Idioms and Anti-Idioms in Python

Release 2.7.18

Guido van Rossum
and the Python development team

May 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 Language Constructs You Should Not Use **2**
- 1.1 from module import * . . . 2
- 1.2 Unadorned `exec`, `execfile()` and friends . . . 3
- 1.3 from module import name1, name2 . . . 3
- 1.4 except: . . . 4

2 Exceptions **4**

3 Using the Batteries **5**

4 Using Backslash to Continue Statements **6**

Index **7**

Author Moshe Zadka

This document is placed in the public domain.

Abstract

This document can be considered a companion to the tutorial. It shows how to use Python, and even more importantly, how *not* to use Python.

1 Language Constructs You Should Not Use

While Python has relatively few gotchas compared to other languages, it still has some constructs which are only useful in corner cases, or are plain dangerous.

1.1 from module import *

Inside Function Definitions

`from module import *` is *invalid* inside function definitions. While many versions of Python do not check for the invalidity, it does not make it more valid, no more than having a smart lawyer makes a man innocent. Do not use it like that ever. Even in versions where it was accepted, it made the function execution slower, because the compiler could not be certain which names were local and which were global. In Python 2.1 this construct causes warnings, and sometimes even errors.

At Module Level

While it is valid to use `from module import *` at module level it is usually a bad idea. For one, this loses an important property Python otherwise has — you can know where each toplevel name is defined by a simple "search" function in your favourite editor. You also open yourself to trouble in the future, if some module grows additional functions or classes.

One of the most awful questions asked on the newsgroup is why this code:

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

does not work. Of course, it works just fine (assuming you have a file called "www".) But it does not work if somewhere in the module, the statement `from os import *` is present. The `os` module has a function called `open()` which returns an integer. While it is very useful, shadowing a builtin is one of its least useful properties.

Remember, you can never know for sure what names a module exports, so either take what you need — `from module import name1, name2`, or keep them in the module and access on a per-need basis — `import module;print module.name`.

When It Is Just Fine

There are situations in which `from module import *` is just fine:

- The interactive prompt. For example, `from math import *` makes Python an amazing scientific calculator.
- When extending a module in C with a module in Python.
- When the module advertises itself as `from import *` safe.

1.2 Unadorned `exec`, `execfile()` and friends

The word "unadorned" refers to the use without an explicit dictionary, in which case those constructs evaluate code in the *current* environment. This is dangerous for the same reasons `from import *` is dangerous — it might step over variables you are counting on and mess up things for the rest of your code. Simply do not do that.

Bad examples:

```
>>> 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()
```

Good examples:

```
>>> 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 module import name1, name2

This is a "don't" which is much weaker than the previous "don't"s but is still something you should not do if you don't have good reasons to do that. The reason it is usually a bad idea is because you suddenly have an object which lives in two separate namespaces. When the binding in one namespace changes, the binding in the other will not, so there will be a discrepancy between them. This happens when, for example, one module is reloaded, or changes the definition of a function at runtime.

Bad example:

```
# foo.py
a = 1

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

Good example:

```
# foo.py
a = 1

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

1.4 except:

Python has the `except:` clause, which catches all exceptions. Since *every* error in Python raises an exception, using `except:` can make many programming errors look like runtime problems, which hinders the debugging process.

The following code shows a great example of why this is bad:

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

The second line triggers a `NameError`, which is caught by the except clause. The program will exit, and the error message the program prints will make you think the problem is the readability of `"file"` when in fact the real error has nothing to do with `"file"`.

A better way to write the above is

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

When this is run, Python will produce a traceback showing the `NameError`, and it will be immediately apparent what needs to be fixed.

Because `except:` catches *all* exceptions, including `SystemExit`, `KeyboardInterrupt`, and `GeneratorExit` (which is not an error and should not normally be caught by user code), using a bare `except:` is almost never a good idea. In situations where you need to catch all "normal" errors, such as in a framework that runs callbacks, you can catch the base class for all normal exceptions, `Exception`. Unfortunately in Python 2.x it is possible for third-party code to raise exceptions that do not inherit from `Exception`, so in Python 2.x there are some cases where you may have to use a bare `except:` and manually re-raise the exceptions you don't want to catch.

2 Exceptions

Exceptions are a useful feature of Python. You should learn to raise them whenever something unexpected occurs, and catch them only where you can do something about them.

The following is a very popular anti-idiom

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

Consider the case where the file gets deleted between the time the call to `os.path.exists()` is made and the time `open()` is called. In that case the last line will raise an `IOError`. The same thing would happen if *file* exists but has no read permission. Since testing this on a normal machine on existent and non-existent files makes it seem bugless, the test results will seem fine, and the code will get shipped. Later an unhandled `IOError` (or perhaps some other `EnvironmentError`) escapes to the user, who gets to watch the ugly traceback.

Here is a somewhat better way to do it.

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

In this version, *either* the file gets opened and the line is read (so it works even on flaky NFS or SMB connections), or an error message is printed that provides all the available information on why the open failed, and the application is aborted.

However, even this version of `get_status()` makes too many assumptions — that it will only be used in a short running script, and not, say, in a long running server. Sure, the caller could do something like

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

But there is a better way. You should try to use as few `except` clauses in your code as you can — the ones you do use will usually be inside calls which should always succeed, or a catch-all in a main function.

So, an even better version of `get_status()` is probably

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

The caller can deal with the exception if it wants (for example, if it tries several files in a loop), or just let the exception filter upwards to *its* caller.

But the last version still has a serious problem — due to implementation details in CPython, the file would not be closed when an exception is raised until the exception handler finishes; and, worse, in other implementations (e.g., Jython) it might not be closed at all regardless of whether or not an exception is raised.

The best version of this function uses the `open()` call as a context manager, which will ensure that the file gets closed as soon as the function returns:

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

3 Using the Batteries

Every so often, people seem to be writing stuff in the Python library again, usually poorly. While the occasional module has a poor interface, it is usually much better to use the rich standard library and data types that come with Python than inventing your own.

A useful module very few people know about is `os.path`. It always has the correct path arithmetic for your operating system, and will usually be much better than whatever you come up with yourself.

Compare:

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

More useful functions in `os.path`: `basename()`, `dirname()` and `splitext()`.

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
```

When it comes to parsing numbers, note that `float()`, `int()` and `long()` all accept string arguments and will reject ill-formed strings by raising an `ValueError`.

4 Using Backslash to Continue Statements

Since Python treats a newline as a statement terminator, and since statements are often more than is comfortable to put in one line, many people do:

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

You should realize that this is dangerous: a stray space after the `\` would make this line wrong, and stray spaces are notoriously hard to see in editors. In this case, at least it would be a syntax error, but if the code was:

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

then it would just be subtly wrong.

It is usually much better to use the implicit continuation inside parenthesis:

This version is bulletproof:

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

Index

B

bare except, 4

E

except
    bare, 4