Idioms and Anti-Idioms in Python

Release 2.6.1

Guido van Rossum
Fred L. Drake, Jr., editor

Contents

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 then 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 are local and which are 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 question 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 builtins 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 then 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 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:

```python
# foo.py
a = 1

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

Good example:

```python
# 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, this makes many programming errors look like runtime problems, and hinders the debugging process.

The following code shows a great example:

```python
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 you will have no idea that this has nothing to do with the readability of `"file"`.

The example above is better written

```python
try:
    foo = opne("file") # will be changed to "open" as soon as we run it
```

```
except IOError:
    sys.exit("could not open file")
```

There are some situations in which the `except:` clause is useful: for example, in a framework when running callbacks, it is good not to let any callback disturb the framework.

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 the file gets deleted between the time the call to `os.path.exists()` is made and the time `open()` is called. That means the last line will throw an `IOError`. The same would happen if *file* exists but has no read permission. Since testing this on a normal machine on existing and non-existing files make it seem bugless, that means in testing the results will seem fine, and the code will get shipped. Then an unhandled `IOError` escapes to the user, who has to watch the ugly traceback.

Here is a better way to do it.

```
def get_status(file):
    try:
        return open(file).readline()
    except (IOError, OSError):
        print "file not found"
        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 the message is printed and the application aborted.

Still, `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
```

So, try to make as few `except` clauses in your code — those will usually be a catch-all in the `main()`, or inside calls which should always succeed.

So, the best version 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.

The last version is not very good either — due to implementation details, the file would not be closed when an exception is raised until the handler finishes, and perhaps not at all in non-C implementations (e.g., Jython).

```python
def get_status(file):
    fp = open(file)
    try:
        return fp.readline()
    finally:
        fp.close()
```

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 then 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 then whatever you come up with yourself.

Compare:

```python
# 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 builtin 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()`. A classical use of `reduce()` is something like

```python
import sys, operator
nums = map(float, sys.argv[1:])
print reduce(operator.add, nums)/len(nums)
```

This cute little script prints the average of all numbers given on the command line. The `reduce()` adds up all the numbers, and the rest is just some pre- and postprocessing.

On the same note, note that `float()`, `int()` and `long()` all accept arguments of type string, and so are suited to parsing — assuming you are ready to deal with the `ValueError` they raise.

4 Using Backslash to Continue Statements

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

```python
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:

```python
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))
```