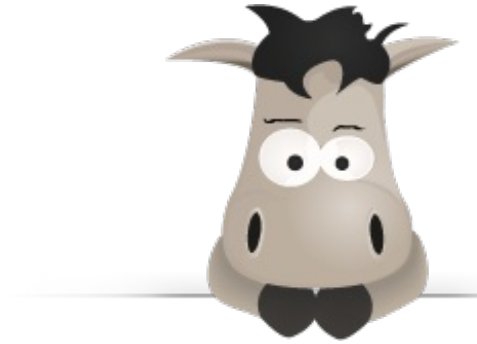

Pratiques avancées et méconnues en Python

Par delroth
et Natim

www.siteduzero.com

Dernière mise à jour le 25/01/2010

Sommaire

Sommaire 2
Pratiques avancées et méconnues en Python 3
La fonction enumerate 3
Les générateurs 4
Les coroutines 6
Les list comprehensions et generator expressions 8
La fonction zip 9
Le module itertools 10
imap, ifilter, ifilterfalse 11
chain 11
takewhile et dropwhile 11
groupby 12
Le mot clé with 12
Partager 13

Pratiques avancées et méconnues en Python

Par  Natim et  delroth

Mise à jour : 25/01/2010
Difficulté : Intermédiaire

Beaucoup de gens pensent maîtriser correctement le langage Python après quelques cours lus, et quelques exercices réalisés. Cette sensation vient de la simplicité apparente de Python pour un débutant. Pourtant, malgré ce que l'on pense, un code Python est relativement souvent améliorable sur de nombreux points, et c'est sur ces points que l'on distingue un codeur expérimenté d'un codeur débutant.

Ce tutoriel recense la plupart de ces points permettant de rendre son code plus lisible et plus *pythonnique* en utilisant des fonctionnalités méconnues du langage et en l'exploitant à 100 %.

De par sa nature, ce tutoriel ne vise clairement pas un public débutant en Python, mais plutôt un public ayant un peu d'expérience en Python et prêt à se documenter pour en savoir plus sur ce qui y est dit, car rien n'est aussi bien documenté que la documentation elle-même.

Bonne lecture à vous.
Sommaire du tutoriel :

- La fonction enumerate
- Les générateurs
- Les coroutines
- Les list comprehensions et generator expressions
- La fonction zip
- Le module itertools
- Le mot clé with

La fonction enumerate

Un besoin assez courant quand on manipule une liste ou tout autre objet itérable est de récupérer en même temps l'élément et son indice à chaque itération. Pour cela, la méthode habituellement utilisée est simple : au lieu d'itérer sur notre liste, on va itérer sur une liste d'entiers partant de 0 et allant de 1 en 1 jusqu'au dernier indice valide de la liste, obtenue via la fonction `xrange`. Cette méthode n'est pas du tout efficace : en effet, manipuler ainsi l'indice est totalement contre-intuitif et va à l'encontre du principe des itérateurs en Python.

Un exemple de code utilisant cette mauvaise méthode :

Code : Python

```
for indice in xrange(0, len(liste)):
    print "liste[%d] = %r" % (indice, liste[indice])
```

Pour réaliser ce genre d'itérations, on va utiliser la fonction `enumerate` : elle permet en effet de récupérer une liste de tuples `(indice, valeur)` en fonction du contenu de la liste, et d'une manière très pratique. On l'utilise comme ceci :

Code : Python

```
for indice, valeur in enumerate(liste):
    print "liste[%d] = %r" % (indice, valeur)
# ou sinon, comme enumerate renvoie une liste de tuples :
for indval in enumerate(liste):
    print "liste[%d] = %r" % indval
```

Cette manière de faire se rapproche beaucoup plus de ce qui doit être fait en Python si l'on veut utiliser correctement le langage : on itère directement sur les valeurs à la sortie d'un générateur, au lieu d'utiliser un indice (manière plus courante dans les langages comme le C n'ayant pas d'itérateurs comme ceux de Python).

Avec `enumerate`, on peut par exemple écrire très facilement un code numérotant les lignes d'un texte :

Code : Python

```
texte = file('nom-de-fichier').read()
for numberedline in enumerate(texte.split('\n')):
    print "%d\t%s" % numberedline
```

Au niveau des performances, l'utilisation de `enumerate` par rapport à des indices ne change strictement rien, avec une différence de temps d'exécution d'un centième de seconde sur un million d'itérations. Aucun argument n'est donc bon pour continuer à utiliser les indices dans ce genre de cas. 

Pour plus d'informations : PEP 279.

Les générateurs

Ce qu'on appelle en Python "*générateurs*" sont des objets que l'on crée d'une manière plutôt originale, en appelant une fonction que l'on crée soi-même et qui possède dans son code l'instruction `yield`. Cette fonction ne doit retourner aucune valeur.

Un exemple de code créant un générateur est le suivant :

Code : Python Console

```
>>> def generateur():
...     while True: yield 1
...
>>> generateur
<function generateur at 0x7f1c0bbe0aa0>
>>> type(generateur)
<type 'function'>
>>> generateur()
<generator object at 0x7f1c0bbf8710>
>>> type(generateur())
<type 'generator'>
```

Nous avons donc créé un *générateur*.
C'est beau, mais vous vous demandez sûrement à quoi ces générateurs servent.
Eh bien ce sont des objets **itérables**, c'est-à-dire qu'ils vont successivement renvoyer différentes valeurs.
Pour récupérer ces différentes valeurs, il y a plusieurs solutions :

- soit utiliser une boucle classique `for ... in ...:` ;
- soit utiliser la méthode `next` du générateur.

Pour bien comprendre, prenons l'exemple d'une fonction nous permettant, à l'instar de la fonction `xrange`, de nous donner une liste de nombres de 0 à y.

Code : Python

```
def range_perso(x, y=0):
    if y < x:      # Si y est inférieur à x on inverse ces deux
valeurs
        x, y  = y, x

    yield x        # On renvoie la première valeur de X
    while x < y:
        x += 1
        yield x    # On retourne x
```

On peut par exemple choisir d'utiliser notre nouvelle fonction / `generateur` `range_perso(x,y)` dans une boucle `for` :

Code : Python

```
for n in range_perso(10):
    print n
```

Le principe de fonctionnement d'un générateur est simple.

Quand on veut récupérer une valeur, il va commencer à exécuter le code de la fonction, jusqu'au moment où l'exécution rencontrera une instruction `yield`.
À ce moment-là, le code s'arrête, et on ressort de la fonction en ayant comme valeur la valeur donnée à l'instruction `yield` rencontrée.
Lorsque l'exécution atteint la fin de la fonction, le générateur envoie une exception de type `StopIteration`.
Cette exception est :

- soit à rattraper manuellement en cas d'utilisation de la méthode `next` du générateur ;
- soit signifie la fin de la boucle `for` dans le cas d'une itération par boucle `for ... in ...:`.

On peut ainsi coder des tonnes de fonctions très pratiques qui renverraient normalement par exemple des listes, en leur faisant renvoyer les valeurs une par une. Par exemple, ce générateur implémente la suite de Syracuse :

Code : Python

```
def syracuse(n):
    yield n
    while n != 1:
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3 * n + 1
        yield n

for n in syracuse(3):
    print n
```

Un générateur ne doit pas forcément s'arrêter.
Par exemple, ce générateur part d'un nombre x, et donne toutes les valeurs en partant de `x` : `x + 1`, `x + 2`, `x + 3`, etc.

Code : Python

```
def countfrom(x):
    while True:
        yield x
        x += 1

for n in countfrom(10):
    print n
    if n > 20: break # Si on ne break pas, on est en boucle infinie
                     # car le générateur ne s'arrête pas.
```

Avec les générateurs, on peut aussi recoder la fonction `enumerate` vue précédemment (qui est en réalité un générateur codé en C et intégré à la bibliothèque standard Python).

Code : Python

```
def enumerate(li):
    for i in xrange(0, len(li)):
        yield i, li[i]
```

La fonction `xrange` est, elle aussi, un générateur codé en C, mais que l'on peut très facilement programmer en Python :

Code : Python

```
def xrange(start, stop=None, step=1):
    if stop is None:
        start, stop = 0, start
    while step > 0 and start < stop or step < 0 and start > stop:
        yield start
        start += step
```

Pour plus d'informations : PEP 255

Les coroutines

Les coroutines sont des fonctions utilisant les générateurs de Python, et permettant de réaliser des choses impossibles autrement.
Leur première implémentation en Python a été réalisée par le projet Stackless Python, puis la PEP 342 a intégré tout ce qui est nécessaire à la création de coroutines à CPython, l'interpréteur Python que vous connaissez. 

Leur principale particularité est de pouvoir mettre en pause leur exécution en renvoyant une valeur (donc, comme un générateur). Pendant cette pause, elles peuvent également recevoir une valeur leur indiquant comment continuer leur exécution. Pour stopper l'exécution de notre coroutine, on va utiliser l'instruction `yield` vue précédemment pour les générateurs. La différence est que l'on pourra faire des choses comme ceci :

Code : Python

```
var = yield 42
```

Notre instruction `yield` *renvoie* donc une valeur qui lui est donnée pendant sa pause, et qui est par défaut `None`. Pour assigner une valeur au résultat de `yield`, on va utiliser la méthode `send` de notre générateur. On peut ainsi faire interagir notre générateur et le reste de notre programme très facilement, sans utiliser de variables globales.

Un exemple d'utilisation de la méthode `send` est celui d'un générateur qui compte de 0 à x, auquel on va pouvoir faire modifier la valeur actuelle du compteur, comme ceci :

Code : Python

```
def compteur(x):
    n = 0
    while n <= x:
        v = (yield n)   # Prenez l'habitude de mettre des parenthèses
                        # autour d'une instruction yield ;)
        if v is not None:
            n = v
        else:
            n += 1

gen = compteur(25)
```

```
for i in gen:
    print i
    # Si on en est à 15, on veut passer directement à 18
    if i == 15:
        gen.send(18)
```

Maintenant, voyons plus précisément les coroutines en elles-mêmes : ce sont deux fonctions qui vont interagir entre elles avec ce système de `yield` et `send`, et ainsi se communiquer des valeurs tout en gardant un contexte évoluant au fur et à mesure. On peut reprendre l'exemple de la PEP 342 en le simplifiant : un script écrivant les lignes d'un texte une par une dans des fichiers différents :

Code : Python

```
def ecrire_lignes_vers(combien, destination):
    """
Écrit `combien` lignes vers `destination`.
"""
    while True:
        lignes = [ ]
        plein = False
        commence = False
        try:
            for i in xrange(combien):
                commence = True
                lignes.append((yield))
            plein = True
        except GeneratorExit:
            # On a appelé la méthode close() de notre générateur
qui
            # le fait normalement s'arrêter ici. Si notre liste de
            # lignes n'est pas pleine, on force son écriture.
            if not plein and commence:
                destination.send(lignes)
            destination.close()
            return
        else:
            # Pas d'exceptions, ça roule, on envoie vers la
            # destination
            destination.send(lignes)

def ecrire_vers_fichiers():
    """
Écrit les lignes vers des fichiers numérotés.
"""
    i = 0
    while True:
        file('decoupe%04d.txt' % i, 'w').write('\n'.join((yield)))
        i += 1

def decouper_fichier_par(fichier, combien):
    destination = ecrire_vers_fichiers()
    ecrivain_lignes = ecrire_lignes_vers(combien, destination)

    # On initialise les générateurs pour éviter une exception
    destination.next()
    ecrivain_lignes.next()

    for ligne in file(fichier).read().split('\n'):
        ecrivain_lignes.send(ligne)
    ecrivain_lignes.close()
```

Cet exemple montre une jolie utilisation des coroutines pour couper un fichier en fichiers de n lignes numérotées dans l'ordre. Elles permettent également bien d'autres choses, qui sont mieux expliquées sur la page wikipédia anglaise des coroutines.

Pour plus d'informations : PEP 342.

Les list comprehensions et generator expressions

Les *list comprehensions* et *generator expressions* sont deux concepts très proches facilitant énormément la manipulation de listes et la création de générateurs simples.
Voyons tout d'abord les *list comprehensions*, plus faciles à aborder que les *generator expressions* que nous verrons ensuite.

Il arrive parfois que l'on doive faire le même traitement sur tous les éléments d'une liste, ou qu'on veuille éliminer tous les éléments d'une liste ne satisfaisant pas un prédicat. Pour cela, plusieurs solutions sont possibles :

- la boucle `for` : c'est une solution plutôt mauvaise car elle nécessite (si on veut faire quelque chose de correct) une copie de la liste ; elle ne permet pas de travailler directement sur nos éléments ;
- les méthodes `map/filter` et les itérateurs `imap/ifilter` : cette solution est clairement meilleure, du moins en termes de performances ; ces fonctions sont codées en C et travaillent directement sur les éléments de notre liste. Cependant, lorsque le traitement à réaliser n'est pas faisable par une fonction déjà créée, on doit créer cette fonction ou utiliser une fonction *lambda*, ralentissant ainsi l'exécution et réduisant la lisibilité ;
- les *list comprehensions* : elles permettent de récupérer une nouvelle liste avec tous les éléments de l'ancienne liste filtrés et sur lesquels on a effectué (ou non) un traitement. C'est à cette méthode, comme vous l'avez sûrement compris, que l'on va s'intéresser.

Une *list comprehension* a la syntaxe suivante :

Code : Python

```
[expression for element in liste if predicat]
```

Le prédicat étant optionnel, la syntaxe peut donc se réduire à ceci :

Code : Python

```
[expression for element in liste]
```

`expression` est une expression Python (donc par exemple un tuple, une liste, une fonction, un type, une *list comprehension*, ou même plus simplement une somme ou un produit), qui peut utiliser notre variable `element`. Cette *list comprehension* va créer une nouvelle liste avec, à la place de `element`, la valeur de `expression` qui peut, bien entendu, dépendre de `element` comme vous l'aurez sûrement compris.

Voyons maintenant quelques choses réalisables facilement avec les *list comprehensions* :

Code : Python Console

```
>>> liste = range(10) # On prend une liste contenant les 10 premiers entiers naturels
>>> liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [element + 1 for element in liste] # La même liste en ajoutant 1 à chaque élément
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [element ** 2 for element in liste] # On élève les éléments au carré
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> liste, liste == range(10) # La liste n'est pas modifiée ;)
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], True)
>>> liste = [element for element in liste if element % 2 == 0] # On prend tous les éléments pairs de notre liste
>>> liste
[0, 2, 4, 6, 8]
>>> [element ** 4 for element in liste if element < 7] # On met à la
```

```
puissance 4 les éléments inférieurs à 7
[0, 16, 256, 1296]
```

Maintenant que les *list comprehensions* n'ont plus de secrets pour vous, voyons leur petite soeur : les *generator expressions*. Ces expressions sont très proches des *list comprehensions* ; la seule différence est que, contrairement aux *list comprehensions* qui renvoient une liste, la *generator expression* renvoie un générateur sur lequel on peut itérer pour récupérer les éléments calculés. Cela permet de réaliser la transformation de nos éléments à la volée quand on itére sur notre générateur.

Leur syntaxe est, elle aussi, simplissime :

Code : Python

```
(expression for element in liste)
```

On peut également, comme pour les *list comprehensions*, rajouter une condition avec `if predicat` au bout.
La seule vraie différence est l'utilisation de parenthèses à la place des crochets. Cela permet d'ailleurs de l'utiliser avec une légère variante syntaxique lors de l'appel d'une fonction :

Code : Python

```
fonction(expression for element in liste)
```

Un petit exemple de leur utilisation est le suivant :

Code : Python Console

```
>>> liste = range(10)
>>> (elem - 5 for elem in liste)
<generator object at 0x7f004ec20878>
>>> gen = (elem - 5 for elem in liste)
>>> for elem in gen: print elem
...
-5
-4
-3
-2
-1
0
1
2
3
4
>>> list(elem - 5 for elem in liste) # Une autre façon d'écrire une
list comprehension ;)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Utilisez les *list comprehensions* et les *generator expressions* le plus souvent possible quand vous réalisez des traitements simples sur les éléments d'une liste : cela simplifie énormément la lecture de votre code par rapport à une boucle.

Pour plus d'informations, PEP 202 et PEP 289.

La fonction zip

La fonction `zip` est une fonction très simple nous venant des langages fonctionnels tels que Objective Caml ou Haskell, et permettant de combiner plusieurs listes en une seule, de manière à rendre les itérations plus efficaces.

En effet, imaginons que nous voulions itérer sur deux listes. La solution naïve venant à l'esprit est la suivante :

Code : Python

```
for i in xrange(min(len(liste1), len(liste2))):
    e1, e2 = liste1[i], liste2[i]
    # Actions en utilisant e1 et e2
```

Cependant, comme il a été dit dans la partie sur `enumerate`, les indices sont à éviter le plus possible dans ce genre de contexte. Pour réaliser cela, on a donc recours à la fameuse fonction `zip` qui nous permet de faire cette itération sur deux listes facilement :

Code : Python

```
for e1, e2 in zip(liste1, liste2):
    # Actions en utilisant e1 et e2
```

Ceci s'explique par la définition de `zip` que l'on peut trouver dans la documentation de Python.

Code : Autre

```
zip(...)
    zip(seq1 [, seq2 [...]]) -> [(seq1[0], seq2[0] ...), (...)]

    Return a list of tuples, where each tuple contains the i-th element
    from each of the argument sequences. The returned list is truncated
    in length to the length of the shortest argument sequence.
```

`zip` réalise donc la fonction suivante : pour une liste `[a, b, c]` et une autre liste `[d, e, f]`, `zip` nous donnera une liste `[(a, d), (b, e), (c, f)]`. En effet :

Code : Python Console

```
>>> zip(['a', 'b', 'c'], ['d', 'e', 'f'])
[('a', 'd'), ('b', 'e'), ('c', 'f')]
```

Pensez donc à `zip` quand vous itérerez sur plusieurs listes. 

En revanche et contrairement à d'autres langages, Python ne fournit pas de fonction `unzip` réalisant le rôle inverse de `zip`. Cependant, on peut très facilement exprimer la fonction `unzip` en fonction de `zip` :

Code : Python Console

```
>>> unzip = lambda liste: [list(li) for li in zip(*liste)]
>>> unzip([(1, 2), (3, 4), (5, 6)])
[[1, 3, 5], [2, 4, 6]]
```

Voilà qui conclut cette partie sur `zip`, une fonction très simple d'utilisation mais très pratique. 

Le module itertools

Le module `itertools` est un module standard Python proposant nombre de générateurs prêts à l'emploi dans votre code, vous permettant ainsi de réaliser des choses compliquées très facilement. Par exemple, des fonctions comme `map` ou `filter` renvoient des listes, alors que l'évaluation complète n'est pas forcément nécessaire. `itertools` pallie ce problème en fournissant des générateurs se comportant de la même manière. 

imap, ifilter, ifilterfalse

Ces trois fonctions, relativement proches les unes des autres, permettent de reproduire le fonctionnement des fonctions standard `map` et `filter` tout en utilisant des générateurs. On trouve également `ifilterfalse` qui a le comportement inverse de `ifilter` et prend donc uniquement les éléments ne satisfaisant pas au prédicat.

Code : Python Console

```
>>> from itertools import imap, ifilter, ifilterfalse
>>> pair = lambda n: n % 2 == 0
>>> carre = lambda n: n ** 2
>>> imap(carre, xrange(10)) # Renvoie un générateur
<itertools.imap object at 0x7f4ab174fc10>
>>> for elem in imap(carre, xrange(10)): print elem
...
0
1
4
9
16
25
36
49
64
81
>>> list(ifilter(pair, xrange(10)))
[0, 2, 4, 6, 8]
>>> list(ifilterfalse(pair, xrange(10)))
[1, 3, 5, 7, 9]
```

chain

`chain` est un générateur lui aussi très simple. Il prend en paramètre une liste d'objets itérables (comme des listes, par exemple), et itère sur chacun d'eux, dans l'ordre.

Par exemple, pour itérer sur les nombres de 0 à 10, puis 15 à 20, et enfin 30 à 45 :

Code : Python Console

```
>>> from itertools import chain
>>> list(chain(xrange(11), xrange(15, 21), xrange(30, 46)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16, 17, 18, 19, 20, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]
```

takewhile et dropwhile

Ces deux fonctions complémentaires permettent de prendre ou de rejeter des éléments tant qu'ils satisfont un prédicat. Une fois qu'un élément satisfait ce prédicat, selon la fonction, on jette tous les éléments ou on les prend tous. Si vous êtes un minimum capables de comprendre l'anglais, vous comprendrez très facilement l'exemple suivant :

Code : Python Console

```
>>> from itertools import takewhile, dropwhile
>>> list(takewhile(lambda i: i < 10, xrange(20))) # On prend tant
que i < 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(dropwhile(lambda i: i < 10, xrange(20))) # On jette tant
que i < 10
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

groupby

Ce générateur magique permet de regrouper les valeurs consécutives sortant d'un générateur en un tuple `(valeur, itérateur sur ces valeurs)`.
Un exemple de son utilisation est le suivant :

Code : Python Console

```
>>> from itertools import groupby
>>> liste = list('aaaaabbbcdddddee')
>>> for element, iterateur in groupby(liste):
...     # element est l'élément répété une ou plusieurs fois
...     # iterateur est un itérateur sur la répétition de cet
élément ; on l'utilise particulièrement pour savoir combien de fois
il est répété
...     print "%s est répété %d fois" % (element,
len(list(iterateur)))
...
a est répété 5 fois
b est répété 3 fois
c est répété 1 fois
d est répété 4 fois
e est répété 2 fois
```

On peut donc réaliser des choses plutôt marrantes avec cette fonction, comme par exemple implémenter très facilement l'algorithme de compression RLE :

Code : Python

```
from itertools import groupby
def rle(data):
    for value, it in groupby(data):
        numb = len(list(it))
        while numb > 255:
            yield "%s%s" % (chr(255), value)
            numb -= 255
        if numb != 0:
            yield "%s%s" % (chr(numb), value)
```

Le mot clé with

Peut-être avez-vous déjà travaillé sur des fichiers en Python, ou sur d'autres ressources nécessitant d'être fermées après utilisation.
Dans des cas critiques, si cette fermeture n'a pas lieu (à cause d'une erreur, par exemple), cela peut provoquer de gros problèmes dans le système. Pour se protéger contre ça, la solution utilisée la plupart du temps est la suivante :

Code : Python

```
f = open('fichier', 'w')
try:
    # Opérations sur f
except Exception:
    # Une erreur a eu lieu, on la gère (ou non)
finally:
    # On referme notre fichier
    f.close()
```

L'instruction `with` permet de beaucoup simplifier ce genre de codes en les transformant en ceci (qui réalise strictement la même chose) :

Code : Python

```
with open('fichier', 'w') as f:
    # Opérations sur f
```

Ce code réalise **vraiment** la même chose que les blocs `try` au-dessus, exceptée la possibilité de gérer nous-mêmes les erreurs. Cela passe par l'utilisation des *contexts managers*, des objets possédant deux méthodes magiques : `__enter__` et `__exit__` appelées à l'entrée ou à la sortie d'un bloc `with`.

Cette méthode n'est cependant disponible qu'avec des versions de Python récentes (au moins Python 2.6 ou au moins Python 3.0).

Pour plus d'informations : PEP 343
Ce tutoriel est terminé.
Nous espérons vous y avoir appris pas mal de choses nouvelles sur le langage Python. Ces notions sont inconnues des débutants Python mais devraient être utilisées par des codeurs expérimentés.

Tutoriel propulsé par **mdown** et **tutosuite**, merci aux auteurs de ces deux outils.
Merci à **Tarek Ziadé** pour son *lightning talk* au PyCon 2008 m'ayant inspiré ce tutoriel.

Merci à toutes les personnes qui ont bien voulu relire et corriger ce tutoriel.

Ce tutoriel est issu d'une réflexion ayant eu lieu sur :

Ce tutoriel a été corrigé par les zCorrecteurs.