• Home Page •

The Prebound Method Pattern ¶

A pattern native to the Python language.

Verdict

> A powerful technique for offering callables at the top level of your module that share state through a common object.

There are occasions on which a Python module wants to offer several routines in the module's global namespace that will need to share state with each other at runtime.

Probably the most famous example is the Python Standard Library's `random` module. While it does provide advanced users with the option to build their own `Random` instance, most programmers opt for the convenient slate of routines available in the module's global namespace — like randrange(), randint(), and choice() — that mirror the methods of a `Random` object.

How do these top-level routines share state?

Behind the scenes these callables are, in fact, methods that have all been bound ahead of time to a single instance of `Random` that the module itself has gone ahead and constructed.

After reviewing the problem that the Prebound Methods pattern solves, we will see how the pattern looks in Python code.

Alternatives ¶

The most primitive approach to sharing state between a pair of module-level callables is to write a pair of functions that manipulate global data that's stored next to them at the top level of the module.

Imagine that we want to offer a simple random number generator. It returns, in an endless loop, the numbers 1 through 255 in a fixed pseudo-random order. We also want to offer a simple `set_seed()` routine that resets the state of the generator to a known value — which is important both for tests that use random numbers, and for simulations driven by pseudo-randomness that want to offer reproducible results.

If Python only supported plain functions, we might store the shared seed as a module global that our functions would directly access and modify:

```
from datetime import datetime
```

```
_seed = datetime.now().microsecond % 255 + 1

def set_seed(value):
    global _seed
    _seed = value

def random():
    global _seed
    _seed, carry = divmod(_seed, 2)
    if carry:
        _seed ^= 0xb8
    return _seed
```

There are several problems with this approach.

First, it is impossible to ever instantiate a second copy of this random number generator. If two threads each wanted their own generator to avoid needing to protect it with locks, then they would be out of luck.

(Okay, not really; this is Python. Think of the possibilities! You could import the module, save a reference to it, remove it from the `sys.modules` dictionary, and then import it again to get a second copy. Or you could manually instantiate a second module object and copy all three names across. By "out of luck" I refer only to sane Pythonic approaches.)

Second, it is more difficult to decouple your random number generator tests from each other if the generator's state is global. Instead of each test getting to create and exercise a separate isolated instance, the tests will all have to share the single generator and correctly reset its state before the next test runs.

Third, this approach abandons encapsulation. This will sound like a fuzzier complaint than the previous two, but it can detract from readability ("readability counts") for a small tight system of two functions and a `seed` to be splayed out as three separate and not obviously related names in a module which might contain dozens of other objects.

To solve the above problems, we indent the two functions and wrap them up as methods of a new Python class. The two methods and their state can then be happily bundled together:

```
from datetime import datetime

class Random8(object):
    def __init__(self):
        self.set_seed(datetime.now().microsecond % 255 + 1)

    def set_seed(self, value):
        self.seed = value

    def random(self):
        self.seed, carry = divmod(self.seed, 2)
        if carry:
            self.seed ^= 0xb8
        return self.seed
```

This imposes an extra step on the caller: the object will have to be instantiated before the two methods can be called. Is there a way to let the caller skip that step?

The software engineer would do well to stop for a moment and to think quite seriously about the fact that in the vast majority of cases simply defining a class like this is enough. It will take only a single statement for your user to create an instance of your class. If their own application's architecture is too deeply compromised for them to easily pass the instance everywhere it's needed, they can always store the instance as a module global in one of their own modules for the use of the rest of their code.

There are several reasons that that the Prebound Methods pattern is a particularly good fit for a random number generator:

1. Instantiating a random number generator requires a system call — in our case, asking for the date; for the Python `random` module, fetching bytes from the system entropy pool. If every module needing a random number had to instantiate its own `Random` object, then this cost would be incurred repeatedly.
2. Pseudo-random number generators are an interesting case of a resource whose behavior can be even more desirable when shared. If you are the lone caller to an instance, you see its completely predictable repeating sequence of values. If instead you are sharing that instance with other code, the generator will appear to skip ahead in its sequence unpredictably each time other callers have themselves called the generator.
3. Since most users of `random`, including several modules within the Standard Library, import it specifically to use its module-level calls, it is rare for the pre-built `Random` instance that powers them to languish unused.

If the costs and benefits strike a similar balance for a module of your own you are designing, then the Prebound Method pattern can let you deliver remarkable convenience.

The pattern ¶

To offer your users a slate of Prebound Methods:

- Instantiate your class at the top level of your module.
- Consider assigning it a private name prefixed with an underscore `_` that does not invite users to meddle with the object directly.
- Finally, assign a bound copy of each of the object's methods to the module's global namespace.

For the random number generator that we used as an illustration above, the entire module might look like this:

```
from datetime import datetime

class Random8(object):
```

```
    def __init__(self):
        self.set_seed(datetime.now().microsecond % 255 + 1)

    def set_seed(self, value):
        self.seed = value

    def random(self):
        self.seed, carry = divmod(self.seed, 2)
        if carry:
            self.seed ^= 0xb8
        return self.seed

_instance = Random8()

random = _instance.random
set_seed = _instance.set_seed
```

Users will now be able to invoke each method as though it were a stand-alone function. But the methods will secretly share state thanks to the common instance that they are bound to, without requiring the user to manage or pass that state explicitly.

When exercising this pattern, please be responsible about the dangers of instantiating an object at import time. This pattern is usually not appropriate for a class whose constructor creates files, reads a database configuration, opens sockets, or that in general will inflict side effects on the program importing them. In that case, you will do better to either avoid the Prebound Methods pattern entirely, or else to defer any actual side effects until one of the methods is called. You could choose a middle ground of providing a `setup()` method and requiring application programmers to invoke it before they can expect any of the other routines to work.

But for lightweight objects that can be instantiated without substantial delay or complication, the Prebound Methods pattern is an elegant way to make the stateful behavior of a class instance available up at a module's global level.

Examples of the pattern are strewn merrily across the Standard Library even beyond the `random` module we used as our example. The `calendar.py` module uses it:

```
c = TextCalendar()

...

monthcalendar = c.monthdayscalendar
prweek = c.prweek
week = c.formatweek
weekheader = c.formatweekheader
prmonth = c.prmonth
month = c.formatmonth
calendar = c.formatyear
prcal = c.pryear
```

As does the venerable old `reprlib.py`, to offer a single `repr()` routine:

```
aRepr = Repr()
repr = aRepr.repr
```

As do several other modules — you will find the Prebound Methods pattern used with each of these Standard Library objects:

- `distutils.log._global_log`
- `multiprocessing.forkserver._forkserver`
- `multiprocessing.semaphore_tracker._semaphore_tracker`
- `secrets._sysrand`

One final hint: it is almost always better to assign methods to global names explicitly. Even if there are a dozen methods, I recommend going ahead and writing a quick stack of a dozen assignment statements aligned at the left-hand column of your module that make visible the whole list of global names you are defining. The fact that Python is a dynamic language might tempt you to automate the series of assignments instead, using attribute introspection and a `for` loop. I advise against it. Python programmers believe that "Explicit is better than implicit" — and materializing the stack of names as real code will better support human readers, language servers, and even venerable old `grep`.
