

---

# 関数型プログラミング HOWTO

リリース 3.11.14

Guido van Rossum and the Python development team

10 月 15, 2025

## 目次

1	はじめに	2
1.1	形式的証明可能性	3
1.2	モジュラー性	4
1.3	デバッグやテストの簡単さ	4
1.4	結合性	4
2	イテレータ (iterator)	5
2.1	イテレータ対応のデータ型	6
3	ジェネレータ式とリスト内包表記	7
4	ジェネレータ (generator)	9
4.1	ジェネレータに値を渡す	11
5	組み込み関数 (built-in function)	12
6	itertools モジュール	15
6.1	新しいイテレータを作る	15
6.2	要素に対して関数を呼ぶ	16
6.3	要素を選択する	17
6.4	組合せ関数	17
6.5	要素をグループ分けする	19
7	functools モジュール	19
7.1	operator モジュール	21
8	小さな関数とラムダ式	21
9	更新履歴と謝辞	23
10	参考資料	23
10.1	一般論	23

10.2 Python 特有の話 . . . . .	24
10.3 Python 文書 . . . . .	24
索引	25

---

## 著者

A. M. Kuchling

## リリース

0.32

この文書では、関数型スタイルでプログラムを実装するのにピッタリな Python の機能を見てまわることしましょう。まず関数型プログラミングという概念を紹介したあと、iterator や generator のような言語機能、および itertools や functools といった関連するライブラリモジュールを見ることにします。

# 1 はじめに

この章は関数型プログラミングの基本概念を説明します; Python の言語機能についてだけ知りたい人は、次の章の **イテレータ** (*iterator*) まで飛ばしてください。

プログラミング言語とは問題を分解するものですが、各言語がサポートする分解方法にはいくつかの種類があります:

- ほとんどのプログラミング言語は **手続き型** です: プログラムは、入力に対して行うべきことをコンピュータに教える指示リストとなります。C, Pascal, さらには Unix シェルまでもが手続き型言語に入ります。
- **宣言型** 言語で書くのは、解くべき問題を説明する仕様書であって、それを効率的に計算処理する方法を見付けるのは言語実装の役目です。SQL はおそらく一番よく知られた宣言型言語です; SQL のクエリは取得したいデータセットを説明しているだけで、テーブルを走査するかインデックスを使うか、どのサブクローズから実行するか等々を決めるのは SQL エンジンなのです。
- **オブジェクト指向** プログラムはオブジェクトの集まりを操作します。オブジェクトには内部状態があり、その状態を調べたり色々と変更したりするためのメソッドがあります。Smalltalk や Java はオブジェクト指向言語です。C++ と Python はオブジェクト指向プログラミングをサポートしていますが、関連する機能を使わなくても構わないようになっています。
- **関数型** プログラミングは問題をいくつかの関数にわけて考えます。理想的に言うと、関数は入力を受けて出力を吐くだけで、同じ入力に対して異なる出力をするような内部状態を一切持ちません。有名な関数型言語には ML 一家 (Standard ML, OCaml 等々) と Haskell があります。

設計者が特定のアプローチを強調することにした言語もありますが、そうすると大抵は、別のアプローチを使うプログラムを書きにくくなります。複数のアプローチに対応した言語もあり、Lisp, C++, Python はそうしたマルチパラダイム言語です; この中のどれを使っても、基本的に手続き型な、または基本的にオブジェクト指向な、とか、基本的に関数型なプログラムやライブラリを書くことができます。大きなプログラムでは、

各部で別々のアプローチを使って書くことがあるかもしれませんが、GUI はオブジェクト指向で、でも処理ロジックは手続き型や関数型で、といったようにです。

関数型プログラムでは、入力は一連の関数を通して流れていきます。それぞれの関数は入力に何らかの作業をして出力します。関数型スタイルにおいては、内部状態を変えてしまったり、返り値に現れない変更をしたりといった副作用のある関数はやめるように言われています。副作用のまったくない関数は **純粋関数型** であるとされます。副作用をなくすということは、プログラムの実行中に順次変化していくデータ構造を持たない、つまり各関数の出力はその入力にしか影響を受けてはいけないということです。

ある言語では純粋さにとても厳しく  $a=3$  や  $c = a + b$  のような代入文すら存在しないほどですが、画面への表示やディスクファイルへの書き込みなど、すべての副作用を避けるのは難しいです。別の例として、`print()` や `time.sleep()` 関数の呼び出しもありますが、どちらも有用な値を返しません。画面にテキストを送ったり、実行を 1 秒間停めたりといった副作用のためだけに呼ばれるのです。

関数型スタイルで書いた Python プログラムはふつう、I/O や代入を完全になくすといった極端なところまでは行かずに、関数型っぽく見えるインターフェースを提供しつつも内部では非関数型の機能を使います。たとえば、関数内でローカル変数の代入は使いますが、グローバル変数は変更せず、他の副作用もないように実装するのです。

関数型プログラミングはオブジェクト指向プログラミングの反対と考えることもできます。オブジェクト指向において、オブジェクトは内部状態とそれを変更するメソッドコールの入ったカプセルであり、プログラムはその状態を適正に変化させていく手順です。一方で、関数型プログラミングは可能なかぎり状態の変更を避け、関数どうしの間を流れるデータだけを扱おうとします。Python ではこの二つのアプローチを結び合わせることができます。アプリケーション内のオブジェクト（メール、トランザクション、等々）を表現したインスタンスを、関数が受け渡しするようにするのです。

関数型デザインは、わけのわからない制約に見えるかもしれません。どうしてオブジェクトも副作用もないほうが良いのでしょうか。実は、関数型スタイルには理論と実践に基づく次の利点があるのです：

- 形式的証明可能性。
- モジュラー性。
- 結合性。
- デバッグやテストの簡単さ。

## 1.1 形式的証明可能性

理論面の利点としては、プログラムが正しいことの数学的証明を他より簡単に構築できるという点があります。

研究者たちは長いあいだ、プログラムが正しいことを数学的に証明する方法の発見に血道をあげてきました。これは、色々な入力でテストして出力が正しかったからまあ正しいだろう、と結論するのとも違いますし、ソースコードを読んで「間違いはなさそうだ」と言うのとも別の話です；目指すのは、出現しうる入力すべてに対してプログラムが正しい結果を出すことの厳密な証明なのです。

プログラムを証明するために使われているのは **不変式** を書き出していくというテクニックで、不変式とは入

力データやプログラム変数のうち常に真である性質のことで、コードの一行一行で、**実行前** の不変式  $X$  と  $Y$  が真なら **実行後** に ちょっと違う不変式  $X'$  と  $Y'$  が真になることを示していき、これをプログラムの終わりまで続けるわけです。すると最終的な不変式はプログラムの出力に合った条件になっているはずで

関数型プログラミングが代入を嫌うのは、この不変式テクニックでは代入を扱いにくいからで、代入は、それまで真だった不変式を壊しておいて、自分は次の行に伝えてゆける不変式を生み出さないことがあるのです。

残念ながら、プログラムの証明はだいたい実際的でもありませんし、Python ソフトウェアにも関係ありません。本当に簡単なプログラムでも、証明には数ページにわたる論文が必要なのです；ある程度の複雑なプログラムではもう尋常でない長さになってしまうので、日常で使っているプログラム (Python インタプリタ、XML パーサ、ウェブブラウザ) はほとんど、あるいはすべて、正しさを証明するのは不可能でしょう。仮に証明を書き出したり生成したりしても、その証明を検証するための疑いが残ります；証明に間違いがあるかもしれない、その場合は証明したと自分で勝手に思い込んでいただけになるのです。

## 1.2 モジュラー性

より実用的には、関数型プログラミングをすると問題を細かく切り分けることになるという利点があります。結果としてプログラムはモジュラー化されます。複雑な変形を施す大きな関数を書くより、一つのことに絞ってそれだけをする小さな関数のほうが書きやすいものです。それに、小さいほうが読むのもエラーをチェックするのも簡単です。

## 1.3 デバッグやテストの簡単さ

テストやデバッグも関数型プログラムなら簡単です。

関数が一般的に小さくて明確に意味付けされているので、デバッグ方法は単純です。プログラムが正しく動かないときには、関数ひとつひとつがデータの正しさをチェックするポイントになるので、それぞれの時点における入力と出力を見ていけば、バグの原因となる関数を素早く切り出すことができるのです。

ひとつひとつの関数がユニットテストの対象になり得るわけですから、テストも簡単です。関数はシステムの状態に依存しませんので、テストの実行前にそうした状態を再現する必要はありません；単に適切な入力を合成して、出力が期待どおりかどうかチェックするだけで良いのです。

## 1.4 結合性

関数型スタイルのプログラムを作っていると、色々な入力や出力のために色々な関数を書くことになります。仕方なく特定のアプリケーションに特化した関数を書くこともあるでしょうけれど、広範なプログラムに使える関数もあることでしょう。たとえば、ディレクトリ名を受け取ってその中の XML ファイル一覧を返す関数や、ファイル名を受け取って内容を返す関数などは、多様な場面に適用できそうです。

時たつうちに自分の特製ライブラリやユーティリティが充実してくると、新しいプログラムも、既存の関数を調整して少し今回に特化した関数を書くだけで組み立てられるようになります。

## 2 イテレータ (iterator)

まずは関数型スタイルのプログラムを書く際の基礎となる重要な Python 機能から見ていきましょう: イテレータです。

イテレータは連続データを表現するオブジェクトです; このオブジェクトは一度に一つの要素ずつデータを返します。Python のイテレータは `__next__()` という、引数を取らず次の要素を返すメソッドを必ずサポートしています。データストリームに要素が残っていない場合、`__next__()` は必ず `StopIteration` 例外を出します。ただし、イテレータの長さは有限である必要はありません; 無限のストリームを生成するイテレータを書くというのもまったく理に適ったことです。

ビルトインの `iter()` 関数は任意のオブジェクトを受けて、その中身や要素を返すイテレータを返そうとします。引数のオブジェクトがイテレータを作れないときは `TypeError` を投げます。Python のビルトインなデータ型にもいくつかイテレータ化のできるものがあり、中でもよく使われるのはリストと辞書です。イテレータを作れるオブジェクトは `iterable` オブジェクトと呼ばれます。

手を動かしてイテレータ化の実験をしてみましょう:

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
>>> 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 は色々な文脈でイテラブルなオブジェクトを期待しますが、最も重要なのは `for` 文です。`for X in Y` という文の `Y` は、イテレータか、あるいは `iter()` でイテレータを作れるオブジェクトである必要があります。次の二つは同じ意味になります:

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

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

イテレータは `list()` や `tuple()` といったコンストラクタ関数を使ってリストやタプルに具現化することができます:

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

(次のページに続く)

```
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

シーケンスのアンパックもイテレータに対応しています: イテレータが N 個の要素を返すということが事前にわかっているならば、N-タプルにアンパックすることができます:

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

ビルトイン関数の `max()` や `min()` など、イテレータ一つだけを引数に取って最大・最小の要素を返すことができます。"`in`" や "`not in`" 演算子もイテレータに対応しています: `X in iterator` は、そのイテレータから返るストリームに `X` があれば真です。ですからイテレータが無限長だと、当然ながら問題に直面します; `max()`, `min()` はいつまでも戻って来ませんし、要素 `X` がストリームに出てこなければ "`in`", "`not in`" オペレータも戻りません。

イテレータは次に進むことしかできませんのでご注意ください; 前の要素を手に入れたり、イテレータをリセットしたり、コピーを作ったりする方法はありません。イテレータがオブジェクトとしてそうした追加機能を持つことはできますが、プロトコルでは `__next__()` メソッドのことしか指定されていません。ですから関数はイテレータの出力を使い尽くしてしまうかもしれませんし、同じストリームに何か別のことをする必要があるなら新しいイテレータを作らなくてははいけません。

## 2.1 イテレータ対応のデータ型

リストやタプルがイテレータに対応している方法については既に見ましたが、実のところ Python のシーケンス型はどれでも、たとえば文字列なども、自動でイテレータ生成に対応しています。

辞書に対して `iter()` すると、辞書のキーでループを回すイテレータが返されます:

```
>>> 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])
Jan 1
Feb 2
Mar 3
Apr 4
May 5
Jun 6
Jul 7
Aug 8
Sep 9
Oct 10
Nov 11
```

(次のページに続く)

Dec 12

Python 3.7 から、辞書の反復順序は挿入順序と同じであることが保証されていることに注意してください。以前のバージョンでは、その振る舞いは仕様が定められておらず、実装ごとに異なることがありました。

辞書は `iter()` を適用するとキーでループを回しますが、辞書には他のイテレータを返すメソッドもあります。明示的に値、あるいはキーと値のペアでイテレートしたければ、`values()`、`items()` というメソッドでイテレータを作ることができます。

逆に `dict()` コンストラクタは、有限な (key, value) タプルのストリームを返すイテレータを受け入れることができます:

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

ファイルも、最後の行まで `readline()` メソッドを呼んでいくことでイテレータ化に対応しています。つまりこうやってファイルの各行を読んでいくことができるわけです:

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

セットはイテラブルを受け取れますし、そのセットの要素でイテレートすることもできます:

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

### 3 ジェネレータ式とリスト内包表記

イテレータの出力に対してよく使う操作トップ 2 は、(1) ひとつずつ全要素に操作を実行する、および (2) 条件に合う要素でサブセットを作る、です。たとえば文字列のリストなら、各行のうしろに付いた邪魔なホワイトスペースを削りたいとか、特定の文字列を含む部分をピックアップしたいなどと思うかもしれません。

リスト内包表記とジェネレータ式 (略して「listcomp」と「genexp」) は、そうした操作向けの簡潔な表記方法です。これは関数型プログラミング言語 Haskell (<https://www.haskell.org/>) にインスパイアされました。文字列のストリームからホワイトスペースをすべて削るのは次のコードでできます:



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

特定の要素だけを選び出すのは "if" 条件式を付けることで可能です:

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

リスト内包表記を使うと Python リストが返って来ます; `stripped_list` は実行結果の行が入ったリストであって、イテレータではありません。ジェネレータ式はイテレータを返し、これだと必要に応じてだけ値を算出しますので、すべての値を一度に出す必要がありません。つまりリスト内包表記のほうは、無限長ストリームや膨大なデータを返すようなイテレータを扱う際には、あまり役に立たないということです。そういった状況ではジェネレータ式のほうが好ましいと言えます。

ジェネレータ式は丸括弧 `()` で囲まれ、リスト内包表記は角括弧 `[]` で囲まれます。ジェネレータ式の形式は次のとおりです:

```
( 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 )
```

リスト内包表記も、外側の括弧が違うだけ (丸ではなく角括弧) で、あとは同じです。

生成される出力は `expression` 部分の値を要素として並べたものになります。if 節はすべて、なくても大丈夫です; あれば `condition` が真のときだけ `expression` が評価されて出力に追加されます。

ジェネレータ式は常に括弧の中に書かなければなりませんが、関数コールの目印になっている括弧でも大丈夫です。関数にすぐ渡すイテレータを作りたいければこう書けるのです:

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

`for...in` 節は複数つながられますが、どれにも、イテレートするためのシーケンスが含まれています。それらのシーケンスは並行して **ではなく**、左から右へ順番にイテレートされるので、長さが同じである必要はありません。`sequence1` の各要素ごとに毎回最初から `sequence2` をループで回すのです。その後 `sequence1` と `sequence2` から出た要素ペアごとに、`sequence3` でループします。

別の書き方をすると、リスト内包表記やジェネレータ式は次の Python コードと同じ意味になります:



```

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.

```

つまり、複数の `for...in` 節があって `if` がいないときの最終出力は、長さが各シーケンス長の積に等しくなるということです。長さ 3 のリスト二つなら、出力リストの長さは 9 要素です:

```

>>> 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)]

```

Python の文法に曖昧さを紛れ込ませないように、`expression` でタプルを作るなら括弧で囲わなくてはなりません。下にあるリスト内包表記で、最初のは構文エラーですが、二番目は有効です:

```

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

```

## 4 ジェネレータ (generator)

ジェネレータは、イテレータを書く作業を簡単にする、特殊な関数です。標準的な関数は値を計算して返しますが、ジェネレータが返すのは、一連の値を返すイテレータです。

Python や C の標準的な関数コールについては、よくご存じに違いありません。関数を呼ぶと、ローカル変数を作るプライベートな名前空間ができますね。その関数が `return` 文まで来ると、ローカル変数が破壊されてから、戻り値が呼び出し元に返ります。次に同じ関数をもう一度呼ぶと、新しいプライベート名前空間に新規のローカル変数が作られるのです。しかし、関数を出るときにローカル変数を捨てなければどうなるでしょうか。その出ていったところから関数を続行できたとしたら、どうでしょう。これこそジェネレータが提供する機能です; すなわち、ジェネレータは続行できる関数と考えることができます。

ごく単純なジェネレータ関数の例がこちらにあります:

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

yield キーワードを含む関数はすべてジェネレータ関数です; Python の bytecode コンパイラがこれを検出して、特別な方法でコンパイルしてくれるのです。

ジェネレータ関数は、呼ばれたときに一回だけ値を返すのではなく、イテレータ プロトコルに対応したオブジェクトを返します。上の例で yield を実行したとき、ジェネレータは return 文のようにして i の値を出力します。yield と return 文の大きな違いは、yield に到達した段階でジェネレータの実行状態が一時停止になって、ローカル変数が保存される点です。次回そのジェネレータの `__next__()` を呼ぶと、そこから関数が実行を再開します。

上記 `generate_ints()` ジェネレータの使用例はこちらです:

```
>>> 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
```

同じく `for i in generate_ints(5)` や `a, b, c = generate_ints(3)` といった書き方もできます。

ジェネレータ関数の中では、return value は `__next__()` メソッドから送出された `StopIteration(value)` を引き起こします。これが発生した場合や、関数の終わりに到達した場合は、値の生成が終了してジェネレーターがそれ以上の値を返さない。

自分でクラスを書いて、ジェネレータで言うところのローカル変数をインスタンス変数として全部保管しておけば、同じ効果を得ることは可能です。たとえば整数のリストを返すのは、`self.count` を 0 にして、`__next__()` メソッドが `self.count` をインクリメントして返すようにすればできます。しかしながら、ある程度複雑なジェネレータになってくると、同じことをするクラスを書くのは格段にややこしいことになります。

Python のライブラリに含まれているテストスイート `Lib/test/test_generators.py` には、ほかにも興味深い例が数多く入っています。これは二分木の通りがけ順 (in-order) 探索を再帰で実装したジェネレータです。

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

(次のページに続く)

```

    yield x

    yield t.label

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

```

ほかにも `test_generators.py` には、N-Queens 問題 (N×N コマのチェス盤に、互いに攻撃できないような配置で N 個のクイーンを置く) やナイト・ツアー (N×N 盤の全コマをナイトが一度ずつ通るような経路を探す) の解を出す例が入っています。

## 4.1 ジェネレータに値を渡す

Python 2.4 までのジェネレータは出力することしかできませんでした。ジェネレータのコードを実行してイテレータを作ってしまったあとで、その関数を再開するときに新しい情報を渡す手段はなかったのです。ジェネレータがグローバル変数を見るようにしたり、ミュータブルなオブジェクトを渡しておいて呼び出し元であるからそれを変更したり、といったハックは可能でしたが、どれもゴチャゴチャしていますね。

Python 2.5 で、ジェネレータに値を渡す簡単な手段ができました。`yield` が、変数に代入したり演算したりできる値を返す式になったのです:

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

上のように、返り値で何かをするときは `yield` 式の前後に **必ず** 括弧を付けるようお勧めします。括弧は常に必要なわけではありませんが、どんなとき付けなくて良いのかを覚えておくより、いつも付けておくほうが楽ですから。

(PEP 342 がその規則を正確に説明していますが、それによると `yield`-式は、代入式で右辺のトップレベルにあるとき以外はいつも括弧を付ける必要があります。つまり `val = yield i` とは書けますが、`val = (yield i) + 12` のように演算子があるときは括弧を使わなくてはいけません。)

ジェネレータに値を送るには `send(value)` メソッドを呼びます。するとジェネレータのコードが実行を再開し、`yield` 式がその値を返すのです。ふつうの `__next__()` メソッドを呼ぶと、`yield` は `None` を返します。

下にあるのは 1 ずつ増える単純なカウンタですが、内部カウンタの値を変更することができるようになっています。

```

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

```

そしてカウンタ変更の例がこちらです:

```
>>> 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
```

`yield` が `None` を返すことはよくあるのですから、そうなっていないかどうか必ずチェックしておくべきです。ジェネレータ関数を再開するために使うメソッドが `send()` しかないのだと確定してるのでない限り、式の値をそのまま使ってははいけません。

ジェネレータには、`send()` のほかにもメソッドが二つあります:

- `throw(value)` はジェネレータ内で例外を投げるために使います; その例外はジェネレータの実行が停止したところの `yield` 式によって投げられます。
- `close()` はジェネレータ内で `GeneratorExit` 例外を投げて、イテレータを終了させます。この例外を受け取ったジェネレータのコードは `GeneratorExit` か `StopIteration` を投げなくてははいけません; この例外を捕捉して何かほかのことをしようとするのは規則違反であり、`RuntimeError` を引き起こします。`close()` はジェネレータが GC されるときにも呼ばれます。

`GeneratorExit` が起こったときにクリーンアップ作業をする必要があるなら、`GeneratorExit` を捕捉するのではなく `try: ... finally:` するようお勧めします。

これらの変更の合わせ技で、ジェネレータは情報の一方的な生産者から、生産者かつ消費者という存在に変貌を遂げたのです。

ジェネレータは **コルーチン** という、より一般化された形式のサブルーチンにもなります。サブルーチンは一カ所 (関数の冒頭) から入って別の一カ所 (`return` 文) から出るだけですが、コルーチンはいろいろな場所 (`yield` 文) から入ったり出たり再開したりできるのです。

## 5 組み込み関数 (built-in function)

よくイテレータと一緒に使うビルトイン関数について、もっと詳しく見ていきましょう。

Python のビルトイン関数 `map()` と `filter()` は、機能がジェネレータ式と重複しています:

`map(f, iterA, iterB, ...)` は以下のシーケンスのイテレータを返します

```
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']
```

もちろん、リスト内包表記でも同じ結果が得られます。

`filter(predicate, iter)` はある条件を満たす要素に渡るイテレータを返すので、同様にリスト内包表記で再現できます。**predicate** は、ある条件に対する真偽値を返す関数です; `filter()` で使うには、その関数の引数は一つでなければなりません。

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

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

これはリスト内包表記でも書けます:

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

`enumerate(iter, start=0)` はイテラブルの要素に順番に番号を振っていき、(*start* から数え始めたときの) 番号とそれぞれの要素を含む 2 タプルを返します。

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

`enumerate()` はよく、リストに対してループさせて、条件に合う所に印を付けていくときに使われます:

```
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)` はイテラブルの要素をすべて集めたリストを作り、ソートして返します。引数 *key* および *reverse* は、リストの `sort()` メソッドにそのまま渡されます。

```
>>> 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]
```

(次のページに続く)

(前のページからの続き)

```
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(ソートに関する詳細な論議は [sortinghowto](#) を参照)

組み込みの `any(iter)` および `all(iter)` は iterable の値の真偽を調べます。`any()` は要素のどれかが真値なら `True` を返し、`all()` は要素が全て真値なら `True` を返します:

```
>>> 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, ...)` はそれぞれの iterable から 1 つの要素を取り、それらをタプルに入れて返します:

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

`zip()` は結果を返す前に入力イテレータを全て消費してメモリ上に `list` を作成しません。代わりに要求されるたびに `tuple` を生成して返します。(この動作を技術的な用語で `lazy evaluation` (訳: 遅延評価) と呼びます。)

このイテレータの用途には、すべて同じ長さのイテラブルを想定しています。長さが違っていれば、出力されるストリームは一番短いイテラブルと同じ長さになります。

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

とは言え、これをやってしまうと長いイテレータから要素をひとつ無駄に多く取って捨ててしまうかもしれませんので、やめておいたほうが良いです。その捨てられた要素を抜かしてしまう危険があるので、もうそのイテレータはそれ以上使えなくなってしまうのです。

## 6 itertools モジュール

itertools モジュールには、よく使うイテレータや、イテレータ同士の連結に使う関数がたくさん含まれています。この章では、そのモジュールの内容を小さな例で紹介していきたいと思います。

このモジュールの関数を大まかに分けるとこうなります:

- 既存のイテレータに基づいて新しいイテレータを作る関数。
- イテレータの要素を引数として扱う関数。
- イテレータの出力から一部を取り出す関数。
- イテレータの出力をグループ分けする関数。

### 6.1 新しいイテレータを作る

`itertools.count(start, step)` は値の間隔が一定の無限ストリームを返します。オプションで開始する数 (デフォルトは 0) や数どうしの間隔 (デフォルトは 1) を与えられます:

```
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)` は与えられたイテラブルの内容をコピーして、その要素を最初から最後まで無限に繰り返していくイテレータを返します。

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

`itertools.repeat(elem, [n])` は、与えられた要素を  $n$  回返しますが、 $n$  がなければ永遠に返し続けます。

```
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, ...)` は任意の数のイテラブルを受け取って、最初のイテレータから要素をすべて返し、次に二番目から要素をすべて返し、ということを要素がなくなるまで続けます。

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

`itertools.islice(iter, [start], stop, [step])` は、イテレータのスライスをストリームで返します。`stop` 引数だけだと、最初の `stop` 個の要素を返します。開始インデックスを渡すと `stop-start` 個で、`step`



の値も渡せばそれに応じて要素を抜かします。Python における文字列やリストのスライスとは違って、マイナスの値は *start*, *stop*, *step* に使えません。

```
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])` はイテレータを複製します; 元のイテレータの内容を同じように返す、独立した *n* 個のイテレータを返すのです。*n* の値は、指定しなければ既定が 2 になっています。複製するには元のイテレータの内容を一部保存しておく必要がありますから、大きなイテレータから複製したうちの一つが他よりも進んでいってしまうと、大量のメモリを消費することがあります。

```
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 要素に対して関数を呼ぶ

いま使った `operator` モジュールには、Python の演算子に対応する関数が入っています。いくつか例を挙げると、`operator.add(a, b)` (二つの値を加算)、`operator.ne(a, b)` (`a != b` と同じ)、`operator.attrgetter('id')` (`.id` 属性を取得するコーラブルを返す) といった関数です。

`itertools.starmap(func, iter)` は、イテラブルがタプルのストリームを返すとみなして、そのタプルを引数にを使って `func` を呼びます:

```
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 要素を選択する

さらに別のグループとして、述語 (predicate) に基づいてイテレータの要素からサブセットを選び出す関数があります。

`itertools.filterfalse(predicate, iter)` は `filter()` とは反対に、述語が偽を返す要素をすべて返します:

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

`itertools.takewhile(predicate, iter)` は述語が真を返している間だけ要素を返します。一度でも述語が偽を返すと、イテレータは出力終了の合図をします。

```
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)` は、述語が真を返しているうちは要素を無視し、偽になってから残りの出力をすべて返します。

```
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)` は 2 つのイテレータを取り、どちらかのイテレータを使い果たすまでの、`selectors` が真となる要素に対応する `data` 要素だけを返します:

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

## 6.4 組合せ関数

`itertools.combinations(iterable, r)` は、`iterable` から `r`-tuple 選択する全ての組み合わせを提供するイテレータを返します

```
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)
```

(次のページに続く)

```
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)
```

それぞれのタプル内では、要素は *iterable* がそれを返したのと同じ順序を保ちます。例えば上の例であれば、1 はいつでも 2, 3, 4, 5 の前に来ます。似たような関数に `itertools.permutations(iterable, r=None)` があり、こちらはこの順序についての制約がなく、*r* 個選択する全ての順列を返します。

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

*r* を与えない場合は *iterable* の長さが使われます。つまり *iterable* の全ての要素を選んだ順列を返します。

これらの関数が生成する組み合わせは、位置が基準ですので、*iterable* の内容が一意でなくとも良いことに注目してください:

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

全く同じタプル ('a', 'a', 'b') が 2 度現れていますが、これは 2 つの 'a' が別の位置からのものだからです。

`itertools.combinations_with_replacement(iterable, r)` 関数は別の制約を取り払います: 一回の選択で同じ要素を繰り返し選んでも良い。概念的には、それぞれのタプルの最初のものとして一つ要素が選ばれ、続いて 2 つ目の選択のかわりにそれで置き換わります

```
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 要素をグループ分けする

最後に議題に上げる関数 `itertools.groupby(iter, key_func=None)` は、これまでで最も複雑です。`key_func(elem)` は、イテラブルから返ってきた要素それぞれのキー値を計算する関数です。この関数が指定されていなければ、キーは単に各要素そのものになります。

`groupby()` は、元になるイテラブルから同じキー値を持つ連続する要素を集めて、キー値とそのキーに対応する要素のイテレータの 2-タプルのストリームを返します。

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

`groupby()` は、元になるイテラブルの内容がキー値でソートされた状態で与えられることを想定しています。ここで、返されるイテレータ自体も元のイテラブルを使うということに注意してください。そのため、`iterator-1` の結果を読み終わるまでは `iterator-2` とそれに対応するキー値を要求することはできません。

## 7 functools モジュール

`functools` モジュールには、高階関数がいくつか入っています。**高階関数** とは、入力として関数を受け取って新たな関数を返す関数です。このモジュールで一番便利なツールは `functools.partial()` 関数です。

関数型スタイルのプログラムでは時折、既存の関数から一部のパラメータを埋めた変種を作りたくなることがあります。Python の関数 `f(a, b, c)` というものがあるとしてください; `f(1, b, c)` と同じ意味の `g(b, c)` という関数を作りたくなることがあります; つまり `f()` のパラメータを一つ埋めるわけです。これは「関数の部分適用」と呼ばれています。

`partial()` のコンストラクタは `(function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2)` という引数を取ります。できあがったオブジェクトはコーラブルなので、それを呼べば、引数の埋まった状態で `function` を実行したのと同じことになります。

以下にあるのは、小さいけれども現実的な一つの例です:

```
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])` はイテラブルの要素に対して次々に演算を実行していった最終結果を出すもので、それゆえ無限長イテラブルには適用できません。`func` には、要素を二つ取って値を一つ返す関数が入ります。`functools.reduce()` はイテレータが返す最初の二要素 A と B を取って `func(A, B)` を計算します。それから三番目の要素 C を要求して `func(func(A, B), C)` を計算すると、その結果をさらに四番目の要素と組み合わせて……ということをイテラブルが尽きるまで続けるのです。もしイテラブルが一つも値を返さなければ `TypeError` が発生します。初期値 `initial_value` があるときには、最初の計算で `func(initial_value, A)` が実行されます。

```
>>> 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
```

`operator.add()` を `functools.reduce()` で使うと、iterable の全要素を合計することになります。これは使用頻度が高いので、そのための `sum()` というビルトイン関数があります:

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

とはいえ、多くの場合 `functools.reduce()` を使うよりは、単に `for` ループを書いたほうがわかりやすくなります:

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

(次のページに続く)

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

関連する関数は `itertools.accumulate(iterable, func=operator.add)` です。この関数は同じ計算をしますが、最終結果を返すのではなく、`accumulate()` はそれぞれの中間結果を生成するイテレータが返り値となります。

```
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 operator モジュール

`operator` モジュールは、既に取り上げましたが、Python の演算子に対応する関数が入っているモジュールです。関数型スタイルのコードにおいて、演算を一つ実行するだけのくだらない関数を書かずに済むので、よく世話になります。

このモジュールの関数を一部だけ紹介しましょう：

- 数学演算子: `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, ...
- 論理演算子: `not_()`, `truth()`
- ビット演算子: `and_()`, `or_()`, `invert()`
- 比較: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, `ge()`
- オブジェクト識別: `is_()`, `is_not()`

ちゃんとした一覧は `operator` モジュールの文書をご覧ください。

## 8 小さな関数とラムダ式

関数型スタイルのプログラムを書いていると、述語として働いたり、何らかの形で要素をつなぎ合わせたりするミニサイズの関数を必要とすることがよくあります。

ちょうど良い関数がビルトインやモジュールで存在していれば、新しい関数を定義する必要はまったくありません：

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

しかし、欲しい関数がないなら書くしかありません。そうした小さな関数を書く方法の一つが `lambda` 式です。`lambda` は引数として複数のパラメータと それをつなぐ式を取り、その式の値を返す無名の関数を作ります:

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

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

もう一つの選択肢は、ふつうに `def` 文で関数を定義するだけです:

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

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

どちらのほうが良いのでしょうか。それは好みの問題です; 著者のスタイルとしてはできるだけ `lambda` を使わないようにしています。

そのようにしている理由の一つに、`lambda` は定義できる関数が非常に限られているという点があります。一つの式として算出できる結果にしなければいけませんので、`if... elif... else` や `try... except` のような分岐を持つことができないのです。`lambda` 文の中でたくさんのことをやろうとすると、ごちゃごちゃして読みにくい式になってしまいます。さて、次のコードは何をしているのでしょうか、素早くお答えください!

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

わかるにはわかるでしょうが、何がどうなっているのか紐解いていくには時間がかかるはずです。短い `def` 文で入れ子にすると、少し見通しが良くなりますが:

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

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

でも単純に `for` ループにすれば良かったのです:

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

あるいは `sum()` ビルトインとジェネレータ式でも良いですね:

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

多くの場合、`functools.reduce()` を使っているところは `for` ループに書き直したほうが見やすいです。

Fredrik Lundh は以前 `lambda` 利用のリファクタリングに関して以下の指針を提案したことがあります:



1. ラムダ関数を書く。
2. そのラムダが一体ぜんたい何をしているのかコメントで説明する。
3. そのコメントをしばらく研究して、本質をとらえた名前を考える。
4. ラムダをその名前で `def` 文に書き換える。
5. コメントを消す。

著者はこの指針を本当に気に入っていますが、こうしたラムダなしスタイルが他より優れているかどうかについて、異論は認めます。

## 9 更新履歴と謝辞

著者は提案の申し出や修正、様々なこの記事の草稿の助けをしてくれた以下の人々に感謝します: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1: posted June 30 2006.

Version 0.11: posted July 1 2006. Typo fixes.

Version 0.2: posted July 10 2006. Merged `genexp` and `listcomp` sections into one. Typo fixes.

Version 0.21: Added more references suggested on the tutor mailing list.

Version 0.30: Adds a section on the `functional` module written by Collin Winter; adds short section on the operator module; a few other edits.

## 10 参考資料

### 10.1 一般論

計算機プログラムの構造と解釈、Harold Abelson と Gerald Jay Sussman、Julie Sussman 著。この本は <https://mitpress.mit.edu/sicp> で見れます。この計算機科学の古典的な教科書では、2 章と 3 章でシーケンスとストリームを使ってプログラム内のデータフローを整理する方法について説明しています。この本では例に Scheme を使っていますが、これらの章で説明されている設計アプローチの多くは、関数型スタイルの Python コードにも適用できます。

<https://www.defmacro.org/ramblings/fp.html>: 関数プログラミングの一般的な入門で Java での例を利用して、長大な歴史の紹介があります。

[https://en.wikipedia.org/wiki/Functional\\_programming](https://en.wikipedia.org/wiki/Functional_programming): 関数プログラミングに関する一般的な内容の記事。

<https://en.wikipedia.org/wiki/Coroutine>: コルーチンに関する記事。

<https://en.wikipedia.org/wiki/Currying>: カリー化の概念に関する記事。

## 10.2 Python 特有の話

<https://gnosis.cx/TPiP/>: David Mertz's の本の最初の章 *Text Processing in Python* では文書処理のための関数プログラミングについて議論しています、この議論の節には "Utilizing Higher-Order Functions in Text Processing" というタイトルがついています。

Mertz は IBM の DeveloperWorks サイトにも関数型プログラミングに関する 3 部構成の記事を書いています; [part 1](#), [part 2](#), [part 3](#),

## 10.3 Python 文書

`itertools` モジュールの文書。

`functools` モジュールについてのドキュメント。

`operator` モジュールの文書。

**PEP 289**: "Generator Expressions"

**PEP 342**: "Coroutines via Enhanced Generators" describes the new generator features in Python 2.5.

## 索引

### P

Python Enhancement Proposals

PEP 289, [24](#)

PEP 342, [11](#), [24](#)