

Python 102



Plus loin avec Python



Squelette d'un module

- Même un fichier destiné à être utilisé comme un script doit être importable
- Un fichier important un module ne devrait subir d'effet de bord en exécutant les fonctionnalités du module importé
- En Python des outils comme **pydoc** ou ceux de tests unitaires ont besoin d'importer vos modules
- Votre code doit toujours vérifier `if __name__ == '__main__'` avant d'exécuter votre programme principal
- Ainsi, ce programme principal ne sera pas exécuté lorsque le module est importé
- [Un squelette de module](#)

Complément sur les listes

- Le type liste dispose de méthodes supplémentaires
- Voici la liste complète des méthodes des objets de type liste
 - **`list.append(x)`**
 - Ajoute un élément à la fin de la liste. Équivalent à `a[len(a):] = [x]`
 - **`list.extend(iterable)`**
 - Étend la liste en y ajoutant tous les éléments de l'iterable. Équivalent à `a[len(a):] = iterable`
 - **`list.insert(i, x)`**
 - Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`
 - **`list.remove(x)`**
 - Supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il existe aucun élément avec cette valeur

Complément sur les listes

➤ `list.pop([i])`

- Enlève de la liste l'élément situé à la position indiquée, et le renvoie en valeur de retour. Si aucune position n'est indiquée, `a.pop()` enlève et renvoie le dernier élément de la liste (les crochets autour du `i` dans la signature de la méthode indiquent bien que ce paramètre est facultatif, et non que vous devez placer des crochets dans votre code)

➤ `list.clear()`

- Supprime tous les éléments de la liste, équivalent à `del a[:]`

➤ `list.index(x[, start[, end]])`

- Renvoie la position du premier élément de la liste ayant la valeur `x` (en commençant par zéro). Une exception **`ValueError`** est levée si aucun élément n'est trouvé.
- Les arguments optionnels `start` et `end` sont interprétés de la même manière que dans la notation des tranches, et sont utilisés pour limiter la recherche à une sous-séquence particulière. L'index renvoyé est calculé relativement au début de la séquence complète, et non relativement à `start`

➤ `list.count(x)`

- Renvoie le nombre d'éléments ayant la valeur `x` dans la liste

Complément sur les listes

- `list.sort(key=None, reverse=False)`
 - Trie les éléments sur place, (les arguments peuvent personnaliser le tri, voir `sorted()` pour leur explication)
- `list.reverse()`
 - Inverse l'ordre des éléments de la liste, sur place
- `list.copy()`
 - Renvoie une copie superficielle de la liste. Équivalent à `a[:]`
- Un exemple utilisant la plupart de ces méthodes

Utiliser les listes comme des piles

- Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti », ou LIFO pour « last-in, first-out »)
- Pour ajouter un élément sur la pile, utilisez la méthode `append()`
- Pour récupérer l'objet au sommet de la pile, utilisez la méthode `pop()`, sans indicateur de position

```
lists-as-piles.py x
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4
5  def main():
6      """Lists as piles
7      """
8
9      # Create a stack
10     my_stack = [1, 2, 3, 4]
11     print("my_stack", my_stack)
12
13     # Push values on the stack
14     my_stack.append(5)
15     my_stack.append(6)
16     my_stack.append(7)
17     print("my_stack", my_stack)
18
19     # Pop values from the stack
20     print("Poped value", my_stack.pop())
21     print("my_stack", my_stack)
22     print("Poped value", my_stack.pop())
23     print("my_stack", my_stack)
24     print("Poped value", my_stack.pop())
25     print("my_stack", my_stack)
26     print("Poped value", my_stack.pop())
27     print("my_stack", my_stack)
28
29
30  if __name__ == '__main__':
31     main()
32
```


Utiliser les listes comme des files

- Il est également possible d'utiliser une liste comme une file, où le premier élément ajouté est le premier récupéré (« premier entré, premier sorti », ou FIFO pour « first-in, first-out »)
- Toutefois, les listes ne sont pas très efficaces pour ce type de traitement
- Alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertions ou de retraits en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).
- Pour implémenter une file, utilisez donc la classe **`collections.deque`** qui a été conçue pour fournir des opérations d'ajouts et de retraits rapides aux deux extrémités

```
1  #!/usr/bin/env python
2  #- coding: utf-8 -*-
3
4  from collections import deque
5
6
7  def main():
8      """Queues
9      """
10
11     # Create a queue
12     my_queue = deque([1, 2, 3])
13     print(my_queue)
14
15     # Append element to the right
16     my_queue.append(10)
17     print(my_queue)
18
19     # Append element to the left
20     my_queue.appendleft(100)
21     print(my_queue)
22
23     # Pop from the right
24     print("Deque", my_queue.pop())
25     print(my_queue)
26
27     # Pop from the left
28     print("Deque", my_queue.popleft())
29     print(my_queue)
30
31
32  if __name__ == '__main__':
33     #
34     # Call the main() function
35     #
36     main()
37
```

Compréhensions de listes

- Les compréhensions de listes fournissent un moyen de construire des listes de manière très concise
- Une application classique est la construction de nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque élément d'une autre séquence, ou de créer une sous-séquence des éléments satisfaisant une condition spécifique
- Par exemple, supposons que l'on veuille créer une liste de carrés
- Notez que cela crée (ou écrase) une variable nommée `x` qui existe toujours après l'exécution de la boucle
- On peut calculer une liste de carrés sans effet de bord, avec :

```
>>> carres = []  
>>> for x in range(10):  
...     carres.append(x**2)  
...  
>>> carres  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
>>>
```

```
>>> carres = [x**2 for x in range(10)]  
>>> carres  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
>>>
```


Compréhensions de listes

- Une compréhension de liste consiste en crochets contenant une expression suivie par une clause `for`, puis par zéro ou plus clauses `for` ou `if`
- Le résultat sera une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent
- Par exemple, cette compréhension de liste combine les éléments de deux listes s'ils ne sont pas égaux (Equivalent, en plus concis que la première solution)

```
>>> combines = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combines.append((x,y))
...
>>> combines
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>>
```

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Compréhensions de listes

- Les compréhensions de listes peuvent contenir des expressions complexes et des fonctions imbriquées

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
>>>
```

Compréhensions de listes imbriquées

- La première expression dans une compréhension de liste peut être n'importe quelle expression, y compris une autre compréhension de liste

```
>>> matrice = [  
...     [1,2,3,4],  
...     [5,6,7,8],  
...     [9,10,11,12]  
... ]  
>>> [[row[i] for row in matrice] for i in range(4)]  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]  
>>>
```

Compréhensions de listes imbriquées

- Les exemples suivants sont équivalents

```
>>> transpose = []
>>> for i in range(4):
...     transpose.append([row[i] for row in matrice])
...
>>> transpose
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

```
>>> transpose = []
>>> for i in range(4):
...     transposed_row = []
...     for row in matrice:
...         transposed_row.append(row[i])
...     transpose.append(transposed_row)
...
>>> transpose
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

Compréhensions de listes imbriquées

- Dans des cas concrets, il est toujours préférable d'utiliser des fonctions natives plutôt que des instructions de contrôle de flux complexes
- La fonction `zip()` ferait dans ce cas un excellent travail

```
>>> list(zip(*matrice))  
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]  
>>>
```


Références

- Python.org : <https://www.python.org/>
- Learning Python : <https://github.com/thierrydecker/learning-python>
- ...

Outils

- IDE Pycharm Community : <https://www.jetbrains.com/pycharm/>
- Analyse en ligne de code Python : <http://www.pythontutor.com/>
- ...