• Home Page •

The Global Object Pattern ¶

Verdict

> Like several other scripting languages, Python parses the outer level of each module as normal code. Un-indented assignment statements, expressions, and even loops and conditionals will execute as the module is imported. This presents an excellent opportunity to supplement a module's classes and functions with constants and data structures that callers will find useful — but also offers dangerous temptations: mutable global objects can wind up coupling distant code, and I/O operations impose import-time expense and side effects.

Every Python module is a separate namespace. A module like `json` can offer a `loads()` function without conflicting with, replacing, or overwriting the completely different `loads()` function defined over in the `pickle` module.

Separate namespaces are crucial to making a programming language tractable. If Python modules were not separate namespaces, you would be unable to read or write Python code by keeping your attention focused on the module in front of you — a line of code might use, or accidentally conflict with, a name defined anywhere else in the Standard Library or a third-party module you have installed. Upgrading a third-party module could break your entire program if the new version defined a new global that conflicted with yours. Programmers who are forced to code in a language without namespaces soon find themselves festooning global names with prefixes, suffixes, and extra punctuation in a desperate race to keep them from conflicting.

While every function and class is, of course, an object — in Python, everything is an object — the Module Global pattern more specifically refers to normal object instances that are given names at the global level of a module.

Two patterns use Module Globals but are important enough to warrant their own articles:

- Prebound Methods are generated when a module builds an object and then assigns one or more of the object's bound methods to names at the module's global level. The names can be used to call the methods later without needing to find the object itself.
- While a Sentinel Object doesn't have to live in a module's global namespace — some sentinel objects are defined as class attributes, while others are private and live inside of a closure — many sentinels, both in the Standard Library and

beyond, are defined and accessed as module globals.

This article will cover some other common cases.

The Constant Pattern ¶

Modules often assign useful numbers, strings, and other values to names in their global scope. The Standard Library includes many such assignments, from which we can excerpt a few examples.

```
January = 1                     # calendar.py
WARNING = 30                    # logging.py
MAX_INTERPOLATION_DEPTH = 10    # configparser.py
SSL_HANDSHAKE_TIMEOUT = 60.0    # asyncio.constants.py
TICK = "'"                      # email.utils.py
CRLF = "\r\n"                   # smtplib.py
```

Remember that these are “constants” only in the sense that the objects themselves are immutable. The names can still be reassigned.

```
import calendar
calendar.January = 13
print(calendar.January)
```

```
13
```

Or deleted, for that matter.

```
del calendar.January
print(calendar.January)
```

```
Traceback (most recent call last):
  ...
AttributeError: module 'calendar' has no attribute 'January'
```

In addition to integers, floats, and strings, constants also include immutable containers like tuples and frozen sets:

```
all_errors = (Error, OSError, EOFError)  # ftplib.py
bytes_types = (bytes, bytearray)         # pickle.py
DIGITS = frozenset("0123456789")         # sre_parse.py
```

More specialized immutable data types also serve as constants:

```
_EPOCH = datetime(1970, 1, 1, tzinfo=timezone.utc)  # datetime
```

On rare occasions, a module global which the code clearly never intends to modify uses a mutable data structure anyway. Plain mutable sets are common in code that pre-dates the invention of the `frozenset`. Dictionaries are still used today because, alas, the Standard Library doesn’t offer a frozen dictionary.

```
# socket.py
_blocking_errnos = { EAGAIN, EWOULDBLOCK }
```

```
# locale.py
windows_locale = {
  0x0436: "af_ZA", # Afrikaans
  0x041c: "sq_AL", # Albanian
  0x0484: "gsw_FR",# Alsatian - France
  ...
  0x0435: "zu_ZA", # Zulu
}
```

Constants are often introduced as a refactoring: the programmer notices that the same value `60.0` is appearing repeatedly in their code, and so introduces a constant `SSL_HANDSHAKE_TIMEOUT` for the value instead. Each use of the name will now incur the slight cost of a search into the global scope, but this is balanced by a couple of advantages. The constant's name now documents the value's meaning, improving the code's readability. And the constant's assignment statement now provides a single location where the value can be edited in the future without needing to hunt through the code for each place `60.0` was used.

These advantages are weighty enough that a constant is sometimes introduced even for a value that's used only once, hoisting a literal that was hidden deep in the code up into visibility as a global.

Some programmers place constant assignments close to the code that use them; others put all constants at the top of the file. Unless a constant is placed so close to its code that it will always be in view of human readers, it can be more friendly to put constants at the top of the module for the easy reference of readers who haven't yet configured their editors to support jump-to-definition.

Another kind of constant is not directed inwards, towards the code in the module itself, but outwards as part of the module's advertised API. A constant like `WARNING` from the `logging` module offers the advantages of a constant to the caller: code will be more readable, and the constant's value could be adjusted later without every caller needing to edit their code.

You might expect that a constant intended for the module's own use, but not intended for callers, would always start with an underscore to mark it private. But Python programmers are not consistent in marking constants private, perhaps because the cost of needing to keep a constant around forever because a caller might have decided to start using it is smaller than the cost of having a helper function or class's API forever locked up.

Import-time computation ¶

Sometimes constants are introduced for efficiency, to avoid recomputing a value every time code is called. For example, even though math operations involving literal numbers are in fact optimized away in all modern Python implementations, developers

often still feel more comfortable making it explicit that the math should be done at import time by assigning the result to a module global:

```
# zipfile.py
ZIP_FILECOUNT_LIMIT = (1 << 16) - 1
```

When the math expression is complicated, assigning a name also enhances the code's readability.

As another example, there exist special floating point values that cannot be written in Python as literals; they can only be generated by passing a string to the float type. To avoid calling `float()` with `'nan'` or `'inf'` every single time such a value is needed, modules often build such values only once as module globals.

```
# encoder.py
INFINITY = float('inf')
```

A constant can also capture the result of a conditional to avoid re-evaluating it each time the value is needed — as long, of course, as the condition won't be changing while the program is running.

```
# shutil.py
COPY_BUFSIZE = 1024 * 1024 if _WINDOWS else 16 * 1024
```

My favorite example of computed constants in the Standard Library is the `types` module. I had always assumed it was implemented in C, to gain special access to built-in type objects like `FunctionType` and `LambdaType` that are defined by the language implementation itself.

It turns out? I was wrong. The `types` module is written in plain Python!

Without any special access to language internals, it does what anyone else would do to learn what type functions have. It creates a function, then asks its type:

```
# types.py
def _f(): pass
FunctionType = type(_f)
```

On the one hand, this makes the `types` module seem almost superfluous — you could always use the same trick to discover `FunctionType` yourself. But on the other hand, importing it from `types` lets both major benefits of the Constant Pattern shine: code becomes more readable because `FunctionType` will have the same name everywhere, and more efficient because the constant only needs to be computed once no matter how many modules in a large system might use it.

Dunder Constants ¶

A special case of constants defined at a module's global level are "dunder" constants whose names start and end with double underscores.

Several Module Global dunder constants are set by the language itself. For the official list, look for the "Modules" subheading in the Python Reference's section on the standard type hierarchy. The two encountered most often are `__name__`, which programs need to check because of Python's awful design decision to assign the fake name `'__main__'` to the module invoked from the command line, and `__file__`, the full filesystem path to the module's Python file itself — which is almost universally used to find data files included in a package, even though the official recommendation these days is to use `pkgutil.get_data()` instead.

```
here = os.path.dirname(__file__)
```

Beyond the dunder constants set by the language runtime, there is one Python recognizes if a module chooses to set it: if `__all__` is assigned a sequence of identifiers, then only those names will be imported into another module that does `from … import *`. You might have expected `__all__` to become less popular as `import *` gained a reputation as an anti-pattern, but it has gained a happy second career limiting the list of symbols included by automatic documentation engines like Sphinx autodoc module.

Even though most modules never plan to modify `__all__`, they inexplicably specify it as a Python list. It is more elegant to use a tuple.

Beyond these official dunder constants, some modules — despite unattractive how many people find dunder names — indulge in the creation of even more. Assignments to names like `__author__` and `__version__` are scattered across the Standard Library and beyond. While they don't appear consistently enough for tooling can assume their presence, occasional readers probably find them informative, and they're easier to get to than official package metadata.

Beware that there does not seem to be agreement, even within the Standard Library, about what type `__author__` should have.

```
# bz2.py
__author__ = "Nadeem Vawda <nadeem.vawda@gmail.com>"
```

```
# inspect.py
__author__ = ('Ka-Ping Yee <ping@lfw.org>',
              'Yury Selivanov <yselivanov@sprymix.com>')
```

Why not `author` and `version` instead, without the dunders? An early reader probably misunderstood dunders, which really meant "special to the Python language runtime," as a vague indication that a value was module metadata rather than module code. A few Standard Library modules do offer their version without dunders, but without even agreeing on the capitalization.

```
VERSION = "1.3.0"  # etree/ElementTree.py
version = "0.20"   # sax/expatreader.py
version = "0.9.0"  # tarfile.py
```

To avoid the inconsistencies surrounding these informal and ad-hoc metadata conventions, a package that expects to be installed with `pip` can learn the names and versions of other installed packages directly from the Python package installation system. More information is available in the setuptools documentation on the `pkg_resources` module.

The Global Object Pattern ¶

In the full-fledged Global Object pattern, as in the Constant pattern, a module instantiates an object at import time and assigns it a name in the module's global scope. But the object does not simply serve as data; it is not merely an integer, string, or data structure. Instead, the object is made available for the sake of the methods it offers — for the actions it can perform.

The simplest Global Objects are immutable. A common example is a compiled regular expression — here are a few examples from the Standard Library:

```
escapesre = re.compile(r'[\\"]')        # email/utils.py
magic_check = re.compile('([*?[])')     # glob.py
commentclose = re.compile(r'--\s*>')    # html/parser.py
HAS_UTF8 = re.compile(b'[\x80-\xff]')   # json/encoder.py
```

Compiling a regular expression as a module global is a good example of the more general Global Object pattern. It achieves an elegant and safe transfer of expense from later in a program's runtime to import time instead. The tradeoffs are:

- The cost of importing the module increases by the cost of compiling the regular expression (plus the tiny cost of assigning it to a global name).
- The import-time cost is now borne by every program that imports the module. Even if a program doesn't happen to call any code that uses the `HAS_UTF8` regular expression shown above, it will incur the expense of compiling it whenever it imports the `json` module. (Plot twist: in Python 3, the pattern is no longer even used in the module! But its name was not marked private with a leading underscore, so I suppose it's not safe to remove — and every `import json` gets to pay its cost forever?)
- But functions and methods that do, in fact, need to use the regular expression will no longer incur a repeated cost for its compilation. The compiled regular expression is ready to start scanning a string immediately! If the regular expression is used frequently, like in the inner loop of a costly operation like parsing, the savings can be considerable.
- The global name will make calling code more readable than if the regular expression, when used locally, is used anonymously in a larger expression. (If readability is the only concern, though, remember that you can define the regular expression's string as a global but skip the cost of compiling it at module level.)

This list of tradeoffs is about the same, by the way, if you move a regular expression out

into a class attribute instead of moving it all the way out to the global scope. When I finally get around to writing about Python and classes, I'll link from here to further thoughts on class attributes.

Global Objects that are mutable ¶

But what about Global Objects that are mutable?

They are easiest to justify when they wrap system resources that are by their nature also global to an operating system process. One example in the Standard Library itself is the `environ` object that gives your Python program the "environment" — the text keys and values supplying your timezone, terminal type, so forth — that was passed to your Python program from its parent process.

Now, it is arguable whether your program should really be writing new values into its environment as it runs. If you're launching a subprocess that needs an environment variable adjusted, the `subprocess` routines offer an `env` parameter. But if code does need to manipulate this global resource, then it makes sense for that access to be mediated by a correspondingly global Python object:

```
# os.py
environ = _createenviron()
```

Through this global object, the various routines, and perhaps threads, in a Python program coordinate their access to this process-wide resource. Any change:

```
import os
os.environ['TERM'] = 'xterm'
```

— will be immediately visible to any other part of the program that reads that environment key:

```
>>> os.environ['TERM']
'xterm'
```

The problems with coupling distant parts of your codebase, and even unrelated parts of different libraries, through a unique global object are well known.

- Tests that were previously independent are suddenly coupled through the global object and can no longer safely be run in parallel. If one test makes a temporary assignment to `environ['PATH']` just before another test launches a binary with `subprocess`, the binary will inherit the test value of `$PATH` — possibly causing an error.
- You can sometimes serialize access to a global object through a lock. But unless you do a thorough audit of all of the libraries your code uses, and continue to audit them when upgrading to new versions, it can be difficult to even know which tests call code that ultimately touches particular global object like `environ`.

- Even tests run serially, not in parallel, will now wind up coupled if one test fails to restore `environ` to its original state before the next test runs. This can, it's true, be mitigated with teardown routines or with mocks that automatically restore state. But unless every single test is perfectly cautious, your test suite can still suffer from exceptions that depend on random test ordering or on whether a previous test succeeded or exited early.
- These dangers beset not only tests but production runs as well. Even if your application doesn't launch multiple threads, there can be surprising cases where a refactoring winds up calling code that performs one operation on `environ` right in the middle of another routine that was also in the middle of transforming its state.

The Standard Library has more examples of the Mutable Global pattern — both public globals and private ones litter its modules. Some correspond to unique resources at the system level:

```
# Lib/multiprocessing/process.py
_current_process = _MainProcess()
_process_counter = itertools.count(1)
```

Others correspond to no outside resource but instead serve as single points of coordination for a process-wide activity like logging:

```
# Lib/logging/__init__.py
root = RootLogger(WARNING)
```

Third-party libraries can supply dozens of more examples, from global HTTP thread pools and database connections to registries of request handlers, library plugins, and third-party codecs. But in every case, the Mutable Global courts all of the dangers listed above in return for the convenience of putting a resource where every module can reach it.

My advice, to the extent that you can, is to write code that accepts arguments and returns values computed from them. Failing that, try passing database connections or open sockets to code that will need to interact with the outside world. It is a compromise for code that finds itself stranded from the resources it needs to resort to accessing a global.

The glory of Python, of course, is that it usually makes even anti-patterns and compromises read fairly elegantly in code. An assignment statement at the global level of a module is as easy to write and read as any other assignment statement, and callers can access the Mutable Global through exactly the same import statement they use for functions and classes.

Import-time I/O ¶

Many of the worst Global Objects are those that perform file or network I/O at import

time. They not only impose the cost of that I/O on every library, script, and test that need the module, but expose them to failure if a file or network is not available.

Library authors have an unfortunate tendency to make assumptions like "the file `/etc/hosts` will always exist" when, in fact, they can't know ahead of time all the exotic environments their code will one day face — maybe a tiny embedded system that in fact lacks that file; maybe a continuous integration environment spinning up containers that lack any network configuration at all.

Even when faced with this possibility, a module author might still try to defend their import-time I/O: "But delaying the I/O until after import time simply postpones the inevitable — if the system doesn't have `/etc/hosts` then the user will get exactly the same exception later anyway." The attempt to make this excuse reveals three misunderstandings:

1. Errors at import time are far more serious than errors at runtime. Remember that at the moment your package is imported, the program's main routine has probably not started running — the caller is usually still up in the middle of the stack of `import` statements at the top of their file. They have probably not yet set up logging and have not yet entered their application's main `try…except` block that catches and reports failures, so any errors during import will probably print directly to the standard output instead of getting properly reported.
2. Applications are often written to survive the failure of some operations so that in an emergency they can still perform other functions. Even if features that need your library will now hit an exception, the application might have many others it can continue to offer — or could, if you didn't kill it with an exception at import time.
3. Finally, library authors need to keep in mind that a Python program that imports their library might not even use it! Never assume that simply because your code has been imported, it will be used. There are many situations where a module gets imported incidentally, as the dependency of yet further modules, but never happens to get called. By performing I/O at import time, you could impose expense and risk on hundreds of programs and tests that don't even need or care about your network port, connection pool, or open file.

For all of these reasons, it's best for your global objects to wait until they're first called before opening files and creating sockets — because it's at the moment of that first call that the library knows the main program is now up and running, and knows that its services are in fact definitely needed in this particular run of the program.

I'll admit that, when my package needs to load a small data file that's embedded in the package itself, I do sometimes break this rule.
