• Home Page •

The Singleton Pattern ¶

A "Creational Pattern" from the Gang of Four book

Verdict

> Python programmers almost never implement the Singleton Pattern as described in the Gang of Four book, whose Singleton class forbids normal instantiation and instead offers a class method that returns the singleton instance. Python is more elegant, and lets a class continue to support the normal syntax for instantiation while defining a custom `__new__()` method that returns the singleton instance. But an even more Pythonic approach, if your design forces you to offer global access to a singleton object, is to use The Global Object Pattern instead.

Disambiguation ¶

Python was already using the term *singleton* before the "Singleton Pattern" was defined by the object oriented design pattern community. So we should start by distinguishing the several meanings of "singleton" in Python.

1. A tuple of length one is called a *singleton.* While this definition might surprise some programmers, it reflects the original definition of a singleton in mathematics: a set containing exactly one element. The Python Tutorial itself introduces newcomers to this definition when its chapter on Data Structures calls a one-element tuple a "singleton" and the word continues to be used in that sense through the rest of Python's documentation. When the Extending and Embedding guide says, "To call the Python function ... with one argument, pass a singleton tuple," it means a tuple containing exactly one item.
2. Modules are "singletons" in Python because `import` only creates a single copy of each module; subsequent imports of the same name keep returning the same module object. For example, when the Module Objects chapter of the Python/C API Reference Manual asserts that "Single-phase initialization creates singleton modules," it means by a "singleton module" a module for which only one object is ever created.
3. A "singleton" is a class instance that has been assigned a global name through The Global Object Pattern. For example, the official Python Programming FAQ answers the question "How do I share global variables across modules?" with the assertion that in Python "using a module is also the basis for implementing the Singleton design" — because not only can a module's global namespace store

constants (the FAQ's example is `x = 0` shared between several modules), but mutable class instances as well.

4. Individual flyweight objects that are examples of The Flyweight Pattern are often called "singleton" objects by Python programmers. For example, a comment inside the Standard Library's `itertoolsmodule.c` asserts that "CPython's empty tuple is a singleton" — meaning that the Python interpreter only ever creates a single empty tuple object, which `tuple()` returns over and over again every time it's passed a zero-length sequence. A comment in `marshal.c` similarly refers to the "empty frozenset singleton." But neither of these singleton objects is an example of the Gang of Four's Singleton Pattern, because neither object is the sole instance of its class: `tuple` lets you build other tuples besides the empty tuple, and `frozenset` lets you build other frozen sets. Similarly, the `True` and `False` objects are a pair of flyweights, not examples of the Singleton Pattern, because neither is the sole instance of `bool`.
5. Finally, Python programmers on a few rare occasions do actually mean "The Singleton Pattern" when they call an object a "singleton": the lone object returned by its class every time the class is called.

The Python 2 Standard Library included no examples of the Singleton Pattern. While it did feature singleton objects like `None` and `Ellipsis`, the language provided access to them through the more Pythonic Global Object Pattern by giving them names in the `__builtin__` module. But their classes were not callable:

```
>>> type(None)
<type 'NoneType'>
>>> NoneType = type(None)
>>> NoneType()
TypeError: cannot create 'NoneType' instances
>>> type(Ellipsis)()
TypeError: cannot create 'ellipsis' instances
```

In Python 3, however, the classes were upgraded to use the Singleton Pattern:

```
>>> NoneType = type(None)
>>> print(NoneType())
None
>>> type(Ellipsis)()
Ellipsis
```

This makes life easier for programmers who need a quick callable that always returns `None`, though such occasions are rare. In most Python projects these classes are never called and the benefit remains purely theoretical. When Python programmers need the `None` object they use The Global Object Pattern and simply type its name.

The Gang of Four's implementation ¶

The C++ language that the Gang of Four were targeting imposed a distinct syntax on object creation, that looked something like:

```
# Object creation in a language
# that has a "new" keyword.

log = new Logger()
```

A line of C++ that says `new` always creates a new class instance — it never returns a singleton. In the presence of this special syntax, what were their options for offering singleton objects?

1. The Gang of Four did not take the easy way out and use The Global Object Pattern because it did not work particularly well in early versions of the C++ language. There, global names all shared a single crowded global namespace, so elaborate naming conventions were necessary to prevent names from different libraries from colliding. The Gang judged that adding both a class and its singleton instance to the crowded global namespace would be excessive. And since C++ programmers couldn't control the order in which global objects were initialized, no global object could depend on being able to call any other, so the responsibility of initializing globals often fell on client code.
2. There was no way to override the meaning of `new` in C++, so an alternative syntax was necessary if all clients were to receive the same object. It was, though, at least possible to make it a compile-time error for client code to call `new` by marking the class constructor as either `protected` or `private`.
3. So the Gang of Four pivoted to a class method that would return the class's singleton object. Unlike a global function, a class method avoided adding yet another name to the global namespace, and unlike a static method, it could support subclasses that were singletons as well.

How could Python code illustrate their approach? Python lacks the complications of `new`, `protected`, and `private`. An alternative is to raise an exception in `__init__()` to make normal object instantiation impossible. The class method can then use a dunder method trick to create the object without triggering the exception:

```
# What the Gang of Four's original Singleton Pattern
# might look like in Python.

class Logger(object):
    _instance = None

    def __init__(self):
        raise RuntimeError('Call instance() instead')

    @classmethod
    def instance(cls):
        if cls._instance is None:
            print('Creating new instance')
            cls._instance = cls.__new__(cls)
            # Put any initialization here.
        return cls._instance
```

This successfully prevents clients from creating new instances by calling the class:

```
log = Logger()
```

```
Traceback (most recent call last):
  ...
RuntimeError: Call instance() instead
```

Instead, callers are instructed to use the `instance()` class method, which creates and returns an object:

```
log1 = Logger.instance()
print(log1)
```

```
Creating new instance
<Logger object at 0x7f0ff5e7c080>
```

Subsequent calls to `instance()` return the singleton without repeating the initialization step (as we can see from the fact that "Creating new instance" isn't printed again), exactly as the Gang of Four intended:

```
log2 = Logger.instance()
print(log2)
print('Are they the same object?', log1 is log2)
```

```
<Logger object at 0x7f0ff5e7c080>
Are they the same object? True
```

There are more complicated schemes that I can imagine for implementing the original Gang of Four class method in Python, but I think the above example does the best job of illustrating the original scheme with the least magic possible. Since the Gang of Four's pattern is not a good fit for Python anyway, I'll resist the temptation to iterate on it further, and instead move on to how the pattern is best supported in Python.

A more Pythonic implementation ¶

In one sense, Python started out better prepared than C++ for the Singleton Pattern, because Python lacks a `new` keyword that forces a new object to be created. Instead, objects are created by invoking a callable, which imposes no syntactic limitation on what operation the callable really performs:

```
log = Logger()
```

To let authors take control of calls to a class, Python 2.4 added the `__new__()` dunder method to support alternative creational patterns like the Singleton Pattern and The Flyweight Pattern.

The Web is replete with Singleton Pattern recipes featuring `__new__()` that each propose a more or less complicated mechanism for working around the method's biggest quirk: the fact that `__init__()` always gets called on the return value, whether the object that's being returned is new or not. To make my own example simple, I will

simply not define an `__init__()` method and thus avoid having to work around it:

```
# Straightforward implementation of the Singleton Pattern

class Logger(object):
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print('Creating the object')
            cls._instance = super(Logger, cls).__new__(cls)
            # Put any initialization here.
        return cls._instance
```

The object is created on the first call to the class:

```
log1 = Logger()
print(log1)
```

```
Creating the object
<Logger object at 0x7fa8e9cf7f60>
```

But the second call returns the same instance. The message "Creating the object" does not print, nor is a different object returned:

```
log2 = Logger()
print(log2)
print('Are they the same object?', log1 is log2)
```

```
<Logger object at 0x7fa8e9cf7f60>
Are they the same object? True
```

The example above opts for simplicity, at the expense of doing the `cls._instance` attribute lookup twice in the common case. For programmers who cringe at such waste, the result can of course be assigned a name and re-used in the return statement. And various other improvements can be imagined that would result in faster bytecode. But however elaborately tweaked, the above pattern is the basis of every Python class that hides a singleton object behind what reads like normal class instantiation.

Verdict ¶

While the Gang of Four's original Singleton Pattern is a poor fit for a language like Python that lacks the concepts of `new`, `private`, and `protected`, it's not as easy to dismiss the pattern when it's built atop `__new__()` — after all, singletons were part of the reason the `__new__()` dunder method was introduced!

But the Singleton Pattern in Python does suffer from several drawbacks.

A first objection is that the Singleton Pattern's implementation is difficult for many Python programmers to read. The alternative Global Object Pattern is easy to read: it's simply the familiar assignment statement, placed up at a module's top level. But a

Python programmer reading a `__new__()` method for the first time is probably going to have to stop and look for documentation to understand what's going on.

A second objection is that the Singleton Pattern makes calls to the class, like `Logger()`, misleading for readers. Unless the designer has put "Singleton" or some other hint in the class name, and the reader knows design patterns well enough to understand the hint, the code will read as though a new instance is being created and returned.

A third objection is that the Singleton Pattern forces a design commitment that The Global Object Pattern does not. Offering a global object still leaves a programmer free to create other instances of the class — which can be particularly helpful for tests, letting them each test a completely separate object without needing to reset a shared object back to a known good state. But the Singleton Pattern makes additional instances impossible. (Unless the caller is willing to stoop to monkey patching; or temporarily modifying `_instance` to subvert the logic in `__new__()`; or creating a subclass that replaces the method. But a pattern you have to work around is generally a pattern you should avoid.)

Why, then, would you use the Singleton Pattern in Python?

The one situation that would really demand the pattern would be an existing class that, because of a new requirement, will now operate best as a single instance. If it's not possible to migrate all client code to stop calling the class directly and start using a global object, then the Singeton Pattern would be a natural approach to pivoting to a singleton while preserving the old syntax.

But, otherwise, the pattern is best avoided in favor of following the advice of the official Python FAQ and using the The Global Object Pattern.
