CHAPTER 6

# Design Patterns with First-Class Functions

Conformity to patterns is not a measure of goodness.[1]

— Ralph Johnson
*Coauthor of the Design Patterns classic*

Although design patterns are language-independent, that does not mean every pattern applies to every language. In his 1996 presentation, "Design Patterns in Dynamic Languages", Peter Norvig states that 16 out of the 23 patterns in the original Design Patterns book by Gamma et al. become either "invisible or simpler" in a dynamic language (slide 9). He was talking about Lisp and Dylan, but many of the relevant dynamic features are also present in Python.

The authors of *Design Patterns* acknowledge in their Introduction that the implementation language determines which patterns are relevant:

> The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor.[2]

In particular, in the context of languages with first-class functions, Norvig suggests rethinking the Strategy, Command, Template Method, and Visitor patterns. The general idea is: you can replace instances of some participant class in these patterns with simple functions, reducing a lot of boilerplate code. In this chapter, we will refactor Strategy

1. From a slide in the talk "Root Cause Analysis of Some Faults in Design Patterns," presented by Ralph Johnson at IME/CCSL, Universidade de São Paulo, Nov. 15, 2014.
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995), p. 4.

using function objects, and discuss a similar approach to simplifying the Command pattern.

## Case Study: Refactoring Strategy

Strategy is a good example of a design pattern that can be simpler in Python if you leverage functions as first-class objects. In the following section, we describe and implement Strategy using the "classic" structure described in *Design Patterns*. If you are familiar with the classic pattern, you can skip to "Function-Oriented Strategy" on page 172 where we refactor the code using functions, significantly reducing the line count.

### Classic Strategy

The UML class diagram in Figure 6-1 depicts an arrangement of classes that exemplifies the Strategy pattern.

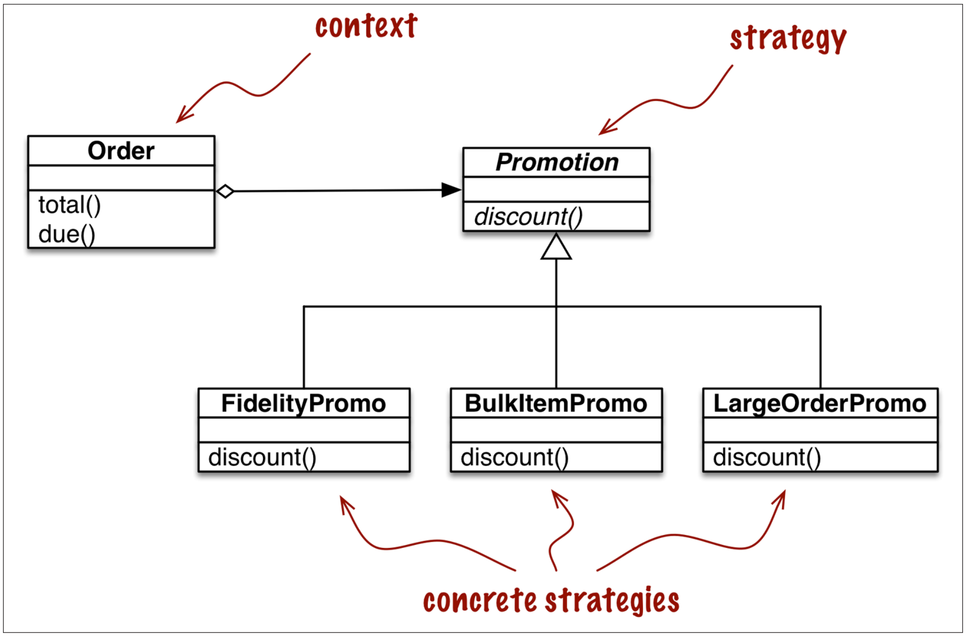

*Figure 6-1. UML class diagram for order discount processing implemented with the Strategy design pattern*

The Strategy pattern is summarized like this in *Design Patterns*:

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

A clear example of Strategy applied in the ecommerce domain is computing discounts to orders according to the attributes of the customer or inspection of the ordered items.

Consider an online store with these discount rules:

- Customers with 1,000 or more fidelity points get a global 5% discount per order.
- A 10% discount is applied to each line item with 20 or more units in the same order.
- Orders with at least 10 distinct items get a 7% global discount.

For brevity, let's assume that only one discount may be applied to an order.

The UML class diagram for the Strategy pattern is depicted in Figure 6-1. Its participants are:

*Context*
: Provides a service by delegating some computation to interchangeable components that implement alternative algorithms. In the ecommerce example, the context is an `Order`, which is configured to apply a promotional discount according to one of several algorithms.

*Strategy*
: The interface common to the components that implement the different algorithms. In our example, this role is played by an abstract class called `Promotion`.

*Concrete Strategy*
: One of the concrete subclasses of Strategy. `FidelityPromo`, `BulkPromo`, and `LargeOrderPromo` are the three concrete strategies implemented.

The code in Example 6-1 follows the blueprint in Figure 6-1. As described in *Design Patterns*, the concrete strategy is chosen by the client of the context class. In our example, before instantiating an order, the system would somehow select a promotional discount strategy and pass it to the `Order` constructor. The selection of the strategy is outside of the scope of the pattern.

*Example 6-1. Implementation Order class with pluggable discount strategies*

```python
from abc import ABC, abstractmethod
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price
    def total(self):
        return self.price * self.quantity


class Order:  # the Context

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())


class Promotion(ABC):  # the Strategy: an abstract base class

    @abstractmethod
    def discount(self, order):
        """Return discount as a positive dollar amount"""


class FidelityPromo(Promotion):  # first Concrete Strategy
    """5% discount for customers with 1000 or more fidelity points"""

    def discount(self, order):
        return order.total() * .05 if order.customer.fidelity >= 1000 else 0


class BulkItemPromo(Promotion):  # second Concrete Strategy
    """10% discount for each LineItem with 20 or more units"""

    def discount(self, order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * .1
        return discount
```

```
class LargeOrderPromo(Promotion):  # third Concrete Strategy
    """7% discount for orders with 10 or more distinct items"""

    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0
```

Note that in Example 6-1, I coded `Promotion` as an abstract base class (ABC), to be able to use the `@abstractmethod` decorator, thus making the pattern more explicit.

In Python 3.4, the simplest way to declare an ABC is to subclass `abc.ABC`, as I did in Example 6-1. From Python 3.0 to 3.3, you must use the `metaclass=` keyword in the `class` statement (e.g., `class Promotion(metaclass=ABCMeta):`).

Example 6-2 shows doctests used to demonstrate and verify the operation of a module implementing the rules described earlier.

*Example 6-2. Sample usage of Order class with different promotions applied*

```
    >>> joe = Customer('John Doe', 0)  ❶
    >>> ann = Customer('Ann Smith', 1100)
    >>> cart = [LineItem('banana', 4, .5),  ❷
    ...         LineItem('apple', 10, 1.5),
    ...         LineItem('watermellon', 5, 5.0)]
    >>> Order(joe, cart, FidelityPromo())  ❸
    <Order total: 42.00 due: 42.00>
    >>> Order(ann, cart, FidelityPromo())  ❹
    <Order total: 42.00 due: 39.90>
    >>> banana_cart = [LineItem('banana', 30, .5),  ❺
    ...                LineItem('apple', 10, 1.5)]
    >>> Order(joe, banana_cart, BulkItemPromo())  ❻
    <Order total: 30.00 due: 28.50>
    >>> long_order = [LineItem(str(item_code), 1, 1.0)  ❼
    ...               for item_code in range(10)]
    >>> Order(joe, long_order, LargeOrderPromo())  ❽
    <Order total: 10.00 due: 9.30>
    >>> Order(joe, cart, LargeOrderPromo())
    <Order total: 42.00 due: 42.00>
```

❶ Two customers: `joe` has 0 fidelity points, `ann` has 1,100.

❷ One shopping cart with three line items.

❸ The `FidelityPromo` promotion gives no discount to `joe`.

❹ ann gets a 5% discount because she has at least 1,000 points.

❺ The banana_cart has 30 units of the "banana" product and 10 apples.

❻ Thanks to the BulkItemPromo, joe gets a $1.50 discount on the bananas.

❼ long_order has 10 different items at $1.00 each.

❽ joe gets a 7% discount on the whole order because of LargerOrderPromo.

Example 6-1 works perfectly well, but the same functionality can be implemented with less code in Python by using functions as objects. The next section shows how.

## Function-Oriented Strategy

Each concrete strategy in Example 6-1 is a class with a single method, discount. Furthermore, the strategy instances have no state (no instance attributes). You could say they look a lot like plain functions, and you would be right. Example 6-3 is a refactoring of Example 6-1, replacing the concrete strategies with simple functions and removing the Promo abstract class.

*Example 6-3. Order class with discount strategies implemented as functions*

```python
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity


class Order:  # the Context

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion(self)  ❶
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())

❷

def fidelity_promo(order):  ❸
    """5% discount for customers with 1000 or more fidelity points"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0


def bulk_item_promo(order):
    """10% discount for each LineItem with 20 or more units"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount


def large_order_promo(order):
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0
```

❶ To compute a discount, just call the `self.promotion()` function.

❷ No abstract class.

❸ Each strategy is a function.

The code in Example 6-3 is 12 lines shorter than Example 6-1. Using the new `Order` is also a bit simpler, as shown in the Example 6-4 doctests.

*Example 6-4. Sample usage of Order class with promotions as functions*

```
    >>> joe = Customer('John Doe', 0)  ❶
    >>> ann = Customer('Ann Smith', 1100)
    >>> cart = [LineItem('banana', 4, .5),
    ...         LineItem('apple', 10, 1.5),
    ...         LineItem('watermellon', 5, 5.0)]
    >>> Order(joe, cart, fidelity_promo)  ❷
    <Order total: 42.00 due: 42.00>
    >>> Order(ann, cart, fidelity_promo)
    <Order total: 42.00 due: 39.90>
```

```
>>> banana_cart = [LineItem('banana', 30, .5),
...                LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, bulk_item_promo)  ❸
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)
...               for item_code in range(10)]
>>> Order(joe, long_order, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>
```

❶ Same test fixtures as Example 6-1.

❷ To apply a discount strategy to an `Order`, just pass the promotion function as an argument.

❸ A different promotion function is used here and in the next test.

Note the callouts in Example 6-4: there is no need to instantiate a new promotion object with each new order: the functions are ready to use.

It is interesting to note that in *Design Patterns* the authors suggest: "Strategy objects often make good flyweights."[3] A definition of the Flyweight in another part of that work states: "A flyweight is a shared object that can be used in multiple contexts simultaneously."[4] The sharing is recommended to reduce the cost of creating a new concrete strategy object when the same strategy is applied over and over again with every new context—with every new `Order` instance, in our example. So, to overcome a drawback of the Strategy pattern—its runtime cost—the authors recommend applying yet another pattern. Meanwhile, the line count and maintenance cost of your code are piling up.

A thornier use case, with complex concrete strategies holding internal state, may require all the pieces of the Strategy and Flyweight design patterns combined. But often concrete strategies have no internal state; they only deal with data from the context. If that is the case, then by all means use plain old functions instead of coding single-method classes implementing a single-method interface declared in yet another class. A function is more lightweight than an instance of a user-defined class, and there is no need for Flyweight because each strategy function is created just once by Python when it compiles the module. A plain function is also "a shared object that can be used in multiple contexts simultaneously."

Now that we have implemented the Strategy pattern with functions, other possibilities emerge. Suppose you want to create a "meta-strategy" that selects the best available discount for a given `Order`. In the following sections, we present additional refactorings

3. See page 323 of *Design Patterns*.

4. *idem*, p. 196

that implement this requirement using a variety of approaches that leverage functions and modules as objects.

## Choosing the Best Strategy: Simple Approach

Given the same customers and shopping carts from the tests in Example 6-4, we now add three additional tests in Example 6-5.

*Example 6-5. The best_promo function applies all discounts and returns the largest*

```
    >>> Order(joe, long_order, best_promo)  ❶
    <Order total: 10.00 due: 9.30>
    >>> Order(joe, banana_cart, best_promo)  ❷
    <Order total: 30.00 due: 28.50>
    >>> Order(ann, cart, best_promo)  ❸
    <Order total: 42.00 due: 39.90>
```

❶ `best_promo` selected the `larger_order_promo` for customer `joe`.

❷ Here `joe` got the discount from `bulk_item_promo` for ordering lots of bananas.

❸ Checking out with a simple cart, `best_promo` gave loyal customer `ann` the discount for the `fidelity_promo`.

The implementation of `best_promo` is very simple. See Example 6-6.

*Example 6-6. best_promo finds the maximum discount iterating over a list of functions*

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo]  ❶

def best_promo(order):  ❷
    """Select best discount available
    """
    return max(promo(order) for promo in promos)  ❸
```

❶ `promos`: list of the strategies implemented as functions.

❷ `best_promo` takes an instance of `Order` as argument, as do the other `*_promo` functions.

❸ Using a generator expression, we apply each of the functions from `promos` to the `order`, and return the maximum discount computed.

Example 6-6 is straightforward: `promos` is a `list` of functions. Once you get used to the idea that functions are first-class objects, it naturally follows that building data structures holding functions often makes sense.

Although Example 6-6 works and is easy to read, there is some duplication that could lead to a subtle bug: to add a new promotion strategy, we need to code the function and remember to add it to the promos list, or else the new promotion will work when explicitly passed as an argument to Order, but will not be considered by best_promotion.

Read on for a couple of solutions to this issue.

## Finding Strategies in a Module

Modules in Python are also first-class objects, and the standard library provides several functions to handle them. The built-in globals is described as follows in the Python docs:

globals()

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

Example 6-7 is a somewhat hackish way of using globals to help best_promo automatically find the other available *_promo functions.

*Example 6-7. The promos list is built by introspection of the module global namespace*

```
promos = [globals()[name] for name in globals()    ❶
            if name.endswith('_promo')  ❷
            and name != 'best_promo']   ❸

def best_promo(order):
    """Select best discount available
    """
    return max(promo(order) for promo in promos)   ❹
```

❶ Iterate over each name in the dictionary returned by globals().

❷ Select only names that end with the _promo suffix.

❸ Filter out best_promo itself, to avoid an infinite recursion.

❹ No changes inside best_promo.

Another way of collecting the available promotions would be to create a module and put all the strategy functions there, except for best_promo.

In Example 6-8, the only significant change is that the list of strategy functions is built by introspection of a separate module called promotions. Note that Example 6-8 depends on importing the promotions module as well as inspect, which provides high-level introspection functions (the imports are not shown for brevity, because they would normally be at the top of the file).

*Example 6-8. The promos list is built by introspection of a new promotions module*

```
promos = [func for name, func in
                inspect.getmembers(promotions, inspect.isfunction)]

def best_promo(order):
    """Select best discount available
    """
    return max(promo(order) for promo in promos)
```

The function `inspect.getmembers` returns the attributes of an object—in this case, the `promotions` module—optionally filtered by a predicate (a boolean function). We use `inspect.isfunction` to get only the functions from the module.

Example 6-8 works regardless of the names given to the functions; all that matters is that the `promotions` module contains only functions that calculate discounts given orders. Of course, this is an implicit assumption of the code. If someone were to create a function with a different signature in the `promotions` module, then `best_promo` would break while trying to apply it to an order.

We could add more stringent tests to filter the functions, by inspecting their arguments for instance. The point of Example 6-8 is not to offer a complete solution, but to highlight one possible use of module introspection.

A more explicit alternative for dynamically collecting promotional discount functions would be to use a simple decorator. We'll show yet another version of our ecommerce Strategy example in Chapter 7, which deals with function decorators.

In the next section, we discuss Command—another design pattern that is sometimes implemented via single-method classes when plain functions would do.

## Command

Command is another design pattern that can be simplified by the use of functions passed as arguments. Figure 6-2 shows the arrangement of classes in the Command pattern.

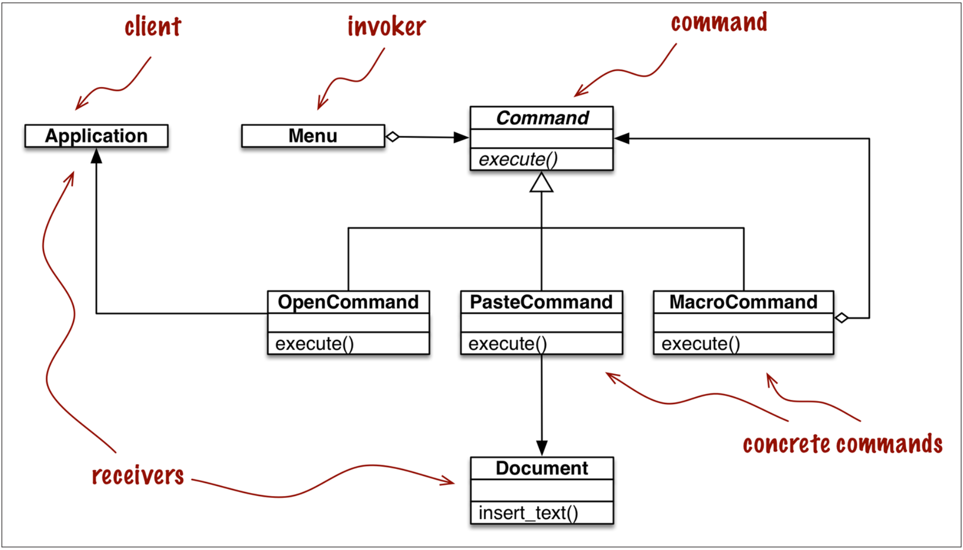

*Figure 6-2. UML class diagram for menu-driven text editor implemented with the Command design pattern. Each command may have a different receiver: the object that implements the action. For PasteCommand, the receiver is the Document. For OpenCommand, the receiver is the application.*

The goal of Command is to decouple an object that invokes an operation (the Invoker) from the provider object that implements it (the Receiver). In the example from *Design Patterns*, each invoker is a menu item in a graphical application, and the receivers are the document being edited or the application itself.

The idea is to put a `Command` object between the two, implementing an interface with a single method, `execute`, which calls some method in the Receiver to perform the desired operation. That way the Invoker does not need to know the interface of the Receiver, and different receivers can be adapted through different `Command` subclasses. The Invoker is configured with a concrete command and calls its `execute` method to operate it. Note in Figure 6-2 that `MacroCommand` may store a sequence of commands; its `execute()` method calls the same method in each command stored.

Quoting from Gamma et al., "Commands are an object-oriented replacement for callbacks." The question is: do we need an object-oriented replacement for callbacks? Sometimes yes, but not always.

Instead of giving the Invoker a `Command` instance, we can simply give it a function. Instead of calling `command.execute()`, the Invoker can just call `command()`. The `MacroCommand` can be implemented with a class implementing `__call__`. Instances of `MacroCommand` would be callables, each holding a list of functions for future invocation, as implemented in Example 6-9.

*Example 6-9. Each instance of MacroCommand has an internal list of commands*

```
class MacroCommand:
    """A command that executes a list of commands"""

    def __init__(self, commands):
        self.commands = list(commands)  # ❶

    def __call__(self):
        for command in self.commands:  # ❷
            command()
```

❶ Building a list from the `commands` arguments ensures that it is iterable and keeps a local copy of the command references in each `MacroCommand` instance.

❷ When an instance of `MacroCommand` is invoked, each command in `self.commands` is called in sequence.

More advanced uses of the Command pattern—to support undo, for example—may require more than a simple callback function. Even then, Python provides a couple of alternatives that deserve consideration:

- A callable instance like `MacroCommand` in Example 6-9 can keep whatever state is necessary, and provide extra methods in addition to `__call__`.
- A closure can be used to hold the internal state of a function between calls.

This concludes our rethinking of the Command pattern with first-class functions. At a high level, the approach here was similar to the one we applied to Strategy: replacing with callables the instances of a participant class that implemented a single-method interface. After all, every Python callable implements a single-method interface, and that method is named `__call__`.

## Chapter Summary

As Peter Norvig pointed out a couple of years after the classic *Design Patterns* book appeared, "16 of 23 patterns have qualitatively simpler implementation in Lisp or Dylan than in C++ for at least some uses of each pattern" (slide 9 of Norvig's "Design Patterns in Dynamic Languages" presentation). Python shares some of the dynamic features of the Lisp and Dylan languages, in particular first-class functions, our focus in this part of the book.

From the same talk quoted at the start of this chapter, in reflecting on the 20th anniversary of *Design Patterns: Elements of Reusable Object-Oriented Software*, Ralph Johnson has stated that one of the failings of the book is "Too much emphasis on patterns as end-points instead of steps in the design patterns."[5] In this chapter, we used the Strategy pattern as a starting point: a working solution that we could simplify using first-class functions.

In many cases, functions or callable objects provide a more natural way of implementing callbacks in Python than mimicking the Strategy or the Command patterns as described by Gamma, Helm, Johnson, and Vlissides. The refactoring of Strategy and the discussion of Command in this chapter are examples of a more general insight: sometimes you may encounter a design pattern or an API that requires that components implement an interface with a single method, and that method has a generic-sounding name such as "execute", "run", or "doIt". Such patterns or APIs often can be implemented with less boilerplate code in Python using first-class functions or other callables.

The message from Peter Norvig's design patterns slides is that the Command and Strategy patterns—along with Template Method and Visitor—can be made simpler or even "invisible" with first-class functions, at least for some applications of these patterns.

## Further Reading

Our discussion of Strategy ended with a suggestion that function decorators could be used to improve on Example 6-8. We also mentioned the use of closures a couple of times in this chapter. Decorators as well as closures are the focus of Chapter 7. That chapter starts with a refactoring of the ecommerce example using a decorator to register available promotions.

"Recipe 8.21. Implementing the Visitor Pattern," in the *Python Cookbook, Third Edition* (O'Reilly), by David Beazley and Brian K. Jones, presents an elegant implementation of the Visitor pattern in which a `NodeVisitor` class handles methods as first-class objects.

On the general topic of design patterns, the choice of readings for the Python programmer is not as broad as what is available to other language communities.

As far as I know, *Learning Python Design Patterns*, by Gennadiy Zlobin (Packt), is the only book entirely devoted to patterns in Python—as of June 2014. But Zlobin's work is quite short (100 pages) and covers eight of the original 23 design patterns.

*Expert Python Programming* by Tarek Ziadé (Packt) is one of the best intermediate-level Python books in the market, and its final chapter, "Useful Design Patterns," presents seven of the classic patterns from a Pythonic perspective.

5. From the same talk quoted at the start of this chapter: "Root Cause Analysis of Some Faults in Design Patterns," presented by Johnson at IME-USP, November 15, 2014.

Alex Martelli has given several talks about Python Design Patterns. There is a video of his EuroPython 2011 presentation and a set of slides on his personal website. I've found different slide decks and videos over the years, of varying lengths, so it is worthwhile to do a thorough search for his name with the words "Python Design Patterns."

Around 2008, Bruce Eckel—author of the excellent *Thinking in Java* (Prentice Hall)—started a book titled *Python 3 Patterns, Recipes and Idioms*. It was to be written by a community of contributors led by Eckel, but six years later it's still incomplete and apparently stalled (as I write this, the last change to the repository is two years old).

There are many books about design patterns in the context of Java, but among them the one I like most is *Head First Design Patterns* by Eric Freeman, Bert Bates, Kathy Sierra, and Elisabeth Robson (O'Reilly). It explains 16 of the 23 classic patterns. If you like the wacky style of the *Head First* series and need an introduction to this topic, you will love that work. However, it is Java-centric.

For a fresh look at patterns from the point of view of a dynamic language with duck typing and first-class functions, *Design Patterns in Ruby* by Russ Olsen (Addison-Wesley) has many insights that are also applicable to Python. In spite of many the syntactic differences, at the semantic level Python and Ruby are closer to each other than to Java or C++.

In Design Patterns in Dynamic Languages (slides), Peter Norvig shows how first-class functions (and other dynamic features) make several of the original design patterns either simpler or unnecessary.

Of course, the original *Design Patterns* book by Gamma et al. is mandatory reading if you are serious about this subject. The Introduction by itself is worth the price. That is the source of the often quoted design principles "Program to an interface, not an implementation" and "Favor object composition over class inheritance."

## Soapbox

Python has first-class functions and first-class types, features that Norvig claims affect 10 of the 23 patterns (slide 10 of Design Patterns in Dynamic Languages). In the next chapter, we'll see that Python also has generic functions ("Generic Functions with Single Dispatch" on page 202), similar to the CLOS multimethods that Gamma et al. suggest as a simpler way to implement the classic Visitor pattern. Norvig, on the other hand, says that multimethods simplify the Builder pattern (slide 10). Matching design patterns to language features is not an exact science.

In classrooms around the world, design patterns are frequently taught using Java examples. I've heard more than one student claim that they were led to believe that the original design patterns are useful in any implementation language. It turns out that the "classic" 23 patterns from the Gamma et al. book apply to "classic" Java very well in spite of being originally presented mostly in the context of C++—a few have Smalltalk examples in the book. But that does not mean every one of those patterns applies equally well in any language. The authors are explicit right at the beginning of their book that "some of our patterns are supported directly by the less common object-oriented languages" (recall full quote on first page of this chapter).

The Python bibliography about design patterns is very thin, compared to that of Java, C++, or Ruby. In "Further Reading" on page 180 I mentioned *Learning Python Design Patterns* by Gennadiy Zlobin, which was published as recently as November 2013. In contrast, Russ Olsen's *Design Patterns in Ruby* was published in 2007 and has 384 pages —284 more than Zlobin's work.

Now that Python is becoming increasingly popular in academia, let's hope more will be written about design patterns in the context of this language. Also, Java 8 introduced method references and anonymous functions, and those highly anticipated features are likely to prompt fresh approaches to patterns in Java—recognizing that as languages evolve, so must our understanding of how to apply the classic design patterns.