcourir le contenu de la fonction mon_generateur1. Puis, comme dans la diapo précédente, on applique plusieurs fois à cet itérateur la fonction next.

Comme le générateur est itérable, on aurait tout aussi bien pu utiliser une boucle for.

Je vous propose de reproduire cet exemple.

---
s7

L'exemple de générateur précédent avait pour objet de vous faire comprendre le mécanisme. Voici un exemple un peu plus intéressant d'un générateur parcourant la série des entiers entre une borne_inf et une borne_sup.

Nous aurions pu ici obtenir le même résultat par un mécanisme de création de liste en compréhension.

---
s8

Voici enfin un exemple de générateur que je vous propose de reproduire et permettant de parcourir une chaîne dans l'ordre inverse de la boucle for.

Les générateurs présentent des fonctionnalités supplémentaires comme le fait d'accepter pendant le parcours des données extérieures de nature à modifier la façon dont se fait ce parcours. Ceci dépasse le cadre de ce module. Vous trouverez des explications supplémentaires sur la documentation officielle Python.

---
s9

Le prochain et dernier complément concerne l'encodage. Je vous propose à cette occasion de traverser un petit document PDF.