Guide pratique : programmation fonctionnelle

Version 3.6.15

Guido van Rossum
and the Python development team

septembre 05, 2021

Python Software Foundation
Email : docs@python.org

Table des matières

Auteur A. M. Kuchling

Version 0.32

Dans ce document, nous allons faire un tour des fonctionnalités de Python adaptées à la réalisation d'un programme dans un style fonctionnel. Après une introduction à la programmation fonctionnelle, nous aborderons des outils tels que les iterators et les generators ainsi que les modules `itertools` et `functools`.

1 Introduction

Cette section détaille les fondamentaux de la programmation fonctionnelle. Si seules les fonctionnalités de Python vous intéressent, vous pouvez sauter cette partie et lire la section suivante sur les *Itérateurs*.

Les langages de programmation permettent de traiter des problèmes selon différentes approches :

— La plupart des langages de programmation suivent une logique **procédurale** : les programmes sont constitués de listes d'instructions qui détaillent les opérations que l'ordinateur doit appliquer aux entrées du programme. C, Pascal ou encore les interpréteurs de commandes Unix sont des langages procéduraux.

— Les langages **déclaratifs** permettent d'écrire la spécification du problème et laissent l'implémentation du langage trouver une façon efficace de réaliser les calculs nécessaires à sa résolution. SQL est un langage déclaratif que vous êtes susceptible de connaître ; une requête SQL décrit le jeu de données que vous souhaitez récupérer et le moteur SQL choisit de parcourir les tables ou d'utiliser les index, l'ordre de résolution des sous-clauses, etc.

— Les programmes **orientés objet** manipulent des ensembles d'objets. Ceux-ci possèdent un état interne et des méthodes qui interrogent ou modifient cet état d'une façon ou d'une autre. Smalltalk et Java sont deux langages orientés objet. C++ et Python gèrent la programmation orientée objet mais n'imposent pas l'utilisation de telles fonctionnalités.

— La programmation **fonctionnelle** implique de décomposer un problème en un ensemble de fonctions. Dans l'idéal, les fonctions produisent des sorties à partir d'entrées et ne possède pas d'état interne qui soit susceptible de modifier la sortie pour une entrée donnée. Les langages fonctionnels les plus connus sont ceux de la famille ML (Standard ML, OCaml et autres) et Haskell.

Les personnes qui conçoivent des langages de programmation peuvent choisir de privilégier une approche par rapport à une autre. Cela complexifie l'écriture de programmes appliquant un paradigme différent de celui considéré. Certains langages sont multi-paradigmes et gère plusieurs approches différentes. Lisp, C++ et Python sont de tels langages ; vous pouvez écrire des programmes ou des bibliothèques dans un style procédural, orienté objet ou fonctionnel dans chacun d'entre eux. Différentes parties d'une application peuvent être écrites selon des approches différentes ; par exemple, l'interface graphique peut suivre le paradigme orienté objet tandis que la logique de traitement est procédurale ou fonctionnelle.

Dans un programme fonctionnel, l'entrée traverse un ensemble de fonctions. Chaque fonction opère sur son entrée et produit une sortie. Le style fonctionnel préconise de ne pas écrire de fonctions ayant des effets de bord, c'est-à-dire qui modifient un état interne ou réalisent d'autres changements qui ne sont pas visibles dans la valeur de sortie de la fonction. Les fonctions qui ne présentent aucun effet de bord sont dites **purement fonctionnelles**. L'interdiction des effets de bord signifie qu'aucune structure de données n'est mise à jour lors de l'exécution du programme ; chaque sortie d'une fonction ne dépend que de son entrée.

Certains langages sont très stricts en ce qui concerne la pureté des fonctions et ne laissent même pas la possibilité d'assigner des variables avec des expressions telles que `a = 3` ou `c = a + b`, cependant il est difficile d'éviter tous les effets de

bord. Afficher un message sur l'écran ou écrire un fichier sur le disque sont des effets de bord. Par exemple, un appel aux fonctions `print()` ou `time.sleep()` en Python ne renvoie aucune valeur utile ; ces fonctions ne sont appelées que pour leur effet de bord (afficher du texte sur l'écran et mettre en pause l'exécution du programme).

Les programmes Python écrits dans un style fonctionnel ne poussent généralement pas le curseur de la pureté à l'extrême en interdisant toute entrée/sortie ou les assignations ; ils exhibent une interface fonctionnelle en apparence mais utilisent des fonctionnalités impures en interne. Par exemple, l'implémentation d'une fonction peut assigner dans des variables locales mais ne modifiera pas de variable globale et n'aura pas d'autre effet de bord.

La programmation fonctionnelle peut être considérée comme l'opposé de la programmation orientée objet. Les objets encapsulent un état interne ainsi qu'une collection de méthodes qui permettent de modifier cet état. Les programmes consistent à appliquer les bons changements à ces états. La programmation fonctionnelle vous impose d'éviter au maximum ces changements d'états en travaillant sur des données qui traversent un flux de fonctions. En Python, vous pouvez combiner ces deux approches en écrivant des fonctions qui prennent en argument et renvoient des instances représentants des objets de votre application (courriers électroniques, transactions, etc.).

Programmer sous la contrainte du paradigme fonctionnel peut sembler étrange. Pourquoi vouloir éviter les objets et les effets de bord ? Il existe des avantages théoriques et pratiques au style fonctionnel :

— Preuves formelles.
— Modularité.
— Composabilité.
— Facilité de débogage et de test.

1.1 Preuves formelles

Un avantage théorique est qu'il plus facile de construire une preuve mathématique de l'exactitude d'un programme fonctionnel.

Les chercheurs ont longtemps souhaité trouver des façons de prouver mathématiquement qu'un programme est correct. Cela ne se borne pas à tester si la sortie d'un programme est correcte sur de nombreuses entrées ou lire le code source et en conclure que le celui-ci semble juste. L'objectif est d'établir une preuve rigoureuse que le programme produit le bon résultat pour toutes les entrées possibles.

La technique utilisée pour prouver l'exactitude d'un programme est d'écrire des **invariants**, c'est-à-dire des propriétés de l'entrée et des variables du programme qui sont toujours vérifiées. Pour chaque ligne de code, il suffit de montrer que si les invariants X et Y sont vrais **avant** l'exécution de cette ligne, les invariants légèrement modifiés X" et Y" sont vérifiés **après** son exécution. Ceci se répète jusqu'à atteindre la fin du programme. À ce stade, les invariants doivent alors correspondre aux propriétés que l'on souhaite que la sortie du programme vérifie.

L'aversion du style fonctionnel envers les assignations de variable est apparue car celles-ci sont difficiles à gérer avec cette méthode. Les assignations peuvent rompre des invariants qui étaient vrais auparavant sans pour autant produire de nouveaux invariants qui pourraient être propagés.

Malheureusement, prouver l'exactitude d'un programme est très peu commode et ne concerne que rarement des logiciels en Python. Même des programmes triviaux nécessitent souvent des preuves s'étalant sur plusieurs pages ; la preuve de l'exactitude d'un programme relativement gros serait gigantesque. Peu, voire aucun, des programmes que vous utilisez quotidiennement (l'interpréteur Python, votre analyseur syntaxique XML, votre navigateur web) ne peuvent être prouvés exacts. Même si vous écriviez ou généreriez une preuve, il faudrait encore vérifier celle-ci. Peut-être qu'elle contient une erreur et que vous pensez désormais, à tort, que vous avez prouvé que votre programme est correct.

1.2 Modularité

Un intérêt plus pratique de la programmation fonctionnelle est qu'elle impose de décomposer le problème en petits morceaux. Les programmes qui en résultent sont souvent plus modulaires. Il est plus simple de spécifier et d'écrire une petite fonction qui ne fait qu'une seule tâche plutôt qu'une grosse fonction qui réalise une transformation complexe. Les petites fonctions sont plus faciles à lire et à vérifier.

1.3 Facilité de débogage et de test

Tester et déboguer un programme fonctionnel est plus facile.

Déboguer est plus simple car les fonctions sont généralement petites et bien spécifiées. Lorsqu'un programme ne fonctionne pas, chaque fonction constitue une étape intermédiaire au niveau de laquelle vous pouvez vérifier que les valeurs sont justes. Vous pouvez observer les entrées intermédiaires et les sorties afin d'isoler rapidement la fonction qui est à l'origine du bogue.

Les tests sont plus faciles car chaque fonction est désormais un sujet potentiel pour un test unitaire. Les fonctions ne dépendent pas d'un état particulier du système qui devrait être répliqué avant d'exécuter un test ; à la place vous n'avez qu'à produire une entrée synthétique et vérifier que le résultat correspond à ce que vous attendez.

1.4 Composabilité

En travaillant sur un programme dans le style fonctionnel, vous écrivez un certain nombre de fonctions avec des entrées et des sorties variables. Certaines de ces fonctions sont inévitablement spécifiques à une application en particulier, mais d'autres peuvent s'appliquer à de nombreux cas d'usage. Par exemple, une fonction qui liste l'ensemble des fichiers XML d'un répertoire à partir du chemin de celui-ci ou une fonction qui renvoie le contenu d'un fichier à partir de son nom peuvent être utiles dans de nombreuses situations.

Au fur et à mesure, vous constituez ainsi une bibliothèque personnelle d'utilitaires. Souvent, vous pourrez construire de nouveaux programmes en agençant des fonctions existantes dans une nouvelle configuration et en écrivant quelques fonctions spécifiques à votre objectif en cours.

2 Itérateurs

Commençons par jeter un œil à une des fonctionnalités les plus importantes pour écrire en style fonctionnel avec Python : les itérateurs.

Un itérateur est un objet qui représente un flux de données ; cet objet renvoie les données un élément à la fois. Un itérateur Python doit posséder une méthode `__next__()` qui ne prend pas d'argument et renvoie toujours l'élément suivant du flux. S'il n'y plus d'élément dans le flux, `__next__()` doit lever une exception `StopIteration`. Toutefois, ce n'est pas indispensable ; il est envisageable d'écrire un itérateur qui produit un flux infini de données.

La fonction native `iter()` prend un objet arbitraire et tente de construire un itérateur qui renvoie le contenu de l'objet (ou ses éléments) en levant une exception `TypeError` si l'objet ne gère pas l'itération. Plusieurs types de données natifs à Python gèrent l'itération, notamment les listes et les dictionnaires. On appelle iterable un objet pour lequel il est possible de construire un itérateur.

Vous pouvez expérimenter avec l'interface d'itération manuellement :

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> it.__next__()  # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python s'attend à travailler sur des objets itérables dans divers contextes et tout particulièrement dans une boucle `for`. Dans l'expression `for X in Y`, Y doit être un itérateur ou un objet pour lequel `iter()` peut générer un itérateur. Ces deux expressions sont équivalentes :

```
for i in iter(obj):
    print(i)

for i in obj:
    print(i)
```

Les itérateurs peuvent être transformés en listes ou en tuples en appelant les constructeurs respectifs `list()` et `tuple()` :

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Le dépaquetage de séquences fonctionne également sur les itérateurs : si vous savez qu'un itérateur renvoie N éléments, vous pouvez les dépaqueter dans un n-uplet :

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

Certaines fonctions natives telles que `max()` et `min()` prennent un itérateur en argument et en renvoie le plus grand ou le plus petit élément. Les opérateurs `"in"` et `"not in"` gèrent également les itérateurs : `X in iterator` est vrai si X a été trouvé dans le flux renvoyé par l'itérateur. Vous rencontrerez bien sûr des problèmes si l'itérateur est infini : `max()`, `min()` ne termineront jamais et, si l'élément X n'apparaît pas dans le flux, les opérateurs `"in"` et `"not in"` non plus.

Notez qu'il n'est possible de parcourir un itérateur que vers l'avant et qu'il est impossible de récupérer l'élément précédent, de réinitialiser l'itérateur ou d'en créer une copie. Des objets itérateurs peuvent offrir ces possibilités de façon facultative, mais le protocole d'itération ne spécifie que la méthode `__next__()`. Certaines fonctions peuvent ainsi consommer l'entièreté de la sortie d'un itérateur et, si vous devez utiliser le même flux pour autre chose, vous devrez en créer un nouveau.

2.1 Types de données itérables

Nous avons vu précédemment comment les listes et les *tuples* gèrent les itérateurs. En réalité, n'importe quel type de séquence en Python, par exemple les chaînes de caractères, sont itérables.

Appeler `iter()` sur un dictionnaire renvoie un itérateur qui parcourt l'ensemble de ses clés :

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Mar 3
Feb 2
Aug 8
Sep 9
Apr 4
Jun 6
Jul 7
Jan 1
May 5
Nov 11
Dec 12
Oct 10
```

Note that the order is essentially random, because it's based on the hash ordering of the objects in the dictionary.

Appliquer `iter()` sur un dictionnaire produit un itérateur sur ses clés mais il est possible d'obtenir d'autres itérateurs par d'autres méthodes. Si vous souhaitez itérer sur les valeurs ou les paires clé/valeur du dictionnaire, vous pouvez explicitement appeler les méthodes `values()` ou `items()` pour obtenir l'itérateur idoine.

Le constructeur `dict()` accepte de prendre un itérateur en argument qui renvoie un flux fini de pairs `(clé, valeur)` :

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}
```

Les fichiers gèrent aussi l'itération en appelant la méthode `readline()` jusqu'à ce qu'il n'y ait plus d'autre ligne dans le fichier. Cela signifie que vous pouvez lire l'intégralité d'un fichier de la façon suivante :

```
for line in file:
    # do something for each line
    ...
```

Les ensembles peuvent être créés à partir d'un itérable et autorisent l'itération sur les éléments de l'ensemble :

```
S = {2, 3, 5, 7, 11, 13}
for i in S:
    print(i)
```

3 Expressions génératrices et compréhension de listes

Deux opérations courantes réalisables sur la sortie d'un itérateur sont 1) effectuer une opération pour chaque élément, 2) extraire le sous-ensemble des éléments qui vérifient une certaine condition. Par exemple, pour une liste de chaînes de caractères, vous pouvez choisir de retirer tous les caractères blancs à la fin de chaque ligne ou extraire toutes les chaînes contenant une sous-chaîne précise.

Les compréhensions de listes et les expressions génératrices sont des façons concises d'exprimer de telles opérations, inspirées du langage de programmation fonctionnel Haskell (https://www.haskell.org/). Vous pouvez retirer tous les caractères blancs initiaux et finaux d'un flux de chaînes de caractères à l'aide du code suivant :

```
line_list = ['  line 1\n', 'line 2  \n', ...]

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

Vous pouvez ne sélectionner que certains éléments en ajoutant une condition « `if` » :

```
stripped_list = [line.strip() for line in line_list
                 if line != ""]
```

La compréhension de liste renvoie une liste Python ; `stripped_list` est une liste contenant les lignes après transformation, pas un itérateur. Les expressions génératrices renvoient un itérateur qui calcule les valeurs au fur et à mesure sans toutes les matérialiser d'un seul coup. Cela signifie que les compréhensions de listes ne sont pas très utiles si vous travaillez sur des itérateurs infinis ou produisant une très grande quantité de données. Les expressions génératrices sont préférables dans ce cas.

Les expressions génératrices sont écrites entre parenthèses (« () ») et les compréhensions de listes entre crochets (« [] »). Les expressions génératrices sont de la forme :

```
( expression for expr in sequence1
             if condition1
             for expr2 in sequence2
             if condition2
             for expr3 in sequence3 ...
             if condition3
             for exprN in sequenceN
             if conditionN )
```

La compréhension de liste équivalente s'écrit de la même manière, utilisez juste des crochets à la place des parenthèses.

Les éléments de la sortie sont les valeurs successives de `expression`. La clause `if` est facultative ; si elle est présente, `expression` n'est évaluée et ajoutée au résultat que si `condition` est vérifiée.

Les expressions génératrices doivent toujours être écrites entre parenthèses, mais les parenthèses qui encadrent un appel de fonction comptent aussi. Si vous souhaitez créer un itérateur qui soit immédiatement passé à une fonction, vous pouvez écrire :

```
obj_total = sum(obj.count for obj in list_all_objects())
```

Les clauses `for ... in` indiquent les séquences sur lesquelles itérer. Celles-ci peuvent être de longueurs différentes car l'itération est réalisée de gauche à droite et non en parallèle. `sequence2` est parcourue entièrement pour chaque élément de `sequence1`. `sequence3` est ensuite parcourue dans son intégralité pour chaque paire d'éléments de `sequence1` et `sequence2`.

Autrement dit, une compréhension de liste ou une expression génératrice est équivalente au code Python ci-dessous :

```
for expr1 in sequence1:
    if not (condition1):
        continue   # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue   # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue   # Skip this element

            # Output the value of
            # the expression.
```

Ainsi lorsque plusieurs clauses `for ... in` sont présentes mais sans condition `if`, la longueur totale de la nouvelle séquence est égale au produit des longueurs des séquences itérées. Si vous travaillez sur deux listes de longueur 3, la sortie contiendra 9 éléments :

```
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

Afin de ne pas créer une ambiguïté dans la grammaire de Python, `expression` doit être encadrée par des parenthèses si elle produit un n-uplet. La première compréhension de liste ci-dessous n'est pas valide syntaxiquement, tandis que la seconde l'est :

```
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```

4 Générateurs

Les générateurs forment une classe spéciale de fonctions qui simplifie la création d'itérateurs. Les fonctions habituelles calculent une valeur et la renvoie, tandis que les générateurs renvoient un itérateur qui produit un flux de valeurs.

Vous connaissez sans doute le fonctionnement des appels de fonctions en Python ou en C. Lorsqu'une fonction est appelée, un espace de nommage privé lui est associé pour ses variables locales. Lorsque le programme atteint une instruction `return`, les variables locales sont détruites et la valeur est renvoyée à l'appelant. Les appels postérieurs à la même fonction créent un nouvel espace de nommage privé et de nouvelles variables locales. Cependant, que se passerait-il si les variables locales n'étaient pas détruites lors de la sortie d'une fonction ? Et s'il était possible de reprendre l'exécution de la fonction là où elle s'était arrêtée ? Les générateurs sont une réponse à ces questions ; vous pouvez considérer qu'il s'agit de fonctions qu'il est possible d'interrompre, puis de relancer sans perdre leur progression.

Voici un exemple simple de fonction génératrice :

```
>>> def generate_ints(N):
...     for i in range(N):
...         yield i
```

N'importe quelle fonction contenant le mot-clé `yield` est un générateur ; le compilateur bytecode de Python détecte ce mot-clé et prend en compte cette particularité de la fonction.

Lorsque vous appelez une fonction génératrice, celle-ci ne renvoie pas une unique valeur ; elle renvoie un objet générateur qui implémente le protocole d'itération. Lorsque l'expression `yield` est exécutée, le générateur renvoie la valeur de `i`, d'une façon similaire à un `return`. La différence principale entre `yield` et `return` est qu'en atteignant l'instruction `yield`, l'état du générateur est suspendu et les variables locales sont conservées. Lors de l'appel suivant à la méthode `__next__()` du générateur, la fonction reprend son exécution.

Voici un exemple d'utilisation du générateur `generate_ints()` :

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a, b, c = generate_ints(3)`.

Dans une fonction génératrice, une instruction `return value` entraine la levée d'une exception `StopIteration(value)` dans la méthode `__next__()`. Lorsque cela se produit (ou que la fin de la fonction est atteinte), le flot de nouvelles valeurs s'arrête et le générateur ne peut plus rien produire.

Vous pouvez obtenir le même comportement que celui des générateurs en écrivant votre propre classe qui stocke les variables locales du générateur comme variables d'instance. Pour renvoyer une liste d'entiers, par exemple, vous pouvez initialiser `self.count` à 0 et écrire la méthode `__next__()` de telle sorte qu'elle incrémente `self.count` puis le renvoie. Cependant, cela devient beaucoup plus complexe pour des générateurs relativement sophistiqués.

Lib/test/test_generators.py, la suite de test de la bibliothèque Python, contient de nombreux exemples intéressants. Voici un générateur qui implémente le parcours d'un arbre dans l'ordre en utilisant des générateurs de façon récursive.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Deux autres exemples de `test_generators.py` permettent de résoudre le problème des N Reines (placer *N* reines sur un échiquier de dimensions *NxN* de telle sorte qu'aucune reine ne soit en position d'en prendre une autre) et le problème du cavalier (trouver un chemin permettant au cavalier de visiter toutes les cases d'un échiquier *NxN* sans jamais visiter la même case deux fois).

4.1 Transmettre des valeurs au générateur

Avant Python 2.5, les générateurs ne pouvaient que produire des sorties. Une fois le code du générateur exécuté pour créer un itérateur, il était impossible d'introduire de l'information nouvelle dans la fonction mise en pause. Une astuce consistait à obtenir cette fonctionnalité en autorisant le générateur à consulter des variables globales ou en lui passant des objets mutables modifiés hors du générateur, mais ces approches étaient compliquées.

À partir de Python 2.5, il existe une méthode simple pour transmettre des valeurs à un générateur. Le mot-clé `yield` est devenu une expression qui renvoie une valeur sur laquelle il est possible d'opérer et que vous pouvez assigner à une variable :

```
val = (yield i)
```

Comme dans l'exemple ci-dessus, nous vous recommandons de **toujours** encadrer les expressions `yield` par des parenthèses lorsque vous utilisez leur valeur de retour. Elles ne sont pas toujours indispensables mais mieux vaut prévenir que guérir : il est plus facile de les ajouter systématiquement que de prendre le risque de les oublier là où elles sont requises.

(Les règles exactes de parenthésage sont spécifies dans la **PEP 342** : une expression `yield` doit toujours être parenthésée sauf s'il s'agit de l'expression la plus externe du côté droit d'une assignation. Cela signifie que vous pouvez écrire `val = yield i` mais que les parenthèses sont requises s'il y a une opération, comme dans `val = (yield i) + 12`.)

Des valeurs peuvent être transmises à un générateur en appelant sa méthode `send(value)`. Celle-ci reprend l'exécution du générateur et l'expression `yield` renvoie la valeur spécifiée. Si c'est la méthode `__next__()` habituelle qui est appelée, alors `yield` renvoie `None`.

Voici un exemple de compteur qui s'incrémente de 1 mais dont il est possible de modifier le compte interne.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

Et voici comment il est possible de modifier le compteur :

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

Puisque `yield` renvoie souvent `None`, vous devez toujours vérifier si c'est le cas. N'utilisez pas la valeur de retour à moins d'être certain que seule la méthode `send()` sera utilisée pour reprendre l'exécution de la fonction génératrice.

En plus de `send()`, il existe deux autres méthodes s'appliquant aux générateurs :

— `throw(type, value=None, traceback=None)` permet de lever une exception dans le générateur ; celle-ci est levée par l'expression `yield` à l'endroit où l'exécution a été mise en pause.
— `close()` lève une exception `GeneratorExit` dans le générateur afin de terminer l'itération. Le code du générateur qui reçoit cette exception doit lever à son tour `GeneratorExit` ou `StopIteration`. Il est illégal d'attraper cette exception et de faire quoi que ce soit d'autre, ceci déclenche une erreur `RuntimeError`. Lorsque le ramasse-miette de Python collecte le générateur, il appelle sa méthode `close()`.
Si vous devez exécuter du code pour faire le ménage lors d'une `GeneratorExit`, nous vous suggérons d'utiliser une structure `try: ... finally` plutôt que d'attraper `GeneratorExit`.

Ces changements cumulés transforment les générateurs de producteurs unidirectionnels d'information vers un statut hybride à la fois producteur et consommateur.

Les générateurs sont également devenus des **coroutines**, une forme généralisée de sous-routine. L'exécution des sous-routines démarre à un endroit et se termine à un autre (au début de la fonction et au niveau de l'instruction `return`), tandis qu'il est possible d'entrer, de sortir ou de reprendre une coroutine à différents endroits (les instructions `yield`).

5 Fonctions natives

Voyons un peu plus en détail les fonctions natives souvent utilisées de concert avec les itérateurs.

`map()` et `filter()` sont deux fonctions natives de Python qui clonent les propriétés des expressions génératrices :

`map(f, iterA, iterB, ...)` renvoie un itérateur sur une séquence `f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...`.

```
>>> def upper(s):
...     return s.upper()
```

```
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

Vous pouvez obtenir le même comportement à l'aide d'une compréhension de liste.

`filter(predicate, iter)` renvoie un itérateur sur l'ensemble des éléments de la séquence qui vérifient une certaine condition. Son comportement peut également être reproduit par une compréhension de liste. Un **prédicat** est une fonction qui renvoie vrai ou faux en fonction d'une certaine condition. Dans le cas de `filter()`, le prédicat ne doit prendre qu'un seul argument.

```
>>> def is_even(x):
...     return (x % 2) == 0
```

```
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

Cela peut se réécrire sous la forme d'une compréhension de liste :

```
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` énumère les éléments de l'itérable en renvoyant des paires contenant le nombre d'éléments déjà listés (depuis le *début*) et l'élément en cours

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print(item)
```

(suite sur la page suivante)

```
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` est souvent utilisée lorsque l'on souhaite boucler sur une liste tout en listant les indices pour lesquels une certaine condition est vérifiée :

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` rassemble tous les éléments de l'itérable dans une liste, les classe et renvoie le résultat classé. Les arguments *key* et *reverse* sont passés à la méthode `sort()` de la liste ainsi construite.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(Pour plus de détails sur les algorithmes de tri, se référer à sortinghowto.)

Les fonctions natives `any(iter)` et `all(iter)` permettent d'observer les valeurs de vérité des éléments d'un itérable. `any()` renvoie `True` si au moins un élément de l'itérable s'évalue comme vrai et `all()` renvoie `True` si tous les éléments s'évaluent comme vrai :

```
>>> any([0, 1, 0])
True
>>> any([0, 0, 0])
False
>>> any([1, 1, 1])
True
>>> all([0, 1, 0])
False
>>> all([0, 0, 0])
False
>>> all([1, 1, 1])
True
```

`zip(iterA, iterB, ...)` rassemble un élément de chaque itérable dans un n-uplet :

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
  ('a', 1), ('b', 2), ('c', 3)
```

Cela ne construit pas de liste stockée en mémoire, ni ne vide les itérateurs d'entrée avant de renvoyer sa valeur ; en réalité les n-uplets sont construits et renvoyés au fur et à mesure (il s'agit techniquement parlant d'un comportement d'"évaluation paresseuse).

Cet itérateur suppose qu'il opère sur des itérables de même longueur. Si la longueur des itérables diffère, le flux résultant a la même longueur que le plus court des itérables.

```
zip(['a', 'b'], (1, 2, 3)) =>
  ('a', 1), ('b', 2)
```

Toutefois, vous devez éviter de dépendre de ce comportement. En effet un élément d'un des itérables les plus longs peut être retiré puis jeté (car l'autre itérable est trop court). Cela signifie que vous ne pouvez alors plus utiliser cet itérable car vous allez sauter l'élément qui vient d'être jeté.

6 Le module *itertools*

Le module `itertools` contient de nombreux itérateurs très utilisés, ainsi que des fonctions pour combiner différents itérateurs. Cette section présente le contenu du module au travers de quelques exemples.

Les fonctions du module se divisent en quelques grandes catégories :

- Les fonctions qui transforment un itérateur existant en un nouvel itérateur.
- Les fonctions qui traitent les éléments d'un itérateur comme les arguments d'une fonction.
- Les fonctions qui permettent de sélectionner des portions de la sortie d'un itérateur.
- Une fonction qui permet de grouper la sortie d'un itérateur.

6.1 Créer de nouveaux itérateurs

`itertools.count(start, step)` renvoie un flux infini de valeurs régulièrement espacées. Vous pouvez spécifier la valeur de départ (par défaut, 0) et l'intervalle entre les nombres (par défaut, 1) :

```
itertools.count() =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
  10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

`itertools.cycle(iter)` sauvegarde une copie du contenu de l'itérable passé en argument et renvoie un nouvel itérateur qui produit tous les éléments du premier au dernier et se répète indéfiniment.

```
itertools.cycle([1, 2, 3, 4, 5]) =>
  1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

`itertools.repeat(elem, [n])` renvoie l'élément passé en argument *n* fois ou répète l'élément à l'infini si *n* n'est pas spécifié.

```
itertools.repeat('abc') =>
  abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
  abc, abc, abc, abc, abc
```

`itertools.chain(iterA, iterB, ...)` reçoit un nombre arbitraire d'itérables en entrée et les concatène, renvoyant tous les éléments du premier itérateur, puis tous ceux du second et ainsi de suite jusqu'à ce que tous les itérables aient été épuisés.

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
  a, b, c, 1, 2, 3
```

`itertools.islice(iter, [start], stop, [step])` renvoie une portion de l'itérateur. En passant seulement l'argument *stop*, il renvoie les *stop* premiers éléments. En spécifiant un indice de début, vous récupérez *stop* - *start*

éléments ; utilisez *step* pour spécifier une valeur de pas. Cependant vous ne pouvez pas utiliser de valeurs négatives pour *start*, *stop* ou *step* (contrairement aux listes et chaînes de caractères de Python).

```
itertools.islice(range(10), 8) =>
  0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
  2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
  2, 4, 6
```

`itertools.tee(iter, [n])` duplique un itérateur et renvoie *n* itérateurs indépendants, chacun copiant le contenu de l'itérateur source. La valeur par défaut pour *n* est 2. La réplication des itérateurs nécessite la sauvegarde d'une partie du contenu de l'itérateur source, ce qui peut consommer beaucoup de mémoire si l'itérateur est grand et que l'un des nouveaux itérateurs est plus consommé que les autres.

```
itertools.tee( itertools.count() ) =>
   iterA, iterB

where iterA ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

and   iterB ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

6.2 Appliquer des fonctions au contenu des itérateurs

Le module `operator` rassemble des fonctions équivalentes aux opérateurs Python. Par exemple, `operator.add(a, b)` additionne deux valeurs, `operator.ne(a, b)` est équivalent à `a != b` et `operator.attrgetter('id')` renvoie un objet appelable qui récupère l'attribut `.id`.

`itertools.starmap(func, iter)` suppose que l'itérable renvoie une séquence de n-uplets et appelle *func* en utilisant tous les n-uplets comme arguments :

```
itertools.starmap(os.path.join,
                  [('/bin', 'python'), ('/usr', 'bin', 'java'),
                   ('/usr', 'bin', 'perl'), ('/usr', 'bin', 'ruby')])
=>
  /bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/ruby
```

6.3 Sélectionner des éléments

Une autre catégorie de fonctions est celle permettant de sélectionner un sous-ensemble des éléments de l'itérateur selon un prédicat donné.

`itertools.filterfalse(predicate, iter)` est l'opposé de `filter()` et renvoie tous les éléments pour lesquels le prédicat est faux :

```
itertools.filterfalse(is_even, itertools.count()) =>
  1, 3, 5, 7, 9, 11, 13, 15, ...
```

`itertools.takewhile(predicate, iter)` renvoie les éléments de l'itérateur tant que ceux-ci vérifient le prédicat. Dès lors que le prédicat renvoie faux, l'itération s'arrête.

```
def less_than_10(x):
    return x < 10

itertools.takewhile(less_than_10, itertools.count()) =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
  0
```

`itertools.dropwhile(predicate, iter)` supprime des éléments tant que le prédicat renvoie vrai puis renvoie le reste des éléments de l'itérable.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

`itertools.compress(data, selectors)` prend un itérateur *data* et un itérateur *selectors* et renvoie les éléments de *data* pour lesquels l'élément correspondant de *selectors* est évalué à vrai. L'itération s'arrête lorsque l'un des deux itérateurs est épuisé :

```
itertools.compress([1, 2, 3, 4, 5], [True, True, False, False, True]) =>
   1, 2, 5
```

6.4 Fonctions combinatoires

`itertools.combinations(iterable, r)` renvoie un itérateur qui produit toutes les combinaisons possibles de *r*-uplets des éléments de *iterable*.

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
  (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 3), (2, 4), (2, 5),
  (3, 4), (3, 5),
  (4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
  (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5),
  (2, 3, 4), (2, 3, 5), (2, 4, 5),
  (3, 4, 5)
```

Les éléments de chaque tuple sont ordonnés dans le même ordre que leur apparition dans *iterable*. Ainsi, dans les exemples ci-dessus, le nombre 1 se trouve toujours avant 2, 3, 4 ou 5. La fonction `itertools.permutations(iterable, r=None)` supprime la contrainte sur l'ordre et renvoie tous les arrangements possibles de longueur *r* :

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
  (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 1), (2, 3), (2, 4), (2, 5),
  (3, 1), (3, 2), (3, 4), (3, 5),
  (4, 1), (4, 2), (4, 3), (4, 5),
  (5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
  (1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
```

(suite sur la page suivante)

(suite de la page précédente)

```
  ...
  (5, 4, 3, 2, 1)
```

Si vous ne spécifiez pas de valeur pour *r*, la longueur de l'itérable est utilisée par défaut, c'est-à-dire que toutes les permutations de la séquence sont renvoyées.

Notez que ces fonctions génèrent toutes les combinaisons possibles en se basant sur la position des éléments et ne requièrent pas que les éléments de *iterable* soient uniques :

```
itertools.permutations('aba', 3) =>
  ('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
  ('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

Le triplet `('a', 'a', 'b')` apparaît deux fois mais les deux chaînes de caractères `'a'` proviennent de deux positions différentes.

La fonction `itertools.combinations_with_replacement(iterable, r)` assouplit une autre contrainte : les éléments peuvent être répétés au sein du même n-uplet. Il s'agit d'un tirage avec remise : le premier élément sélectionné pour chaque n-uplet est replacé dans la séquence avant le tirage du deuxième.

```
itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2) =>
  (1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 2), (2, 3), (2, 4), (2, 5),
  (3, 3), (3, 4), (3, 5),
  (4, 4), (4, 5),
  (5, 5)
```

6.5 Grouper les éléments

La dernière fonction que allons voir, `itertools.groupby(iter, key_func=None)` est la plus complexe. `key_func(elem)` est une fonction qui produit une clé pour chaque élément renvoyé par l'itérable. Si vous ne spécifiez pas de fonction *key*, alors celle-ci est l'identité par défaut (c'est-à-dire que la clé d'un élément est l'élément lui-même).

`groupby()` rassemble tous éléments consécutifs de l'itérable sous-jacent qui ont la même clé et renvoie un flux de paires contenant la clé et un itérateur produisant la liste des éléments pour cette clé.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
             ...
            ]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
  ('AL', iterator-1),
  ('AK', iterator-2),
  ('AZ', iterator-3), ...

where
iterator-1 =>
  ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
  ('Anchorage', 'AK'), ('Nome', 'AK')
```

(suite sur la page suivante)

(suite de la page précédente)

```
iterator-3 =>
  ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

`groupby()` fait l'hypothèse que le contenu de l'itérable sous-jacent est d'ores et déjà ordonné en fonction de la clé. Notez que les itérateurs générés utilisent également l'itérable sous-jacent. Vous devez donc consommer l'intégralité des résultats du premier itérateur renvoyé (*iterator-1* dans l'exemple ci-dessus) avant de récupérer le deuxième itérateur (*iterator-2* dans l'exemple ci-dessus) et la clé à laquelle il est associé.

7 Le module *functools*

Le module `functools` introduit par Python 2.5 contient diverses fonctions d'ordre supérieur. Une **fonction d'ordre supérieur** prend une ou plusieurs fonctions en entrée et renvoie une fonction. L'outil le plus important de ce module est la fonction `functools.partial()`.

En programmant dans un style fonctionnel, il est courant de vouloir construire des variantes de fonctions existantes dont certains paramètres sont prédéfinis. Par exemple, considérons une fonction Python `f(a, b, c)`. Si vous voulez une nouvelle fonction `g(b, c)` équivalente à `f(1, b, c)`, c'est-à-dire fixer le premier paramètre de `f()`. La fonction `g()` est une appelée « application partielle » de `f()`.

Le constructeur de `partial()` prend en argument `(fonction, arg1, arg2, ..., kwarg1=value1, kwarg2=value2, ...)`. Un appel à l'objet ainsi créé invoque la fonction `fonction` avec les arguments spécifiés.

Voici un exemple court mais réaliste :

```
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

`functools.reduce(func, iter, [initial_value])` applique une opération cumulative à tous les éléments d'un itérable et ne peut donc être appliquée à des itérables infinis. *func* doit être une fonction qui prend deux éléments et renvoie une seule valeur. `functools.reduce()` prend les deux premiers éléments A et B renvoyés par l'itérateur et calcule `func(A, B)`. Elle extrait ensuite le troisième élément C et calcule `func(func(A, B), C)` puis combine ce résultat avec le quatrième élément renvoyé etc. jusqu'à épuisement de l'itérable. Une exception `TypeError` est levée si l'itérable ne renvoie aucune valeur. La valeur initiale *initial_value*, si spécifiée, est utilisée comme point de départ et le premier calcul est alors `func(inital_value, A)`.

```
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> functools.reduce(operator.concat, [])
Traceback (most recent call last):
  ...
TypeError: reduce() of empty sequence with no initial value
>>> functools.reduce(operator.mul, [1, 2, 3], 1)
6
>>> functools.reduce(operator.mul, [], 1)
1
```

Si vous combinez `operator.add()` avec `functools.reduce()`, vous allez additionner tous les éléments de l'itérable. Ce cas est suffisamment courant pour qu'il existe une fonction native `sum()` qui lui est équivalent :

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4], 0)
10
>>> sum([1, 2, 3, 4])
10
>>> sum([])
0
```

Cependant, il peut être plus lisible dans de nombreuses situations impliquant `functools.reduce()` de simplement écrire la boucle `for` :

```
import functools
# Instead of:
product = functools.reduce(operator.mul, [1, 2, 3], 1)

# You can write:
product = 1
for i in [1, 2, 3]:
    product *= i
```

`itertools.accumulate(iterable, func=operator.add)` est une fonction similaire qui réalise le même calcul mais, plutôt que de renvoyer seulement le résultat final, `accumulate()` renvoie un itérateur qui génère la séquence de tous les résultats intermédiaires :

```
itertools.accumulate([1, 2, 3, 4, 5]) =>
  1, 3, 6, 10, 15

itertools.accumulate([1, 2, 3, 4, 5], operator.mul) =>
  1, 2, 6, 24, 120
```

7.1 Le module *operator*

Le module `operator` mentionné précédemment contient un ensemble de fonctions reproduisant les opérateurs de Python. Ces fonctions sont souvent utiles en programmation fonctionnelle car elles permettent de ne pas avoir à écrire des fonctions triviales qui ne réalisent qu'une seule opération.

Voici quelques fonctions de ce module :

- Les opérations mathématiques : `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, ...
- Les opérations logiques : `not_()`, `truth()`.
- Les opérations bit à bit : `and_()`, `or_()`, `invert()`.
- Les comparaisons : `eq()`, `ne()`, `lt()`, `le()`, `gt()`, et `ge()`.
- L'identification des objets : `is_()`, `is_not()`.

Veuillez vous référer à la documentation du module *operator* pour une liste complète.

8 Expressions lambda et fonctions courtes

Dans un style de programmation fonctionnel, il est courant d'avoir besoin de petites fonctions utilisées comme prédicats ou pour combiner des éléments d'une façon ou d'une autre.

S'il existe une fonction native Python ou une fonction d'un module qui convient, vous n'avez pas besoin de définir de nouvelle fonction :

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

If the function you need doesn't exist, you need to write it. One way to write small functions is to use the `lambda` statement. `lambda` takes a number of parameters and an expression combining these parameters, and creates an anonymous function that returns the value of the expression :

```
adder = lambda x, y: x+y

print_assign = lambda name, value: name + '=' + str(value)
```

Une autre façon de faire est de simplement utiliser l'instruction `def` afin de définir une fonction de la manière habituelle :

```
def adder(x, y):
    return x + y

def print_assign(name, value):
    return name + '=' + str(value)
```

La méthode à préférer est une question de style, en général l'auteur évite l'utilisation de `lambda`.

Une des raisons est que `lambda` ne peut pas définir toutes les fonctions. Le résultat doit pouvoir se calculer en une seule expression, ce qui signifie qu'il est impossible d'avoir des comparaisons `if ... elif ... else` à plusieurs branches ou des structures `try ... except`. Si vous essayez de trop en faire dans une expression `lambda`, vous finirez avec une expression illisible. Par exemple, pouvez-vous dire du premier coup d'œil ce que fait le code ci-dessous ?

```
import functools
total = functools.reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
```

Vous pouvez sûrement comprendre ce que fait ce code mais cela prend du temps de démêler l'expression pour y voir plus clair. Une clause `def` concise améliore la situation :

```
import functools
def combine(a, b):
    return 0, a[1] + b[1]

total = functools.reduce(combine, items)[1]
```

Toutefois l'idéal aurait été de simplement se contenter d'une boucle `for` :

```
total = 0
for a, b in items:
    total += b
```

ou de la fonction native `sum()` et d'une expression génératrice :

```
total = sum(b for a, b in items)
```

Les boucles `for` sont souvent plus lisibles que la fonction `functools.reduce()`.

Frederik Lundh a suggéré quelques règles pour le réusinage de code impliquant les expressions `lambda` :

1. Écrire une fonction lambda.
2. Écrire un commentaire qui explique ce que fait cette satanée fonction lambda.
3. Scruter le commentaire pendant quelques temps et réfléchir à un nom qui synthétise son essence.
4. Réécrire la fonction lambda en une définition *def* en utilisant ce nom.
5. Effacer le commentaire.

J'aime beaucoup ces règles mais vous êtes libre de ne pas être d'accord et de ne pas préférer ce style sans lambda.

9 Historique des modifications et remerciements

L'auteur souhaiterait remercier les personnes suivantes pour leurs suggestions, leurs corrections et leur aide sur les premières versions de cet article : Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1 : publiée le 30 juin 2006.

Version 0.11 : publiée le 1er juillet 2006. Correction orthographique.

Version 0.2 : publiée le 10 juillet 2006. Fusion des sections *genexp* et *listcomp*. Correction orthographique.

Version 0.21 : ajout de plusieurs références suggérées sur la liste de diffusion *tutor*.

Version 0.30 : ajout d'une section sur le module `functional` écrite par Collin Winter ; ajout d'une courte section sur le module `operator` ; quelques autres modifications.

10 Références

10.1 Général

Structure and Interpretation of Computer Programs par Harold Abelson et Gerald Jay Sussman avec Julie Sussman. Disponible à l'adresse https://mitpress.mit.edu/sicp/. Ce livre est un classique en informatique. Les chapitres 2 et 3 présentent l'utilisation des séquences et des flux pour organiser le flot de données dans un programme. Les exemples du livre utilisent le langage Scheme mais la plupart des approches décrites dans ces chapitres s'appliquent au style fonctionnel de Python.

http://www.defmacro.org/ramblings/fp.html : une présentation générale à la programmation fonctionnelle avec une longue introduction historique et des exemples en Java.

https://fr.wikipedia.org/wiki/Programmation_fonctionnelle : l'entrée Wikipédia qui décrit la programmation fonctionnelle.

https://fr.wikipedia.org/wiki/Coroutine : l'entrée pour les coroutines.

https://fr.wikipedia.org/wiki/Curryfication : l'entrée pour le concept de curryfication (création d'applications partielles).

10.2 Spécifique à Python

http://gnosis.cx/TPiP/ : le premier chapitre du livre de David Mertz *Text Processing in Python* présente l'utilisation de la programmation fonctionnelle pour le traitement de texte dans la section « Utilisation des fonctions d'ordre supérieur pour le traitement de texte ».

Mertz a également écrit une série de 3 articles (en anglais) sur la programmation fonctionnelle pour le site de IBM *DeveloperWorks*, voir la partie 1, la partie 2 et la partie 3.

10.3 Documentation Python

Documentation du module `itertools`.

Documentation du module `functools`.

Documentation du module `operator`.

PEP 289 : *« Generator Expressions »*

PEP 342 : *« Coroutines via Enhanced Generators »* décrit les nouvelles fonctionnalités des générateurs en Python 2.5.

Index

P