```
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Arthur', 'King')
>>> object3.name, object3.job
('Brian', None)
```

К настоящему времени вы должны знать, что абсолютно все в Python является объектом "первого класса" — в том числе классы, которые обычно представляют собой лишь входные данные для компилятора в языках, подобных C++. Передавать их таким способом вполне естественно. Однако как упоминалось в начале текущей части книги, задачи ООП в Python решаются только с помощью объектов, *полученных* из классов.

Для чего используются фабрики?

Так в чем же польза от функции `factory` (помимо повода проиллюстрировать в книге объекты классов)? К сожалению, трудно показать приложения паттерна проектирования "Фабрика" без приведения кода большего объема, чем для этого есть место. Тем не менее, в целом такая фабрика способна сделать возможной изоляцию кода от деталей динамически конфигурируемого создания объектов.

Например, вспомните пример функции `processor`, абстрактно представленной в главе 26, и затем снова в качестве примера композиции ранее в этой главе. Она принимает объекты `reader` и `writer` для обработки произвольных потоков данных. Первоначальной версии функции `processor` вручную передавались экземпляры специализированных классов вроде `FileWriter` и `SocketReader` с целью настройки обрабатываемых потоков данных; позже мы передавали жестко закодированные объекты файла, потока и форматера. В более динамическом сценарии для настройки потоков данных могли бы применяться внешние механизмы, такие как конфигурационные файлы или графические пользовательские интерфейсы.

В динамическом мире подобного рода может отсутствовать возможность жесткого кодирования в сценариях процедуры для создания объектов интерфейса к потокам данных и взамен их придется создавать во время выполнения в соответствии с содержимым конфигурационного файла.

Такой файл может просто задавать строковое имя класса потока данных, подлежащего импортированию из модуля, плюс необязательный аргумент для вызова конструктора. Здесь могут пригодиться функции или код в стиле фабрик, потому что он позволяет извлекать и передавать классы, код которых не реализован заблаговременно в программе. На самом деле классы могут даже вообще не существовать в момент, когда пишется код:

```
classname = ...извлечь из конфигурационного файла и произвести разбор...
classarg = ...извлечь из конфигурационного файла и произвести разбор...
import streamtypes                          # Настраиваемый код
aclass = getattr(streamtypes, classname)    # Извлечь из модуля
reader = factory(aclass, classarg)          # Или aclass(classarg)
processor(reader, ...)
```

Здесь снова используется встроенная функция `getattr` для извлечения атрибута модуля по строковому имени (она похожа на выражение *объект.атрибут*, но *атрибут* является строкой). Поскольку в приведенном фрагменте кода предполагается наличие у конструктора единственного аргумента, строго говоря, он не нуждается в функции `factory` — мы могли бы создавать экземпляр с помощью `aclass(classarg)`. Однако фабричная функция может оказаться более полезной при наличии неизвестных списков аргументов, а общий паттерн проектирования "Фабрика" способен повысить гибкость кода.