Guide pratique : programmation fonctionnelle

Version 2.7.18

Guido van Rossum
and the Python development team

mai 20, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

Author A. M. Kuchling

Release 0.31

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

This section explains the basic concept of functional programming ; if you're just interested in learning about Python language features, skip to the next section.

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.

The designers of some computer languages choose to emphasize one particular approach to programming. This often makes it difficult to write programs that use a different approach. Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm ; you can write programs or libraries that are largely procedural, object-oriented, or functional in all of these languages. In a large program, different sections might be written using different approaches ; the GUI might be object-oriented while the processing logic is procedural or functional, for example.

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.

Some languages are very strict about purity and don't even have assignment statements such as `a=3` or `c = a + b`, but it's difficult to avoid all side effects. Printing to the screen or writing to a disk file are side effects, for example. For example, in Python a `print` statement or a `time.sleep(1)` both return no useful value ; they're only called for their side effects of sending some text to the screen or pausing execution for a second.

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.

An iterator is an object representing a stream of data ; this object returns the data one element at a time. A Python iterator must support a method called `next()` that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, `next()` must raise the `StopIteration` exception. Iterators don't have to be finite, though ; it's perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in `iter()` function takes an arbitrary object and tries to return an iterator that will return the object's contents or elements, raising `TypeError` if the object doesn't support iteration. Several of Python's built-in data types support iteration, the most common being lists and dictionaries. An object is called an **iterable** object if you can get an iterator for it.

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

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

(suite sur la page suivante)

(suite de la page précédente)

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

Python expects iterable objects in several different contexts, the most important being the `for` statement. In the statement `for X in Y`, Y must be an iterator or some object for which `iter()` can create an iterator. These two statements are equivalent :

```
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)
```

Built-in functions such as `max()` and `min()` can take a single iterator argument and will return the largest or smallest element. The `"in"` and `"not in"` operators also support iterators : `X in iterator` is true if X is found in the stream returned by the iterator. You'll run into obvious problems if the iterator is infinite ; `max()`, `min()` will never return, and if the element X never appears in the stream, the `"in"` and `"not in"` operators won't return either.

Note that you can only go forward in an iterator ; there's no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the `next()` method. Functions may therefore consume all of the iterator's output, and if you need to do something different with the same stream, you'll have to create a new iterator.

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.

Calling `iter()` on a dictionary returns an iterator that will loop over the dictionary's keys :

```
>>> 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.

Applying `iter()` to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate iterator.

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'}
```

Files also support iteration by calling the `readline()` method until there are no more lines in the file. This means you can read each line of a file like this :

```
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 = set((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
```

Any function containing a `yield` keyword is a generator function ; this is detected by Python's bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn't return a single value ; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `.next()` method, the function will resume executing.

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

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
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)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values ; after executing a `return` the generator cannot return any further values. `return` with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class can be much messier.

The test suite included with Python's library, `test_generators.py`, contains a number of more interesting examples. Here's one generator that implements an in-order traversal of a tree using generators recursively.

```
# 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.

(PEP 342 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. This method resumes the generator's code and the `yield` expression returns the specified value. If the regular `next()` method is called, the `yield` returns `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)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    print it.next()
StopIteration
```

Because `yield` will often be returning `None`, you should always check for this case. Don't just use its value in expressions unless you're sure that the `send()` method will be the only method used to resume your generator function.

In addition to `send()`, there are two other new methods on generators :

— `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator ; the exception is raised by the `yield` expression where the generator's execution is paused.

— `close()` raises a `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise `GeneratorExit` or `StopIteration`; catching the exception and doing anything else is illegal and will trigger a `RuntimeError`. `close()` will also be called by Python's garbage collector when the generator is garbage-collected.
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.

Two of Python's built-in functions, `map()` and `filter()`, are somewhat obsolete; they duplicate the features of list comprehensions but return actual lists instead of iterators.

`map(f, iterA, iterB, ...)` returns a list containing `f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ....`

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

```
>>> map(upper, ['sentence', 'fragment'])
['SENTENCE', 'FRAGMENT']
```

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

As shown above, you can achieve the same effect with a list comprehension. The `itertools.imap()` function does the same thing but can handle infinite iterators; it'll be discussed later, in the section on the `itertools` module.

`filter(predicate, iter)` returns a list that contains all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A **predicate** is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.

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

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

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

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

`filter()` also has a counterpart in the `itertools` module, `itertools.ifilter()`, that returns an iterator and can therefore handle infinite sequences just as `itertools.imap()` can.

`reduce(func, iter, [initial_value])` doesn't have a counterpart in the `itertools` module because it cumulatively performs an operation on all the iterable's elements and therefore can't be applied to infinite iterables. `func`

must be a function that takes two elements and returns a single value. `reduce()` takes the first two elements A and B returned by the iterator and calculates `func(A, B)`. It then requests the third element, C, calculates `func(func(A, B), C)`, combines this result with the fourth element returned, and continues until the iterable is exhausted. If the iterable returns no values at all, a `TypeError` exception is raised. If the initial value is supplied, it's used as a starting point and `func(initial_value, A)` is the first calculation.

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

If you use `operator.add()` with `reduce()`, you'll add up all the elements of the iterable. This case is so common that there's a special built-in called `sum()` to compute it :

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

For many uses of `reduce()`, though, it can be clearer to just write the obvious `for` loop :

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

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

`enumerate(iter)` counts off the elements in the iterable, returning 2-tuples containing the count and each element.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print item
(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, [cmp=None], [key=None], [reverse=False])` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The `cmp`, `key`, and `reverse` arguments are passed through to the constructed list's `.sort()` method.

```
>>> 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]
```

(For a more detailed discussion of sorting, see the Sorting mini-HOWTO in the Python wiki at https://wiki.python.org/moin/HowTo/Sorting.)

The `any(iter)` and `all(iter)` built-ins look at the truth values of an iterable's contents. `any()` returns `True` if any element in the iterable is a true value, and `all()` returns `True` if all of the elements are true values :

```
>>> 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
```

6 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 a small function that returns the value of the expression :

```
lowercase = lambda x: x.lower()

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

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

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

```
def lowercase(x):
    return x.lower()
```

(suite sur la page suivante)

(suite de la page précédente)

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

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

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

One reason for my preference is that `lambda` is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can't have multiway `if... elif... else` comparisons or `try... except` statements. If you try to do too much in a `lambda` statement, you'll end up with an overly complicated expression that's hard to read. Quick, what's the following code doing ?

```
total = 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 :

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

total = 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)
```

Many uses of `reduce()` are clearer when written as `for` loops.

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.

7 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.

7.1 Créer de nouveaux itérateurs

`itertools.count(n)` returns an infinite stream of integers, increasing by 1 each time. You can optionally supply the starting number, which defaults to 0 :

```
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.cycle(iter)` saves a copy of the contents of a provided iterable and returns a new iterator that returns its elements from first to last. The new iterator will repeat these elements infinitely.

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

`itertools.repeat(elem, [n])` returns the provided element `n` times, or returns the element endlessly if `n` is not provided.

```
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, ...)` takes an arbitrary number of iterables as input, and returns all the elements of the first iterator, then all the elements of the second, and so on, until all of the iterables have been exhausted.

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

`itertools.izip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple :

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

It's similar to the built-in `zip()` function, but doesn't construct an in-memory list and exhaust all the input iterators before returning ; instead tuples are constructed and returned only if they're requested. (The technical term for this behaviour is lazy evaluation.)

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.

```
itertools.izip(['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.

`itertools.islice(iter, [start], stop, [step])` returns a stream that's a slice of the iterator. With a single `stop` argument, it will return the first `stop` elements. If you supply a starting index, you'll get `stop-start` elements, and if you supply a value for `step`, elements will be skipped accordingly. Unlike Python's string and list slicing, you can't use negative values for `start`, `stop`, or `step`.

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

(suite sur la page suivante)

(suite de la page précédente)

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

`itertools.tee(iter, [n])` replicates an iterator; it returns `n` independent iterators that will all return the contents of the source iterator. If you don't supply a value for `n`, the default is 2. Replicating iterators requires saving some of the contents of the source iterator, so this can consume significant memory if the iterator is large and one of the new iterators is consumed more than the others.

```
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, ...
```

7.2 Appliquer des fonctions au contenu des itérateurs

Two functions are used for calling other functions on the contents of an iterable.

`itertools.imap(f, iterA, iterB, ...)` returns a stream containing `f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...`:

```
itertools.imap(operator.add, [5, 6, 5], [1, 2, 3]) =>
  6, 8, 8
```

The `operator` module contains a set of functions corresponding to Python's operators. Some examples are `operator.add(a, b)` (adds two values), `operator.ne(a, b)` (same as `a!=b`), and `operator.attrgetter('id')` (returns a callable that fetches the `"id"` attribute).

`itertools.starmap(func, iter)` assumes that the iterable will return a stream of tuples, and calls `f()` using these tuples as the arguments :

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

7.3 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.ifilter(predicate, iter)` returns all the elements for which the predicate returns true :

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

itertools.ifilter(is_even, itertools.count()) =>
  0, 2, 4, 6, 8, 10, 12, 14, ...
```

`itertools.ifilterfalse(predicate, iter)` is the opposite, returning all elements for which the predicate returns false :

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

`itertools.takewhile(predicate, iter)` returns elements for as long as the predicate returns true. Once the predicate returns false, the iterator will signal the end of its results.

```
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)` discards elements while the predicate returns true, and then returns the rest of the iterable's results.

```
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, ...
```

7.4 Grouper les éléments

The last function I'll discuss, `itertools.groupby(iter, key_func=None)`, is the most complicated. `key_func(elem)` is a function that can compute a key value for each element returned by the iterable. If you don't supply a key function, the key is simply each element itself.

`groupby()` collects all the consecutive elements from the underlying iterable that have the same key value, and returns a stream of 2-tuples containing a key value and an iterator for the elements with that key.

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

def get_state ((city, state)):
    return state

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()` assumes that the underlying iterable's contents will already be sorted based on the key. Note that the returned iterators also use the underlying iterable, so you have to consume the results of iterator-1 before requesting iterator-2 and its corresponding key.

8 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()`.

The constructor for `partial` takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

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')
```

8.1 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 :

- Math operations : `add()`, `sub()`, `mul()`, `div()`, `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.

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 also wrote a 3-part series of articles on functional programming for IBM's DeveloperWorks site ; see

part 1, part 2, and part 3,

10.3 Documentation Python

Documentation du module `itertools`.

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