Python 良い慣用句、悪い慣用句

リリース *2.7.15*

Guido van Rossum
and the Python development team

2 月 16, 2019

Python Software Foundation
Email: docs@python.org

目 次

著者 Moshe Zadka

概要

この文書はチュートリアルのおまけと考えてもらって結構です。 Python をどう使うべきか、そして更に重要なこととして、どう使うべきで **ない** かを例示しています。

第1章　使うべきでない構文

Python の落とし穴は他言語と比べればほとんどないようなものですが、中には特殊な場面でしか役に立たなかったり、単に危険なだけの構文も存在しています。

1.1 from モジュール import *

1.1.1 関数定義の中で

関数定義内での `from モジュール import *` は **不正** です。多くの古い Python では不正としてチェックされないものの、だからと言って有効になるわけではありません。やり手の弁護士を雇って無罪になっても、潔白になれるわけではないのと同じですね。ですから絶対にそういう使い方をしないでください。不正にされないバージョンでも、コンパイラはどの名前がローカルでどの名前がグローバルなのか判然としなくなるので、関数の実行が遅くなってしまいます。Python 2.1 で、この構文は警告や、場合によってはエラーも出すようになりました。

1.1.2 モジュールレベルで

モジュールレベルで `from モジュール import *` を使うのは有効ではありますが、大抵は、やめたほうが良いですよ。理由のひとつとして、それをやると本来 Python が持っている大事な特徴を失ってしまうということが挙げられます。その特徴とは、トップレベルの名前がそれぞれどこで定義されているのか、エディタの検索機能だけでわかるというものです。それに、将来モジュールに関数やクラスが増えていくと、ややこしいことになりかねません。

ニュースグループで出てくる最低の質問には、なぜこのコード:

```
f = open("www")
f.read()
```

が動かないのか、というものがあります。もちろんこれで動きますよ (ただし "www" というファイルがあれば)。でもモジュールのどこかに `from os import *` があればダメです。`os` モジュールは、整数を返す `open()` 関数を持っているのです。これはとても便利ではありますが、ビルトインを隠してしまうのは、非常に不便な特徴のひとつと言えます。

モジュールがエクスポートする名前は確実にはわからないのですから、必要なものだけ `from モジュール import 名前1, 名前2` で取るか、モジュールから出さずに `import モジュール;print モジュール.name` としておいて必要に応じてアクセスするようにしましょう。

1.1.3 問題ない状況

`from モジュール import *` が問題とならない状況もあります:

- 対話的プロンプト。たとえば `from math import *` すれば Python がスーパー関数電卓に変身します。
- C のモジュールを Python のモジュールで拡張するとき。
- そのモジュールは `from import *` 可だ、と作者が言っているとき。

1.2 そのまんまな exec, execfile() と仲間たち

「そのまんま」という言葉は辞書を明示せずに使うという意味で、そういう構文ではコードが **その時点の** 環境に対して評価されます。これは `from import *` と同じ理由で危険です — 使っている最中の変数を土足で踏んで行って、コード全体をメチャクチャにしてしまう可能性があるからです。これは、とにかくやめてください。

悪い見本:

```
>>> for name in sys.argv[1:]:
>>>     exec "%s=1" % name
>>> def func(s, **kw):
>>>     for var, val in kw.items():
>>>         exec "s.%s=val" % var  # invalid!
>>> execfile("handler.py")
>>> handle()
```

良い見本:

```
>>> d = {}
>>> for name in sys.argv[1:]:
>>>     d[name] = 1
>>> def func(s, **kw):
>>>     for var, val in kw.items():
>>>         setattr(s, var, val)
>>> d={}
>>> execfile("handle.py", d, d)
>>> handle = d['handle']
>>> handle()
```

1.3 from モジュール import 名前 1, 名前 2

今回のは、これまでの「ダメ」よりかなり弱い「ダメ」ですが、やはりそれなりの理由がなければ、やめておいたほうが良いことに変わりありません。これが大抵うまくないのは、いつの間にか二つ別々の名前空間に住む一つのオブジェクトを持つことになるからです。一方の名前空間でそのバインディングが変更されたとき、もう一方のバインディングは変更されないので、食い違いができてしまいます。これが起こるのは、たとえば、モジュールを読み直したり、ランタイムで関数の定義を変更したときなどです。

悪い見本:

```
# foo.py
a = 1

# bar.py
from foo import a
if something():
    a = 2 # danger: foo.a != a
```

良い見本:

```
# foo.py
a = 1

# bar.py
import foo
if something():
    foo.a = 2
```

1.4 except:

Python には `except:` 節があり、これはあらゆる例外を捕捉します。 Python のエラーは **すべて** 例外を出しますから、 `except:` を使うと各種のプログラミングエラーがランタイムの問題のように見えてしまい、デバッグの邪魔になります。

以下のコードは、 `except:` をなぜ避けるべきなのかを示す良い例です:

```
try:
    foo = opne("file") # misspelled "open"
except:
    sys.exit("could not open file!")
```

この 2 行目は `NameError` を引き起こし、続く except 節で捕捉されます。プログラムがエラー終了しますが、実際のエラーが `"file"` とは何の関係もないのに、にプログラムが表示するエラーメッセージは `"file"` の読み込みにあったように誤解させます。

前述の例はこう書くべきでした:

```
try:
    foo = opne("file")
except IOError:
    sys.exit("could not open file")
```

このプログラムを実行した場合は、 Python は `NameError` のトレースバックを表示し、修正すべき問題を即座に明らかにしてくれます。

`except:` は **全て** の例外を補足します。これには `SystemExit`, `KeyboardInterrupt`, `GeneratorExit` (これはエラーではなく、通常はユーザーコードでキャッチするべきではない例外です) も含まれるので、ほとんど全ての場合に `except:` を利用してはいけません。 "通常の" すべてのエラーをキャッチする必要がある場合、たとえばコールバックを実行するようなフレームワークの場合は、全ての通常の例外の基底クラスである `Exception` をキャッチすることができます。不幸なことに、 Python 2.x ではサードパーティのコードが `Exception` を継承していない例外を発生させる可能性があるので、 `except:` を使ってキャッチしたくない例外を手動で再度 raise する必要がある場合があるかもしれません。

第2章　例外

例外は Python の有用な機能です。何か期待している以外のことが起これば例外を出す、という習慣を持つべきですが、それと同時に、何か対処できるときにだけ捕捉する、ということも習慣にしてください。

以下は非常にありがちな悪い見本です:

```
def get_status(file):
    if not os.path.exists(file):
        print "file not found"
        sys.exit(1)
    return open(file).readline()
```

ここで、 `os.path.exists()` を呼んでから `open()` を呼ぶまでの間にファイルが消された場合を考えてください。そうなれば最後の行は `IOError` を投げるでしょう。同じことは、 *file* は存在しているけれど読み出し権限がなかった、という場合にも起こります。これをテストする際、ふつうのマシンで、存在するファイルと存在しないファイルに対してだけやったのではバグがないように見えてしまい、テスト結果が大丈夫そうなのでコードはそのまま出荷されてしまうことになります。こうして、対処されない `IOError` (またはその他の `EnvironmentError`) はユーザの所まで逃げのびて、汚いトレースバックを見せることになるのです。

もっと良い方法はこちら:

```
def get_status(file):
    try:
        return open(file).readline()
    except EnvironmentError as err:
        print "Unable to open file: {}".format(err)
        sys.exit(1)
```

このバージョンでは、ファイルが開かれて一行目も読まれる (だから、あてにならない NFS や SMB 接続でも動く) か、あるいはファイルを開くのに失敗した理由についての全ての情報を含むエラーメッセージを表示してアプリケーションを強制終了するかの **いずれか** 一方しか起こりません。

とはいえ、このバージョンの `get_status()` でさえ、前提としている条件が多過ぎます — すぐ終わるスクリプトでだけ使って、長く動作させる、いわゆるサーバでは使わない前提なのです。もちろん呼び出し側はこうすることもできます:

```
try:
    status = get_status(log)
except SystemExit:
    status = None
```

でも、もっと良い方法があります。コードで使う `except` 節を、できるだけ少なくするのです — 使うとすれば、必ず成功するはずの呼び出し内か、main 関数での全捕捉ですね。

というわけで、たぶんもっと良い `get_status()` はこちら:

```
def get_status(file):
    return open(file).readline()
```

呼び出した側は、望むなら例外を処理することもできますし (たとえばループで複数ファイルに試行するときとか)、そのまま **自分の** 呼び出し親まで上げることもできます。

しかし、この最終バージョンにも深刻な問題があります — CPython 実装の細部に原因があるのですが、例外が起きたときにはその例外ハンドラが終了するまでファイルが閉じられないのです; しかも、なお悪いことに、他の実装 (たとえば Jython) では例外の有無に関わらず閉じられません。

この関数の一番良いバージョンでは `open()` をコンテクストマネジャとして使って、関数が返るとすぐにファイルが閉じられるようにしています:

```
def get_status(file):
    with open(file) as fp:
        return fp.readline()
```

第3章　使い捨てじゃなくて充電池を使う

どうも皆、Python ライブラリに最初からあるものを自分で書こうとして、大抵うまくいっていないようです。そういう場当たりなモジュールには貧弱なインタフェースしかないことを考えると、ふつうは Python に付いてくる高機能な標準ライブラリとデータ型を使うほうが、自分でひねり出すより格段に良いですよ。

便利なのにほとんど知られていないモジュールに `os.path` があります。このモジュールには OS に合ったパス演算が備わっていて、大抵は自分でどれだけ苦労して作ったものよりもずっと良いものです。

比べてください:

```
# ugh!
return dir+"/"+file
# better
return os.path.join(dir, file)
```

`os.path` にはさらに便利な関数が他にもあります: `basename()` や `dirname()`, `splitext()` などです。

There are also many useful built-in functions people seem not to be aware of for some reason: `min()` and `max()` can find the minimum/maximum of any sequence with comparable semantics, for example, yet many people write their own `max()`/`min()`. Another highly useful function is `reduce()` which can be used to repeatedly apply a binary operation to a sequence, reducing it to a single value. For example, compute a factorial with a series of multiply operations:

```
>>> n = 4
>>> import operator
>>> reduce(operator.mul, range(1, n+1))
24
```

数値の字句解析をするときには、 `float()`, `int()`, `long()` は全て文字列の引数を受け取って、不正なフォーマットの文字列の場合には `ValueError` を発生させることを覚えておくと便利です。

第4章　バックスラッシュで文を続ける

Python は改行を文の終わりとして扱いますので、そして文は一行にうまく収まらないことがよくありますので、こうする人が多いです:

```
if foo.bar()['first'][0] == baz.quux(1, 2)[5:9] and \
   calculate_number(10, 20) != forbulate(500, 360):
      pass
```

これは危険だということに気づいたほうが良いですよ: はぐれスペースが \ の後に来ればその行の意味が変わってしまいますが、スペースはエディタで見えにくいことに定評があるのです。今回の場合は構文エラーにはなりますが、もしこうなら:

```
value = foo.bar()['first'][0]*baz.quux(1, 2)[5:9] \
        + calculate_number(10, 20)*forbulate(500, 360)
```

微妙に違う意味になるだけで、エラーが出ません。

それで、ふつうは括弧に入れて暗黙のうちに行をつなげるほうが賢明です:

このバージョンで鉄壁です:

```
value = (foo.bar()['first'][0]*baz.quux(1, 2)[5:9]
        + calculate_number(10, 20)*forbulate(500, 360))
```

索引