• Home Page •

The Prototype Pattern ¶

A "Creational Pattern" from the Gang of Four book

Verdict

> The Prototype pattern isn't necessary in a language powerful enough to support first-class functions and classes.

It is almost embarrassing how many good alternatives Python offers to the Prototype pattern.

As the problem and its solutions are simple, this article will be brief. We will define the problem, list several Pythonic solutions, and finish by studying how the Gang of Four solved the same problem under more onerous constraints.

The problem ¶

The Prototype Pattern suggests a mechanism by which a caller can provide a framework with a menu of classes to instantiate when the user — or some other runtime source of dynamic requests — selects the classes from a menu of choices.

The problem would be a much simpler one if none of the classes in the menu needed `__init__()` arguments:

```
class Sharp(object):
    "The symbol ♯."

class Flat(object):
    "The symbol ♭."
```

Instead, the Prototype pattern is called into play when classes will need to be instantiated that require arguments:

```
class Note(object):
    "Musical note 1 ÷ `fraction` measures long."
    def __init__(self, fraction):
        self.fraction = fraction
```

It is this situation — supplying a framework with a menu of classes that will need to be instantiated with pre-specified argument lists — that the Prototype pattern is designed to address.

Pythonic solutions ¶

Python offers several possible mechanisms for supplying a framework with the classes we want to instantiate and the arguments we want them instantiated with.

In the examples below, we will provide the menu of classes in a simple data structure — a Python dictionary — but the same principles apply even if the framework wants the menu items supplied in some other data structure, or supplied separately in a series of `register()` calls.

One approach is to design the classes so that they only need positional arguments, not keyword arguments. The framework will then be able to store the arguments as a simple tuple, which can be supplied separately from the class itself — the familiar approach of the Standard Library Thread class which asks for a callable `target=` separately from the `args=(...)` it will be passed. Our menu items might look like:

```
menu = {
    'whole note': (Note, (1,)),
    'half note': (Note, (2,)),
    'quarter note': (Note, (4,)),
    'sharp': (Sharp, ()),
    'flat': (Flat, ()),
}
```

Alternatively, each class and arguments could live in the same tuple:

```
menu = {
    'whole note': (Note, 1),
    'half note': (Note, 2),
    'quarter note': (Note, 4),
    'sharp': (Sharp,),
    'flat': (Flat,),
}
```

The framework would then invoke each callable using some variation of `tup[0](*tup[1:])`.

In the more general case, though, a class might require not just positional arguments but also keyword arguments. In response, we could pivot to providing simple callables to the framework, using lambda expressions for the classes that require arguments:

```
menu = {
    'whole note': lambda: Note(fraction=1),
    'half note': lambda: Note(fraction=2),
    'quarter note': lambda: Note(fraction=4),
    'sharp': Sharp,
    'flat': Flat,
}
```

While lambdas don't support quick introspection — it isn't easy for a framework or debugger to inspect them to learn what callable they will invoke or what arguments they will supply — they work fine if all the framework needs to do is invoke them.

Another approach is to use a partial for each item, which packages together a callable

with both positional and keywords arguments that will be supplied when the partial itself is later called:

```
from functools import partial

# Keyword arguments for illustration only;
# in this case could instead write 'partial(Note, 1)'

menu = {
    'whole note': partial(Note, fraction=1),
    'half note': partial(Note, fraction=2),
    'quarter note': partial(Note, fraction=4),
    'sharp': Sharp,
    'flat': Flat,
}
```

I will stop there, though you are free to keep imagining more alternatives — for example, you could supply a class, a tuple, and a dictionary `(cls, args, kw)` for each menu item and the framework would call `cls(*args, **kw)` when each menu item is selected. The choices in Python for tackling this problem are numerous because classes and functions in Python are first-class and are therefore eligible to be passed as arguments and stored in data structures just like any other objects.

Implementing ¶

But the Gang of Four did not have the luxury of such easy circumstances as Python programmers enjoy. Armed with only polymorphism and the method call, they sallied forth to create a workable pattern.

You might at first imagine that, in the absence of tuples and the ability to apply them as argument lists, we are going to need factory classes which will each remember a particular list of arguments and then supply those arguments when they are asked for a new object:

```
# What the Prototype pattern avoids:
# needing one factory for every class.

class NoteFactory(object):
    def __init__(self, fraction):
        self.fraction = fraction

    def build(self):
        return Note(self.fraction)

class SharpFactory(object):
    def build(self):
        return Sharp()

class FlatFactory(object):
    def build(self):
        return Flat()
```

Fortunately, the situation is not so grim. If you re-read the factory classes above, you will notice that they each look similar — eerily similar — remarkably similar! — to the

target classes we want to create. The `NoteFactory`, exactly like the `Note` itself, stores an attribute `fraction`. The stack of factories winds up looking, at least in their attribute lists, like the stack of classes we want to instantiate.

This symmetry suggest a way to solve our problem without having to mirror each class with a factory. What if we used the original objects themselves to store the arguments, and gave them the ability to provide new instances?

The result is the Prototype pattern! All of the factory classes disappear. Instead, each instance gains a `clone()` method to which it responds by building a new instance with exactly the same arguments it received:

```
# The Prototype pattern: teach each object
# instance how to build copies of itself.

class Note(object):
    "Musical note 1 ÷ `fraction` measures long."
    def __init__(self, fraction):
        self.fraction = fraction

    def clone(self):
        return Note(self.fraction)

class Sharp(object):
    "The symbol ♯."
    def clone(self):
        return Sharp()

class Flat(object):
    "The symbol ♭."
    def clone(self):
        return Flat()
```

While we could make this example more complicated — for example, each `clone()` method should probably call `type(self)` instead of hard-coding its class name, in case the method gets called on a subclass — this at least illustrates the pattern. The Prototype Pattern is not as convenient as the mechanisms available in the Python language, but this clever simplification made it much easier for the Gang of Four to accomplish parametrized object creation in some of the underpowered object oriented languages that were popular last century.
