• Home Page •

The Factory Method Pattern ¶

A "Creational Pattern" from the Gang of Four book

Verdict

> The "Factory Method" pattern is a poor fit for Python. It was designed for underpowered programming languages where classes and functions can't be passed as parameters or stored as attributes. In those languages, the Factory Method serves as an awkward but necessary escape route. But it's not a good design for Python applications.

You will often create objects in Python that themselves need to turn around and create more objects. Imagine an HTTP connection pool, for example, that sometimes needs to create new connections to replace old connections as they time out and close.

Now comes the challenge. What if your program will run behind a special corporate proxy, and the HTTP connection pool will only be able to communicate if it creates and uses specially configured network connections instead of the generic sockets that it usually creates?

The Factory Method is one pattern by which the HTTP connection pool class could offer you a way to choose what kind of connection it creates. Before learning how this awkward pattern works, let's review the Pythonic design patterns that solve the same problem.

Dodge: use Dependency Injection ¶

First, we should stop for a moment and double-check something. Does the class you're designing *really* need to go around creating other classes?

If you know up front all the objects that the class will need, you should consider providing them to the class yourself through Dependency Injection.

You will see excellent examples of this simpler pattern everywhere that Python libraries let you pass an already-open file object instead of making you supply a path and insisting on opening the file themselves. Take the Standard Library's JSON parser as an example. Instead of asking for a path, its `load()` method lets you open the file:

```
import json
with open('input_data.json') as f:
    data = json.load(f)
```

By leaving you in charge of instantiating the file object, this maneuver accomplishes many goals at once.

- Decoupling: the library doesn't need to know all the parameters accepted by the *open()* method, and doesn't need to accept every one of them as a parameter itself. If the file object were to grow more creation parameters in the future, `json.load()` won't need to change.
- Efficiency: If you already have the file open for other reasons, you can go ahead and provide the open file object. The library won't insist on re-opening the file. This is crucial if the file is a read-once source of data like an anonymous pipe.
- Flexibility: You can pass any file-like object you want. It can be a subclass of the standard file object, or be completely independent. You could pass a `StringIO` that operates directly on data in RAM instead of needing the JSON written to disk first. Or you could offer a wrapper around a socket that lets you parse JSON data as it arrives off the network without needing to be stored locally first.

In a simple case like this, the Dependency Injection pattern completely eliminates the need for an object like a JSON parser to have its design made more complicated by the details of how other objects get created.

So (a) if your class always needs a particular object built, and (b) if users will normally want to at least customize the parameters with which it is instantiated, you should strongly consider Dependency Injection. In that case, you might want to read Miško Hevery's 2008 post about dependency injection for an important note about the dangers of applying Dependency Injection halfway!

Instead: use a Class Attribute Factory ¶

The next alternative comes to us from the 1990s. The class that needs to be created can be attached as an attribute on the class that will be doing the creating.

You can see this pattern in some of the older modules in the Python Standard Library. For example, when an `HTTPConnection` is done sending a request, it expects to receive and parse a response. But what class should it use to parse and represent the response? What if you happen to be talking to a server that sends non-standard information in its response for which you require special handling?

What I will here call the Class Attribute Factory pattern takes advantage of the fact that classes are first-class objects in Python. It simply attaches the class as an attribute of the class that will be doing the creating. You'll find the following code in the Standard Library:

```
class HTTPConnection:
    ...
    response_class = HTTPResponse
    ...
    def getresponse(self):
        ...
```

```
        response = self.response_class(self.sock, method=self._method)
        ...
```

This gives the code using the `HTTPConnection` complete control over what happens when it is time to build a response. All it has to do is create a subclass and use the subclass instead:

```
class SpecialHTTPConnection(HTTPConnection):
    response_class = SpecialHTTPResponse
```

That's all there is to it! While subclassing generally means that a software design is running at least somewhat against the grain of the Python language, the approach will still work well. The subclass will behave exactly like a normal Standard Library HTTP connection except when the moment comes to instantiate a response. At that point the connection will access its `response_class` attribute, receive the alternative class you provided instead of the normal one, and the alternative response class will then be in control.

There is more flexibility here we should explore, but first let's look at the more modern alternative to the Class Attribute Factory.

Instead: use an Instance Attribute Factory ¶

Why should you have to subclass an object merely to customize its behavior?

It's a serious question — very serious, for at one point in the history of programming some adherents of a doctrine called "object orientation" were suggesting that if you wanted a button that said "Submit" that you shouldn't be able to simply instantiate a button with a parameter like `label="Submit"` but that you should instead have to subclass the button and override its default `label()` method to return something new.

Happily, an alternative to subclass-powered customization has swept the Python community: what I'll call the Instance Attribute Factory. Among the many good examples of its practice is the `json` module in the Standard Library, which was added around 2008.

Here's one example from the `json` module. Every time the JSON module encounters a number in its input, it has to instantiate some kind of Python object capable of representing the number. But which number class should it instantiate? An integer, if the fractional part of the number is zero? A float, the only numeric type in JavaScript? Or a `Decimal` that's guaranteed to not lose any precision?

Watch how elegantly the `json` module handles the question:

```
class JSONDecoder(object):
    ...
    def __init__(self, ... parse_float=None, ...):
        ...
        self.parse_float = parse_float or float
        ...
```

Whenever it encounters a number in its input, it simply calls `self.parse_float()` with the string as input.

This is Python code that's running on all pistons. If the developer does not intervene, each number is interpreted using a lighting-fast call to the `float` type itself. If instead the developer has provided their own callable for parsing numbers, then that callable is transparently used instead.

The beauty is that it all happens without a single additional class! Instead of forcing the programmer to create a new class each time they want to customize behavior, individual `JSONDecoder` instances can each be configured directly. You can create a custom decoder with a single line of code:

```
from decimal import Decimal
from json import JSONDecoder

my_decoder = JSONDecoder(parse_float=Decimal)
```

Besides the benefits of clarity and brevity, an advantage of customizing an object through its parameters is that parameters compose so beautifully in Python. If several pieces of code have parameters for the decoder that they need to combine, the task is no more difficult than building an empty `dict` and then using `update()` to fill it with each set of parameters, setting the parameters last that should be allowed to override the earlier ones.

Instance attributes override class attributes ¶

I should admit that the previous two design patterns are not as completely different as I have tried to make it sound. At bottom, both classes — the `HTTPConnection` and the `JSONDecoder` — make the exact same move when they are ready to create a new object: they start with `self` and use `.` to access some specific attribute on it. The only difference in the two design patterns above is in how they choose to supply the attribute. The first pattern happens to use a class attribute, while the second uses an instance attribute.

But the two are not mutually exclusive. There is no rule that if you have a class attribute named `.response_class` that you can't also have an instance attribute named `.response_class` — and the rule in the case where you have both is simple: the instance attribute wins.

Which means I should admit that, really, even though I made a big deal about claiming that the `HTTPConnection` forces you to subclass it, it's not true. You can override the

default but just setting an instance attribute instead, just like the `JSONDecoder` does! The only difference is that the `HTTPConnection` won't give you any help — you'll have to reach in and set the instance attribute yourself:

```
conn = HTTPConnection()
conn.response_class = SpecialHTTPResponse
```

So even when an old-fashioned class looks like it wants you to create a subclass that specifies a new value for one of its class attributes, you can often use the more modern Instance Attribute Factory instead!

There are tiny differences in semantics and performance between class attributes and instance attributes, but I'll refer you to the Python documentation and to Stack Overflow if you think your code is wandering towards an edge case where you care about the difference.

In general you should choose between the above patterns based on readability. If you can imagine developers ever wanting to customize object creation, then go ahead and try making the object creation routine (the "factory") a parameter in your `__init__()` method and store it as an instance attribute. If instead you think that customization will be extremely rare, then make it a class attribute, remembering that the developer can always reach in and override it in those rare cases where they need to.

Any callables accepted ¶

In the examples above, we used actual classes like `Decimal` and the fictional `SpecialHTTPResponse` when setting an attribute like `response_class` or `parse_float`. But you'll note that the only thing the callers cared about was that these classes were callables. There's happily no `new` keyword in Python, so object instantiation looks exactly like a normal function or method call.

This means that you can substitute a function for any of these callbacks, and it will work just as well! For example, you could provide the JSON decoder with a function like this as its `parse_float` parameter:

```
def parse_number(string):
    if '.' in string:
        return Decimal(string)
    return int(string)
```

You can provide not only a function, but any other kind of callable as well — maybe a bound method, or a class method like an alternative constructor. You could even provide a callable that you've spun up dynamically using functional programming techniques like partial application:

```
from decimal import Context, ROUND_DOWN
from functools import partial

parse_number = partial(Decimal, context=Context(2, ROUND_DOWN))
```

Feel free to enjoy this Pythonic freedom of providing any kind of callable, and not limiting yourself to just providing classes, whether you are using the Class Attribute Factory or an Instance Attribute Factory.

Implementing ¶

Having described the happy alternatives, I should finish by showing you the Factory Method itself. Imagine that you were using a language where:

- Classes are not first-class objects. You are not allowed to leave a class sitting around as an attribute of either a class instance or of another class itself.
- Functions are not first-class objects. You're not allowed to save a function as an instance of a class or class instance.
- No other kind of callable exists that can be dynamically specified and attached to an object at runtime.

Under such dire constraints, you would turn to subclassing as a natural way to attach verbs — new actions — to existing classes, and you might use method overriding as a basic means of customizing behavior. And if on one of your classes you designed a special method whose only purpose was to isolate the act of creating new objects, then you'd be using the Factory Method pattern.

The Factory Method pattern can often be observed anywhere that code from an underpowered but object-oriented language has been translated straight into Python. The `logging` module from the Standard Library comes immediately to mind. Here's an excerpt:

```
class Handler(Filterer):
    ...
    def __init__(self, level=NOTSET):
        ...
        self.createLock()
    ...
    def createLock(self):
        """
        Acquire a thread lock for serializing access to the underlying
I/O.
        """
        self.lock = threading.RLock()
    ...
```

What if you wanted to create a `Handler` that uses a special kind of lock? The intention here is that you subclass `Hander` and override `createLock()` to return your own favorite kind of lock instead. It's a clunky approach, it takes several lines of code, and it won't compose well if there are several ways you want to customize `Handler` objects in a variety of situations — you'll wind up with classes all over the place.

But it will work.

It just won't be very Pythonic.
