• Home Page •

# The Abstract Factory Pattern ¶

*A "Creational Pattern" from the* Gang of Four book

Verdict

> The Abstract Factory is an awkward workaround for the lack of first-class functions and classes in less powerful programming languages. It is a poor fit for Python, where we can instead simply pass a class or a factory function when a library needs to create objects on our behalf.

The Python Standard Library's `json` module is a good example of a library that needs to instantiate objects on behalf of its caller. Consider a JSON string like this one:

```
text = '{"total": 9.61, "items": ["Americano", "Omelet"]}'
```

By default, the `json` module's `loads()` function will create `unicode` objects for the strings `"Americano"` and `"Omelet"`, a `list` to hold them, a Python `float` for `9.61`, and a `dict` with `unicode` keys for the top-level JSON object.

But some users will not be content with these defaults. For example, an accountant would probably be unhappy with the choice of a `float` to represent an exact amount like "9 dollars 61 cents" and would prefer an exact `Decimal` instead.

The need to control what kind of numeric object the `json` module creates is a specific instance of a general problem:

- In the course of performing its duties, a routine is going to need to create a number of objects on behalf of the caller.
- But the class that the routine would instantiate by default does not cover all possible cases.
- So instead of hard-coding that default class and making customization impossible, the routine wants to let the caller specify which classes it will instantiate.

First, we'll look at the Pythonic approach to this problem. Then we'll start placing a series of restrictions on our Python code to more closely model legacy object oriented languages, until the Abstract Factory pattern emerges as the best solution within those limitations.

## The Pythonic approach: callable factories ¶

In Python, a "callable" — any object `f` that executes code when invoked using the syntax `f(a, b, c)` — is a first-class object. To be "first class" means that a callable can be passed as a parameter, returned as a return value, and can even be stored in a data structure like a list or dictionary.

First-class callables offer a powerful mechanism for implementing object "factories", a fancy term for "routines that build and return new objects."

A beginning Python programmer might expect that each time they need to supply a factory, they will be responsible for writing a function:

```
import json
from decimal import Decimal

def build_decimal(string):
    return Decimal(string)

print(json.loads(text, parse_float=build_decimal))
```

```
{'total': Decimal('9.61'), 'items': ['Americano', 'Omelet']}
```

This simple factory ran successfully! The number returned is a decimal instead of a float.

(Note my choice of a verb `build_decimal()` as the name of this function, instead of a noun like `decimal_factory()` — I find a function name easier to read when it tells me what the function *does* instead of telling me what *kind* of function it is.)

While the above function will certainly work, there is an elision we can perform thanks to the fact that Python types are themselves callables. Because `Decimal` is a callable taking a string argument and returning a decimal object instance, we can dispense with our own factory and pass the `Decimal` type directly to the JSON loader! Unless we need to edit the string first, like by removing a leading currency symbol, `Decimal` can completely replace our little factory:

```
print(json.loads(text, parse_float=Decimal))
```

```
{'total': Decimal('9.61'), 'items': ['Americano', 'Omelet']}
```

There is one implementation detail that deserves mention. If you study the `json` module you will discover that `load()` is simply a wrapper around the `JSONDecoder` class. How does the decoder instance itself support an alternative factory? Its initialization method stores the `parse_float` argument as an instance attribute, defaulting to Python's built-in `float` type if no override was specified:

```
self.parse_float = parse_float or float
```

It can then invoke it later as `self.parse_float(…)`.

If you are interested in variations on this pattern — where a class uses its instance attributes to remember how it's supposed to create a specific kind of object — then try reading about the Factory Method which explores several variations on this maneuver.

But to arrive at the Abstract Factory pattern, we need to head in a different direction. Here we'll pursue what happens to an object factory itself — whether `Decimal()` or our hand-written `build_decimal()` — if we begin restricting the set of Python features we let ourselves use.

## Restriction: outlaw passing callables ¶

What if Python didn't let you pass callables as parameters?

That restriction would remove an entire dimension from Python's flexibility. Instead of supporting both "nouns" and "verbs" as arguments — both class instances and callable functions — some legacy languages only support passing class instances. Under that restriction, every simple factory would need to pivot from a function to a method:

```
# In Python: a factory function.

def build_decimal(string):
    return Decimal(string.lstrip('$'))

# In some legacy languages: the code must
# move inside a class method instead.

class DecimalFactory(object):
    @staticmethod
    def build(string):
        return Decimal(string.lstrip('$'))
```

In traditional Object Oriented programming, the word "factory" is the name of this kind of class — a class that offers a method that builds an object. In naming the equivalent Python function `build_decimal()`, therefore, I'm not only indulging in my own preference for giving functions verb-names rather than noun-names, but being as precise as possible in naming: the "factory" is not the callable, but the class that holds it.

Instead of continuing our earlier example of JSON parsing, let's switch to a simpler task that can fit in a couple of lines of code: parsing a comma-separated list of numbers. Here's how the parser would invoke the builder method on our factory class.

```
class Loader(object):
    @staticmethod
    def load(string, factory):
        string = string.rstrip(',')  # allow trailing comma
        return [factory.build(item) for item in string.split(',')]

result = Loader.load('464.80, 993.68', DecimalFactory)
print(result)
```

```
[Decimal('464.80'), Decimal('993.68')]
```

Note that, thanks to the fact that Python classes offer static and class methods that can be invoked without an instance, we have not yet been reduced to needing to instantiate the factory class — we are simply passing the Python class in as a first-class object.

## Restriction: outlaw passing classes ¶

Next, let's also pretend that a Python class cannot be passed as a value, but that only object instances can be assigned to names and passed as parameters.

This restriction is going to prevent us from passing the `DecimalFactory` class as an argument to the `load()` method. Instead, we're going to have to uselessly instantiate `DecimalFactory` and pass the resulting object:

```
f = DecimalFactory()

result = Loader.load('464.80, 993.68', f)
print(result)
```

```
[Decimal('464.80'), Decimal('993.68')]
```

Note the difference between this pattern and the Factory Method. Here, we are neither asked nor required to subclass `Loader` itself in order to customize the objects it creates. Instead, object creation is entirely parametrized by the separate factory object we choose to pass in.

Note also the clear warning sign in the factory's own code that `build()` should, in Python, not really be the method of an object. Scroll back up and read the method's code. Where does it accept as an argument, or use in its result, the object `self` on which it is being invoked? It makes no use of it at all! The method never mentions `self` in its code. As Jack Diederich propounded in his famous talk Stop Writing Classes, a method that never uses `self` should not actually be a method in Python. But such are the depths to which we've been driven by these artificial restrictions.

## Generalizing: the complete Abstract Factory ¶

Two final moves will illustrate the full design pattern.

First, let's expand our factory to create every kind of object that the loader needs to create — in this case, not just the numbers that are being parsed, but even the container that will hold them. Now that we have switched to instantiating the factory, we can write these as plain methods instead of static methods:

```
class Factory(object):
    def build_sequence(self):
        return []

    def build_number(self, string):
        return Decimal(string)
```

And here is an updated loader that uses this factory:

```
class Loader(object):
    def load(string, factory):
        sequence = factory.build_sequence()
        for substring in string.split(','):
            item = factory.build_number(substring)
            sequence.append(item)
        return sequence

f = Factory()
result = Loader.load('1.23, 4.56', f)
print(result)
```

```
[Decimal('1.23'), Decimal('4.56')]
```

Every choice it needs to make about object instantiation is deferred to the factory instead of taking place in the parser itself.

Second, consider the behavior of languages that force you to declare ahead of time the type of each method parameter. You would overly restrict your future choices if your code insisted that the `factory` parameter could only ever be an instance of this particular class `Factory` because then you could never pass in anything that didn't inherit from it.

Instead, to more happily separate specification from implementation, you would create an abstract class. It's this final step that merits the word "abstract" in the pattern's name "Abstract Factory". Your abstract class would merely promise that the `factory` argument to `load()` would be a class adhering to the required interface:

```
from abc import ABCMeta, abstractmethod

class AbstractFactory(metaclass=ABCMeta):

    @abstractmethod
    def build_sequence(self):
        pass

    @abstractmethod
    def build_number(self, string):
        pass
```

Once the abstract class is in place and `Factory` inherits from it, though, the operations that take place at runtime are exactly the same as they were before. The factory's methods are called with various arguments, which direct them to create various kinds of object, which they construct and return without the caller needing to know the details.

It's like something you might do in Python, but made overly complicated. So avoid the Abstract Factory and use callables as factories instead.

---