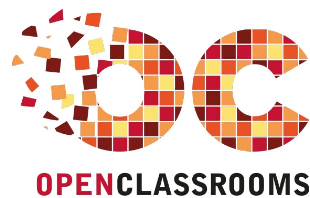
Utilisation avancée des listes en Python

Par Les Validateurs
et Matthieu Schaller (Nanoc)



www.openclassrooms.com

Licence Creative Commons 7 2.0
Dernière mise à jour le 29/01/2010

Sommaire

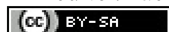
Sommaire ... 2
Utilisation avancée des listes en Python ... 3
Rappel des méthodes principales de list ... 3
list.append ... 3
list.extend ... 4
list.remove ... 4
list.reverse ... 4
list.sort ... 5
Les fonctions héritées du fonctionnel ... 5
map ... 6
filter ... 6
reduce ... 7
Les compréhensions de liste ... 7
Conversions list <=> string ... 8
Les ensembles ... 9
Appartenance ... 9
Réunion ... 9
Intersection ... 9
Différence ... 9
Différence symétrique ... 10
Optimisation des opérations sur les listes avec Numeric ... 10
Partager ... 11

#

Utilisation avancée des listes en Python

Par Matthieu Schaller (Nanoc) et

Les Validateurs

Mise à jour : 29/01/2010
Difficulté : Facile


La liste est la structure de donnée la plus utilisée en Python. Pour programmer correctement dans ce langage, il est donc crucial de savoir l'utiliser efficacement dans tous les cas. Le but de ce tutoriel est de vous apprendre certaines fonctions et techniques à utiliser avec les listes pour les manier de manière optimale dans vos programmes.

Bonne lecture .

Ce tutoriel a été écrit à l'origine par delroth et placé sous licence Creative Commons BY-SA.

Sommaire du tutoriel :

- Rappel des méthodes principales de list
- Les fonctions héritées du fonctionnel
- Les compréhensions de liste
- Conversions list <=> string
- Les ensembles
- Optimisation des opérations sur les listes avec Numeric

Rappel des méthodes principales de list

Commençons avec des choses faciles . Les listes possèdent plusieurs méthodes très utiles à leur manipulation, et qu'il est nécessaire de connaître pour utiliser ces structures de données efficacement. Pour obtenir une liste de toutes ces fonctions, rien ne vaut la documentation officielle de Python ou un petit help(list) dans la console Python .

list.append

Prototype :

Code : Python

```
list.append(element)
```

Comme son nom l'indique (si vous savez parler anglais, évidemment), cette fonction sert à ajouter un élément à la fin d'une liste. Elle est comme vous pouvez le deviner très utilisée.

Exemple d'utilisation :

Code : Python

```
li = [1, 2, 3, 4, 5]
li.append(6)
print li # [1, 2, 3, 4, 5, 6]
```

list.extend

Prototype :

Code : Python

```
list.extend(autre_liste)
```

Cette fonction est un peu la « petite sœur » de list.append : en effet, elle réalise le même rôle, si ce n'est qu'elle prend en paramètre une liste et qu'elle va ajouter à la fin de la liste tous les éléments de la liste passée en paramètre.

Exemple d'utilisation :

Code : Python

```
li = [1, 2, 3, 4, 5]
li.extend([6, 7, 8]) # Pour info, équivaut à li += [6, 7, 8]
print li # [1, 2, 3, 4, 5, 6, 7, 8]
```

list.remove

Prototype :

Code : Python

```
list.remove(valeur)
```

Encore une fois, la maîtrise de l'anglais est utile pour comprendre cette fonction . list.remove retire la première valeur trouvée qui est égale au paramètre. Cela permet de supprimer une valeur dont on ne veut pas dans la liste.

Exemple d'utilisation :

Code : Python

```
li = [1, 2, 3, 4, 5]
li.remove(1)
print li # [2, 3, 4, 5]
```

list.reverse

Prototype :

Code : Python

```
list.reverse()
```

Cette fonction permet de renverser l'ordre des éléments d'une liste, mais **sans renvoyer de valeur** : en effet, le renversement se fait directement dans la liste sur laquelle on a utilisé la méthode. Si on veut récupérer la valeur, on peut utiliser la fonction reversed(liste) (qui n'est pas une méthode de la classe list).

Exemple d'utilisation :

Code : Python

```
li = [1, 2, 3, 4, 5]
li.reverse()
print li # [5, 4, 3, 2, 1]
```

```python
print reversed(li) # [1, 2, 3, 4, 5]
print li # [5, 4, 3, 2, 1] <-- pas modifié
```

list.sort

Prototype :

Code : Python

```python
list.sort()
```

Cette fonction permet de trier une liste dans l'ordre croissant ou dans l'ordre alphabétique, selon le type des données de la liste. Des paramètres additionnels peuvent être donnés à la fonction pour changer le mode de tri et l'ordre dans lequel le tri s'effectue, je vous invite pour cela à lire la documentation Python de la fonction . De la même façon qu'il existe reversed(list) pour list.reverse(), on a sorted(list) pour list.sort() .

Exemple d'utilisation :

Code : Python

```python
li = [3, 1, 4, 2, 5]
li.sort()
print li # [1, 2, 3, 4, 5]

li = [3, 1, 4, 2, 5]
print sorted(li) # [1, 2, 3, 4, 5]
print li # [3, 1, 4, 2, 5] <-- pas modifié
```

Voilà, ce résumé des fonctions principales de la classe list est terminé, nous allons maintenant passer à des applications plus complexes sur les listes, avec les fonctions issues du paradigme fonctionnel : respectivement map, filter et reduce.

Les fonctions héritées du fonctionnel

Accrochez vous, car ce chapitre sera sûrement celui où vous apprendrez le plus de choses de ce tutoriel .

Tout d'abord, je parle depuis le début de fonctions issues du « fonctionnel ». Mais savez-vous tout simplement ce que cela signifie ? . En réalité, la caractéristique principale d'un langage fonctionnel est de souvent utiliser des procédés tels que la récursivité ou les fonctions *callback*. Pour la récursivité, le tutoriel de bluestorm vous en dira surement beaucoup plus que moi . Je vais surtout m'attarder sur le principe de fonction callback, très important pour ce chapitre.

Les fonctions callback, bien que le nom puisse faire penser à quelque chose de compliqué (ou pas), sont en fait des fonctions comme les autres. La seule différence est que ces fonctions seront utilisées comme argument d'une autre fonction (vous suivez toujours ?). Dans la pratique, en Python, rien ne permet de différencier une fonction callback d'une autre fonction, car elles s'utilisent exactement de la même manière. Un exemple de code parlera sûrement plus .

Code : Python

```python
def mettre_au_carre(x):
    return x ** 2 # x ** 2 = x puissance 2, pour les cerveaux lents
:-'

def appliquer_fonction(fonc, valeur):
    return fonc(valeur) # On utilise la fonction callback fonc avec
comme paramètre valeur.

print appliquer_fonction(mettre_au_carre, 3) # Affiche 9, c'est à
dire mettre_au_carre(3)
```

Comme vous voyez, on considère dans ce code mettre_au_carre comme une variable normale, alors que c'est une fonction. En vérité, contrairement à certains langages comme le PHP où l'on fait la différence entre fonctions et variables, le Python met les deux dans le même panier.

Voyons maintenant l'utilisation de ces fonctions callback pour les fonctions map, filter et reduce, issues de la programmation fonctionnelle.

map

Prototype :

Code : Python

```
map(callback, liste)
```

Équivalent à :

Code : Python

```
# [a, b, c] -> [callback(a), callback(b), callback(c)]
def map(callback, liste):
    nouvelle_liste = []
    for element in liste:
       nouvelle_liste.append(callback(element))
    return nouvelle_liste
```

La fonction map permet de transformer une liste via l'utilisation d'une fonction callback. Quelques exemples parleront sûrement plus qu'une longue explication :

Code : Python

```
def carre(x): return x ** 2
def pair(x): return not bool(x % 2)

print map(carre, [1, 2, 3, 4, 5]) # Affiche [1, 4, 9, 16, 25], c'est
à dire le carré de chaque élément

print map(pair, [1, 2, 3, 4, 5]) # Affiche [False, True, False,
True, False], c'est à dire si le nombre est pair.
```

Voilà pour la fonction map, passons maintenant à la deuxième : filter.

filter

Prototype :

Code : Python

```
filter(callback, liste)
```

Équivalent à :

Code : Python

```
# [a, b, c] -> [a si callback(a) == True, b si callback(b) == True,
c si callback(c) == True]
def filter(callback, liste):
    nouvelle_liste = []
    for element in liste:
       if callback(element): nouvelle_liste.append(element)
```

```
        return nouvelle_liste
```

La fonction filter ne permet pas réellement de « transformer » une liste, mais plutôt d'en retirer les valeurs que l'on ne veut pas. Encore une fois, des exemples pourraient être utiles .

Code : Python

```
def petit_carre(x): return x ** 2 < 16
def pair(x): return not bool(x % 2)

print filter(petit_carre, [1, 2, 3, 4, 5]) # Affiche [1, 2, 3],
c'est à dire les nombres dont les carrés sont inférieurs à 16.

print filter(pair, [1, 2, 3, 4, 5]) # Affiche [2, 4], c'est à dire
les nombres pairs de la liste.
```

Hop, c'est fini pour filter, voyons maintenant la plus difficile de toutes : reduce .

reduce

Prototype :

Code : Python

```
reduce(callback, liste, valeur_initiale)
```

Équivalent à :

Code : Python

```
# [a, b, c] -> callback(callback(a, b), c)
def reduce(callback, liste, valeur_initiale):
    if liste == []: return valeur_initiale
    else: return callback(liste[0], reduce(callback, liste[1:],
valeur_initiale))
```

Hum... En réalité, reduce est une fonction assez compliquée à comprendre, mais pourtant extrèmement puissante. Voyez plutôt les exemples .

Code : Python

```
def addition(x, y): return x + y
def produit(x, y): return x * y
def appcarre(element, liste): return liste + [element ** 2]

print reduce(addition, [1, 2, 3], 0) # Équivaut à ((0 + 1) + 2) + 3
print reduce(produit, [1, 2, 3], 1) # Équivaut à ((1 * 1) * 2) * 3
print reduce(appcarre, [1, 2, 3], []) # Équivaut à map(carre, [1, 2,
3]) ;)
```

reduce est assez... tordue quand on ne connaît pas . Elle est d'ailleurs assez peu utilisée, mais permet comme vous le voyez de très facilement coder les autres fonctions telles que map ou filter.

Les compréhensions de liste

Les compréhensions de liste (*list comprehensions* en anglais) sont des outils très puissants permettant d'utiliser map et filter (vues au dessus) avec une syntaxe plus proche de celle habituelle en Python. De plus, elles permettent de combiner un map et un filter en même temps .

Je vais vous montrer la syntaxe avec les exemples vus précédemment :

Code : Python

```
# Affiche les carrés des éléments
liste = [1, 2, 3, 4, 5, 6, 7]
print [x ** 2 for x in liste] # Équivaut à notre map, en plus
lisible et plus simple :) .

# Affiche les nombres pairs
print [x for x in liste if x % 2 == 0] # Plus simple que filter,
également :)

# Affiche les carrés pairs (combinaison des deux)
print [x ** 2 for x in liste if x ** 2 % 2 == 0] # ou
print [x for x in [a ** 2 for a in liste] if x % 2 == 0]
```

Cela permet de faire des choses assez jolies, dont nous verrons des exemples dans le chapitre suivant avec les conversions list <=> string.

Avant d'y aller, je tiens à préciser un point : la communauté pythonienne a tendance à préférer les list comprehensions aux fonctions telles que map ou filter, qui risquent d'ailleurs de disparaître d'un moment à l'autre .

Conversions list <=> string

Nous allons dans ce chapitre apprendre à convertir des list en string et des string en list. Ce sont des opérations assez courantes lorsqu'il s'agit d'afficher le contenu d'une liste ou de couper une chaîne en plusieurs parties.

Tout d'abord, attaquons nous à la conversion list -> string. La méthode utilisée pour cela est chaine.join(liste). Elle s'utilise de cette façon :

Code : Python

```
print ", ".join(["a", "b", "c", "d"]) # Affiche les éléments de la
liste avec ", " entre eux, c'est à dire "a, b, c, d".
```

Comme vous pouvez le voir, il n'y a vraiment aucune difficulté là dedans . Passons à la suite .

Pour convertir une chaîne en liste, on la découpe selon un séparateur que l'on a choisi, pour celà, on utilise chaine.split(separateur), qui s'utilise comme cela :

Code : Python

```
print "a b c d".split(" ") # Affiche [a, b, c, d]
```

On utilise souvent join avec les list comprehensions pour faire des affichages de plusieurs lignes en même temps. Par exemple, ce code permet d'afficher un tableau associant un joueur à un score :

Code : Python

```
joueurs = [
    (15, "delroth"),
    (2, "zozor"),
    (0, "vous")
]

joueurs.sort() # Il trie d'abord avec le premier élément (le
score), puis le deuxième (ordre alphabétique)

# Affichage du tableau, en une ligne ;)
print '\n'.join(["%s a %d points" % (joueur, score) for score,
joueur in joueurs])
```

Voilà un petit exemple de la puissance de ces fonctions . Passons maintenant aux fonctions permettant de travailler avec des ensembles, qui sont d'après moi les plus marrantes (mais pas forcément utiles par contre).

Les ensembles

Bon, pour cette partie, je vais supposer que vous êtes au moins en 2nde générale, car ça nécessite des notions de mathématiques que l'on voit dans cette classe là .

On va donc parler des ensembles et de leur représentation en Python. Pour créer un ensemble, on utilise un objet de type set (*ensemble* en anglais). Ensuite, cet objet a plein de méthodes cool à utiliser pour gérer les opérations avec les autres ensembles . On crée un objet de type set en lui passant en paramètre une liste d'éléments qui sont dans l'ensemble.

Code : Python

```
ens1 = set([1, 2, 3, 4, 5, 6])
ens2 = set([2, 3, 4])
ens3 = set([6, 7, 8, 9])
```

Comme vous le voyez, rien de compliqué là dedans . Passons maintenant à leur utilisation.

Appartenance

Notation mathématique : $A \subseteq B$
Notation Python :

Code : Python

```
a.issubset(b)
#ou
a & b == a
```

Réunion

Notation mathématique : $A \cup B$
Notation Python :

Code : Python

```
a.union(b)
#ou
a | b
```

Intersection

Notation mathématique : $A \cap B$
Notation Python :

Code : Python

```
a.intersect(b)
#ou
a & b
```

Différence

Notation mathématique : $A\backslash B$
Notation Python :
Code : Python

```
a.difference(b)
#ou
a - b
```

Différence symétrique

Notation mathématique : $A\Delta B$
Notation Python :
Code : Python

```
a.symmetric_difference(b)
#ou
a ^ b
```

Pour transformer un ensemble en liste, on utilise :
Code : Python

```
list(ensemble)
```

Voici quelques exemples :
Code : Python

```
# ens1 = set([1, 2, 3, 4, 5, 6])
# ens2 = set([2, 3, 4])
# ens3 = set([6, 7, 8, 9])
print ens1 & ens2 # set([2, 3, 4]) car ce sont les seuls à être en
même temps dans ens1 et ens2
print ens1 | ens3 # set([1, 2, 3, 4, 5, 6, 7, 8, 9]), les deux
réunis
print ens1 & ens3 # set([6]), même raison que deux lignes au dessus
print ens1 ^ ens3 # set([1, 2, 3, 4, 5, 7, 8, 9]), l'union moins les
éléments communs
print ens1 - ens2 # set([1, 5, 6]), on enlève les éléments de ens2
```

Voilà qui termine cette partie sur les ensembles, quelque chose de très peu connu en Python mais qui gagnerait sûrement à se faire connaître .

Optimisation des opérations sur les listes avec Numeric

Numeric est un module Python dédié à la gestion des calculs lourds. Nous allons ici l'utiliser pour réaliser des opérations sur des listes, plus particulièrement appelées *array* dans Numeric.

Tout d'abord, je vous laisse installer Numeric, ce tutoriel n'est pas là pour ça . Le site officiel du module est ici, et le nom du paquet Debian est python-numeric (pour les utilisateurs de *buntu ou Debian).

Voyons maintenant son utilisation. Pour créer une variable de type array, on utilise ceci :
Code : Python

```
arr = Numeric.array(liste)
```

Si ce code ne fonctionne pas, vérifiez d'avoir correctement importé le module Numeric.

Si on veut tout simplement un tableau rempli de zéros, on peut utiliser la fonction zeros(taille) :

Code : Python

```
arr = Numeric.zeros(taille)
```

Si on s'en fiche complètement de ce qu'il y a dedans à la création, on peut utiliser empty(taille), plus rapide que zeros(taille).

Code : Python

```
arr = Numeric.empty(taille)
```

Après cela, on peut utiliser notre tableau comme une liste Python : arr[i], arr[b:e], ... De plus, certains opérateurs sont surchargés pour les calculs mathématiques : ainsi, le + additionne deux tableaux contrairement à la version liste qui concatène les tableaux.

La vitesse de Numeric est un grand atout pour Python, qui permet de réaliser du traitement d'image ou des calculs lourds sans problème.
J'espère vous avoir montré dans ce tutoriel des choses que vous ne connaissiez pas, et vous avoir appris à les utiliser correctement et à bon escient :) . En effet, nombre de choses expliquées dans ce tutoriel sont très utilisées en Python, comme les compréhensions de liste ou les fonctions join et split, il est donc nécessaire de savoir les manier efficacement.

Partager

Ce tutoriel a été corrigé par les zCorrecteurs.