Python: Les boucles for imbriquées

Lorsqu'on programme, il arrive souvent qu'on doive traiter des données organisées en plusieurs dimensions. Par exemple, imaginez que vous deviez parcourir un tableau en deux dimensions (comme une grille ou un échiquier), générer une table de multiplication ou dessiner des motifs complexes comme une pyramide d'étoiles. Dans ces cas-là, une simple boucle ne suffit pas, car il faut répéter des actions sur plusieurs niveaux.

5TTR 6TTR Exploration

C'est ici que les **boucles imbriquées** deviennent indispensables. Elles permettent d'effectuer des actions à l'intérieur d'autres actions répétées, comme parcourir chaque cellule d'un tableau ligne par ligne, ou manipuler des ensembles de données structurées. Les boucles imbriquées sont donc la solution parfaite pour répondre à ces besoins de traitement multiple et de manipulation de données complexes.

> Autrement dit, dès que tu dois manipuler/traiter des données organisées en plusieurs dimensions ou à plusieurs niveaux, tu devras probablement utiliser plusieurs boucles imbriquées. Et tu utiliseras autant de boucles imbriquées qu'il y a de dimmensions/niveaux.

Dans cet article, nous allons explorer les boucles imbriquées en Python, comprendre leur utilité et illustrer leur fonctionnement avec des exemples concrets.

Introduction aux Boucles Imbriquées

Les boucles imbriquées se produisent lorsque vous placez une boucle à l'intérieur d'une autre. Cela vous permet de parcourir des données en deux dimensions/niveaux, ou de répéter un ensemble d'actions de manière itérative en fonction de plusieurs critères.

La structure d'une boucle imbriquée en Python est assez simple :

```
1  for i in range(3):  # Boucle extérieure
2      for j in range(3):  # Boucle intérieure
3          print(f"i = {i}, j = {j}")
```

Dans cet exemple, la boucle extérieure s'exécute trois fois, et à chaque itération de celle-ci, la boucle intérieure est exécutée trois fois à son tour. Le résultat est une série d'affichages où `i` prend les valeurs 0, 1, 2, et pour chaque `i`, `j` prend également les valeurs 0, 1, 2.

Résultat :

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
```

```
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
i = 2, j = 0
i = 2, j = 1
i = 2, j = 2
```

Exercices sur les Boucles Imbriquées

Voyons maintenant quelques exemples concrets d'utilisation des boucles imbriquées.

Afficher un carré de caractères

Ecris un programme qui permet d'afficher un carré de caractères (en utilisant toujours le même caractère).

Exemple pour un carré de 5x5:

```
*****
*****
*****
*****
*****
```

Afficher un carré de nombres aléatoires

Ecris un programme qui affiche un carré de nombres aléatoires.

Exemple pour un carré de 5x5:

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

Carré de Nombres Consécutifs

Les boucles imbriquées sont également utiles pour manipuler des données sous forme de grille ou de tableau. Par exemple, nous pouvons afficher un carré de nombres consécutifs.

```python
taille = int(input("Entrez la taille du carré : "))
compteur = 1

for i in range(taille):  # Boucle pour les lignes
    for j in range(taille):  # Boucle pour les colonnes
        print(f"{compteur:4}", end=" ")  # Affichage du nombre
        compteur += 1  # Incrémentation du compteur
    print()  # Changement de ligne
```

Résultat pour une taille de 4 :

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
```

Explication :

- La boucle extérieure parcourt les lignes du carré.
- La boucle intérieure parcourt les colonnes et affiche un nombre qui est incrémenté à chaque itération.
- L'utilisation de `f"{compteur:4}"` permet d'aligner les nombres correctement pour un affichage propre.

Triangle d'étoiles.

Ici vous allez afficher un triangle d'étoiles.

** Exemple de sortie pour une hauteur de 5**:

```
*
**
***
****
*****
```

Table de Multiplication

Un excellent exemple pour illustrer les boucles imbriquées est la création d'une table de multiplication. Dans cet exercice, nous allons afficher une table de multiplication allant de 1 à 10.

```python
for i in range(1, 11):  # Boucle pour les lignes
    for j in range(1, 11):  # Boucle pour les colonnes
        print(f"{i * j:4}", end=" ")  # Calcul du produit
    print()  # Changement de ligne après chaque ligne de la table
```

Résultat :

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Explication :

- La boucle extérieure contrôle la ligne de la table de multiplication.
- La boucle intérieure parcourt les colonnes et affiche le produit de `i` et `j`.
- Le `end=" "` dans la fonction `print()` empêche le saut de ligne automatique à la fin de chaque appel et aligne les résultats sur une seule ligne.

Damier (Échiquier) de Caractères

Enfin, un dernier exemple pour manipuler des boucles imbriquées consiste à créer un damier de caractères. Alternons les caractères `X` et `O` dans un damier de 8x8.

```python
for i in range(8):  # Boucle pour les lignes
    for j in range(8):  # Boucle pour les colonnes
        if (i + j) % 2 == 0:
            print("X", end=" ")
        else:
            print("O", end=" ")
    print()  # Changement de ligne
```

Résultat :

```
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
```

Explication :

- La boucle extérieure parcourt les lignes et la boucle intérieure les colonnes.
- À chaque position `(i, j)`, si la somme des indices est paire, un `X` est affiché, sinon c'est un `O`.

Damier (Échiquier) de Caractères - v2

Pour afficher un véritable damier sur la console, vous pouvez remplacer les '`O`' par des '`░`' (`alt-176`) et les '`X`' par des '`▓`' (`alt-178`).

Générateur d'adresses IP

Une adresse IP est composée de 4 nombres allant de 0 à 255, séparés par des points. Ex: `192.168.0.1`. Elle possèdent donc 4 'dimensions' ou niveaux. Crée un programme capable de générer toutes les combinaisons possibles, depuis `0.0.0.0` jusqu'à `255.255.255.255`.

Cela pourrait par exemple être utilisé pour scanner un réseau et détecter les équipements accessibles. Tu verras que ça prend du temp pour générer et afficher les 255^4 combinaisons possibles. Dans ce cadre, on essaie généralement de se limiter à des ***ranges*** plus restreints. On pourrait par exemple scanner toutes les adresses du range `192.168.*.*` ou encore `10.10.10.*` .

Attaque par force brute

Une attaque par force brute est une méthode pour deviner un mot de passe ou une clé en essayant toutes les combinaisons possibles jusqu'à ce que la bonne soit trouvée. Bien que cette méthode puisse être lente et inefficace face à des mots de passe longs ou complexes, elle reste une technique basique utilisée par les attaquants.

L'objectif d'une attaque par force brute est simple : **tester chaque combinaison possible d'un ensemble de caractères** jusqu'à trouver la bonne. Plus le mot de passe est long et plus l'ensemble de caractères est grand, plus l'attaque prendra du temps.

Pour générer toutes les combinaisons de caractères d'un mot de passe, on peut utiliser des boucles imbriquées. Imaginons que nous souhaitions tester toutes les combinaisons possibles d'un mot de passe composé de 4 caractères. Si les caractères disponibles sont [a-z], nous pouvons générer toutes les combinaisons possibles avec 4 boucles imbriquées.

Indice:

```python
import string
print(string.ascii_letters) # lettres de a-z en minuscules et majuscules (52
print(string.ascii_lowercase) # lettres de a-z en minuscules (26 possibilités)possibilités)
```

Générer les combinaisons

Crée un algorithme à base de boucles imbriquées qui permet de générer toutes les combinaisons de lettres possibles (de `a` à `z` en minuscules).

CHALLENGE en conditions "réelles"

Les mots de passe sont généralement cryptés avant d'être stocker sur un ordinateur. Une forme de cryptage qui était couramment utilisée est le hash md5. Il s'agit d'une fonction appliquée à un texte qui génère une clé "unique", le ***hash***, à partir de ce texte. Ce cryptage est irréversible. C'est à dire qu'il est impossible de retrouver mathématiquement le texte original depuis le hash généré.

Voici un exemple de fonction pour calculer le hash mld5 d'un texte en Python:

```python
import hashlib

def md5(texte):

    # Créer un objet MD5
    hash_md5 = hashlib.md5()

    # Mettre à jour l'objet MD5 avec le texte (encodé en bytes)
    hash_md5.update(texte.encode('utf-8'))

    # Récupérer la valeur du hash sous forme hexadécimale
    md5_hash = hash_md5.hexdigest()
```

```
13
14      return md5_hash
15
```

Ta misison: tu as récupéré une série de mots de passes *hashés* sur un serveur. Retrouve les mots de passe originaux (4 lettres) qui correspondent aux hash suivants:

- `881cc4157ed641a365a86452f27ed745`
- `40ea57d3ee3c07bf1c102b466e1c3091`
- `8d777f385d3dfec8815d20f7496026dc`
- `a7c3c2aa70d99921f9fb23ac87382997`
- `02c425157ecd32f259548b33402ff6d3`

Aller plus loin

Maintenant, allons plus loin:

- trouve un moyen de **mesurer le temps** que prend ton programme pour trouver le mot de passe (en millisecondes).
- **mesure** les temps d'exécution pour trouver les 5 mots de passe précédents.
- double le nombre de combinaisons possibles en utilisant des lettres minuscules et majuscules
- **mesure** les temps d'exécution pour trouver les 5 mots de passe précédents.
- teste avec ce hash: `3f85a60ff25237dc58f2ad63a1c75e78`
- maintenant, on refait la même chose avec **5** caractères (minuscules et majuscules). Trouves les mdp suivants:
  - `a85e7c8ba38d00af2f47681123ae4ec2`
  - `522a8fa409901e49937e4261e60d1713`
- ** Que remarques-tu au niveau du temps d'exécution, d'abord en ayant doublé le nombre de caractères et ensuite en ayant ajouté **1 seul** caractère?

> Ces exemples sont limités à des mots de passe d'une longueur fixe. Dans la pratique, les mots de passe peuvent être plus longs et l'ensemble des caractères beaucoup plus large (lettres majuscules, minuscules, chiffres, symboles, etc.). Pour générer des mots de passe de longueur variable, il est préférable d'utiliser une approche récursive ou de gérer dynamiquement les boucles.

3. Conclusion

Les boucles imbriquées sont des outils puissants pour effectuer des opérations répétitives sur des structures à plusieurs dimensions, comme des tableaux, des grilles ou des formes géométriques. Elles permettent d'automatiser et de simplifier des tâches complexes qui seraient fastidieuses à coder ligne par ligne.

Il est important de bien comprendre la logique des boucles imbriquées pour les utiliser efficacement et éviter les erreurs. Plus vous pratiquez avec des exemples comme ceux-ci, plus vous serez à l'aise avec leur utilisation dans des contextes variés.

N’hésitez pas à essayer ces exemples, et à les modifier pour explorer davantage le potentiel des boucles imbriquées dans vos propres projets Python.

© 2025 Mr. Lambrechts https://v2.ttrinfo.be 7 | 7