PYTHON PROGRAMMING *with* DESIGN PATTERNS

JAMES W. COOPER

Python Programming with Design Patterns

This page intentionally left blank

Python Programming with Design Patterns

James W. Cooper

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town •
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi •
Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the web: informit.com/aw

Library of Congress Control Number: 2021947622

ISBN-13: 978-0-13-757993-8
ISBN-10: 0-13-757993-4

ScoutAutomatedPrintCode

Editor-in-Chief
Mark Taub

Executive Editor
Debra J. Willimans

Development Editor
Chris Zahn

Managing Editor
Sandra Schroeder

Senior Project Editor
Lori Lyons

Copy Editor
Krista Hansing Editorial Services

Production Manager
Remya Divakaran/Codemantra

Indexer
Ken Johnson

Proofreader
Charlotte Kughen

Compositor
Codemantra

Pearson’s Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world’s leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

To Vicki

Contents at a Glance

IV: Behavioral Patterns

V: A Brief Introduction to Python

Table of Contents

Preface

When I began studying Python, I was impressed by how simple coding was and how easy it was to get started writing basic programs. I tried several development environments, and in all cases, I was able to get simple programs running in moments.

The Python syntax was simple, and there were no brackets or semicolons to remember. Other than remembering to use the Tab key (to generate those four-space indentations), coding in Python was easy.

But it was only after I played with Python for a few weeks that I began to see how sophisticated the language really is and how much you can really do with it. Python is a fully object-oriented language, making it easy to create classes that hold their own data without a lot of syntactic fussing.

In fact, I started trying to write some programs that I had written years ago in Java, and I was amazed by how much simpler they were in Python. And with the powerful IDEs, it was hard to make many mistakes.

When I realized how much I could get done quickly in Python, I also realized that it was time to write a book about powerful programs you can write in Python. This led to my writing new, clean, readable versions of the 23 classic design patterns that I had originally coded some years before.

The result is this book, which illustrates the basics of object-oriented programming, visual programming, and how to use all of the classic patterns. You can find complete working code for all these programs on GitHub at https://github.com/jwcnmr/jameswcooper/tree/main/Pythonpatterns.

This book is designed to help Python programmers broaden their knowledge of object-oriented programming (OOP) and the accompanying design patterns.

- If you are new to Python but have experience in other languages, you will be able to charge ahead by reviewing Chapter 31 through Chapter 35 and then starting back at Chapter 1.
- If you are experienced in Python but want to learn about OOP and design patterns, start at Chapter 1. If you like, you can skip Chapter 2 and Chapter 3 and go right through the rest of the book.
- If you are new to programming in general, spend some time going over Chapter 31 through 35 to try some of the programs. Then start on Chapter 1 to learn about OOP and design patterns.

You will likely find that Python is the easiest language you ever learned, as well as the most effortless language for writing the objects you use in design patterns. You'll see what they are for and how to use them in your own work.

In any case, the object-oriented programming methods presented in these pages can help you write better, more reusable program code.

Book Organization

This book is organized into five parts.

Part I, "Introduction"

Design patterns essentially describe how objects can interact effectively. This book starts by introducing objects in Chapter 1, "Introduction to Objects," and providing graphical examples that clearly illustrate how the patterns work.

Chapter 2, "Visual Programming in Python," and Chapter 3, "Visual Programming of Tables of Data," introduce the Python tkinter library, which gives you a way to create windows, buttons, lists, tables, and more with minimal complexity.

Chapter 4, "What Are Design Patterns?", begins the discussion of design patterns by exploring exactly what they are.

Part II, "Creational Patterns"

Part II starts by outlining the first group of patterns that the "Gang of Four" named Creational Patterns.

Chapter 5, "The Factory Pattern," describes the basic Factory pattern, which serves as the simple basis of the three factory patterns that follow. In this chapter, you create a Factory class that decides which of several related classes to use, based on the data itself.

Chapter 6, "The Factory Method Pattern," explores the Factory method. In this pattern, no single class makes the decision as to which subclass to instantiate. Instead, the superclass defers the decision to each subclass.

Chapter 7, "The Abstract Factory Pattern," discusses the Abstract Factory pattern. You can use this pattern when you want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several groups of classes.

Chapter 8, "The Singleton Pattern," looks at the Singleton pattern, which describes a class in which there can be no more than one instance. It provides a single global point of access to that instance. You don't use this pattern all that often, but it is helpful to know how to write it.

In Chapter 9, "The Builder Pattern," you see that the Builder pattern separates the construction of a complex object from its visual representation, so that several different representations can be created, depending on the needs of the program.

Chapter 10, "The Prototype Pattern," shows how to use the Prototype pattern when creating an instance of a class is time consuming or complex. Instead of creating more instances, you make copies of the original instance and modify them as appropriate.

Chapter 11, "Summary of Creational Patterns," just summarizes the patterns in Part II.

Part III, “Structural Patterns”

Part III begins with a short discussion of Structural Patterns.

Chapter 12, “The Adapter Pattern,” examines the Adapter pattern, which is used to convert the programming interface of one class into that of another. Adapters are useful whenever you want unrelated classes to work together in a single program.

Chapter 13, “The Bridge Pattern,” takes up the similar Bridge pattern, which is designed to separate a class’s interface from its implementation. This enables you to vary or replace the implementation without changing the client code.

Chapter 14, “The Composite Pattern,” delves into systems in which a component may be an individual object or may represent a collection of objects. The Composite pattern is designed to accommodate both cases, often in a treelike structure.

In Chapter 15, “The Decorator Pattern,” we look at the Decorator pattern, which provides a way to modify the behavior of individual objects without having to create a new derived class. Although this can apply to visual objects such as buttons, the most common use in Python is to create a kind of macro that modifies the behavior of a single class instance.

In Chapter 16, “The Façade Pattern,” we learn to use the Façade pattern to write a simplifying interface to code that otherwise might be unduly complex. This chapter deals with such an interface to a couple of different databases.

Chapter 17, “The Flyweight Pattern,” describes the Flyweight pattern, which enables you to reduce the number of objects by moving some of the data outside the class. You can consider this approach when you have multiple instances of the same class.

Chapter 18, “The Proxy Pattern,” looks at the Proxy pattern, which is used when you need to represent an object that is complex or time consuming to create, by a simpler one. If creating an object is expensive in time or computer resources, Proxy enables you to postpone creation until you need the actual object.

Chapter 19, “Summary of Structural Patterns,” summarizes these Structural patterns.

Part IV, “Behavioral Patterns”

Part IV outlines the Behavioral Patterns.

Chapter 20, “Chain of Responsibility Pattern,” looks at how the Chain of Responsibility pattern allows a decoupling between objects by passing a request from one object to the next in a chain until the request is recognized.

Chapter 21, “The Command Pattern,” shows how the Command pattern uses simple objects to represent the execution of software commands. Additionally, this pattern enables you to support logging and undoable operations.

Chapter 22, “The Interpreter Pattern,” looks at the Interpreter pattern, which provides a definition of how to create a little execution language and include it in a program.

In Chapter 23, “The Iterator Pattern,” we explore the well-known Iterator pattern, which describes the formal ways you can move through a collection of data items.

Chapter 24, “The Mediator Pattern,” takes up the important Mediator pattern. This pattern defines how communication between objects can be simplified by using a separate object to keep all objects from having to know about each other.

Chapter 25, “The Memento Pattern,” saves the internal state of an object, so you can restore it later.

In Chapter 26, “The Observer Pattern,” we look at the Observer pattern, which enables you to define the way a number of objects can be notified of a change in a program state.

Chapter 27, “The State Pattern,” describes the State pattern, which allows an object to modify its behavior when its internal state changes.

Chapter 28, “The Strategy Pattern,” describes the Strategy pattern, which, like the State pattern, switches easily between algorithms without any monolithic conditional statements. The difference between the State and Strategy patterns is that the user generally chooses which of several strategies to apply.

In Chapter 29, “The Template Pattern,” we look at the Template pattern. This pattern formalizes the idea of defining an algorithm in a class but leaves some of the details to be implemented in subclasses. In other words, if your base class is an abstract class, as often happens in these design patterns, you are using a simple form of the Template pattern.

Chapter 30, “The Visitor Pattern,” explores The Visitor pattern, which turns the tables on the object-oriented model and creates an external class to act on data in other classes. This is useful if there are a fair number of instances of a small number of classes and you want to perform some operation that involves all or most of them.

Part V, “A Brief Introduction to Python”

In this last section of the book, we provide a succinct summary of the Python language. If you are only passingly familiar with Python, this will get you up to speed. It is sufficiently thorough to instruct beginner as well.

In Chapter 31, “Variables and Syntax in Python,” we review the basic Python variables and syntax, and in Chapter 32, “Making Decisions in Python,” we illustrate the ways your programs can make decisions.

In Chapter 33, “Development Environments,” we provide a short summary of the most common development environments, and in Chapter 34, “Python Collections and Files,” we discuss arrays and files.

Finally in Chapter 35, “Functions,” we take up how to use functions on Python.

Enjoy writing design patterns and learning the ins and outs of the powerful Python language!

Register Your Book

Acknowledgments

I must start by thanking the late John Vlissides, one of the original “Gang of Four,” for his clear explanations of several points about these design patterns. He worked just a few doors down from me at IBM Research and didn’t mind my dropping in for a chat about patterns from time to time.

I also really appreciated early supportive comments from Arianne Dee and Ausif Mahmood, as well as Vaughn Cooper.

Of course, my editor, Debra J. Williams, has been both supportive and creative in helping me bring this project to fruition, as have the reviewers, Nick Cohron and Regina R. Monaco. And from a development point of view, Chris Zahn has been terrific.

I hope you enjoy writing patterns in Python as much as I have.

James Cooper
Wilton, CT
July 2021

About the Author

James W. Cooper holds a PhD in chemistry and worked in academia, for the scientific instrument industry, and for IBM for 25 years, primarily as a computer scientist at IBM's Thomas J. Watson Research Center. Now retired, he is the author of 20 books, including 3 on design patterns in various languages. His most recent books are *Flameout: The Rise and Fall of IBM Instruments* (2019) and *Food Myths Debunked* (2014).

James holds 11 patents and has written 60 columns for *JavaPro Magazine*. He has also written nearly 1,000 columns for the now vanished Examiner.com on foods and chemistry, and he currently writes his own blog: FoodScienceInstitute.com. Recently, he has written columns on Python for Medium.com and Substack.

He is also involved in local theater groups and is the treasurer for Troupers Light Opera, where he performs regularly.

This page intentionally left blank

4

What Are Design Patterns?

Sitting at your desk in front of your workstation, you stare into space, trying to figure out how to write a new program feature. You know intuitively what must be done, what data and what objects come into play, but you have this underlying feeling that there is a more elegant and general way to write this program.

In fact, you probably don't write any code until you can build a picture in your mind of what the code does and how the pieces of the code interact. The more you can picture this "organic whole," the more likely you are to feel comfortable that you have developed the best solution to the problem. If you don't grasp this whole right away, you might keep staring out the window for a time, even though the basic solution to the problem is quite obvious.

In one sense, you feel that the most elegant solution will be more reusable and more maintainable, but even if you are the sole likely programmer, you feel reassured when you have designed a solution that is relatively elegant and doesn't expose too many internal inelegancies.

One of the main reasons computer science researchers began to recognize design patterns is to satisfy this need for elegant but simple reusable solutions. The term *design patterns* sounds a bit formal to the uninitiated and can be somewhat off-putting when you first encounter it. But, in fact, design patterns are just convenient ways of reusing object-oriented code between projects and programmers. The idea behind design patterns is simple: to write down and catalog common interactions between objects that programmers have frequently found useful.

One frequently cited pattern from early literature on programming frameworks is the Model-View-Controller framework for Smalltalk (Krasner and Pope, 1988), which divided the user interface problem into three parts. The parts were referred to as a *data model*, containing the computational parts of the program; the *view,* which presents the user interface; and the *controller*, which interacts between the user and the view (see Figure 4-1).

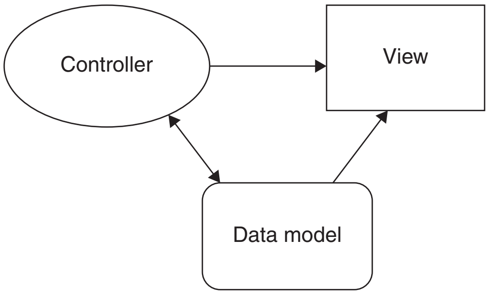

Figure 4-1 Model-View-Controller illustration

Each of these aspects of the problem is a separate object, and each has its own rules for managing its data. Communication among the user, the GUI, and the data should be carefully controlled, and this separation of functions accomplished that very nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful design pattern.

In other words, design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming. If you have been trying to keep objects minding their own business, you are probably already using some of the common design patterns.

Design patterns started to be recognized more formally in the early 1990s by Erich Gamma,[1] who described patterns incorporated in the GUI application framework ET++. The culmination of these discussions and a number of technical meetings was the book *Design Patterns: Elements of Reusable Software,* by Gamma, Helm, Johnson, and Vlissides.[2] This best-selling book, commonly referred to as the Gang of Four, or "GoF" book, has had a powerful impact on programmers seeking to understand how to use design patterns. It describes 23 commonly occurring and generally useful patterns and comments on how and when you might apply them. Throughout the following chapters, we refer to this groundbreaking book as *Design Patterns*.

Since the publication of the original *Design Patterns*, many other useful books have been published. These include our popular *Java Design Patterns: A Tutorial*[3] and an analogous book on C# design patterns.[4] Rhodes[5] maintains an interesting site describing how Python can make use of design patterns, as well.

Defining Design Patterns

We all talk about the way we do things in our everyday work, hobbies, and home life, and recognize repeating patterns all the time:

- Sticky buns are like dinner rolls, but I add brown sugar and nut filling to them.
- Her front garden is like mine, but in mine I use *astilbe*.
- This end table is constructed like that one, but in this one, the doors replace drawers.

We see the same thing in programming, when we tell a colleague how we accomplished a tricky bit of programming so that they don't have to re-create it from scratch. We simply recognize effective ways for objects to communicate while maintaining their own separate existences.

To summarize:

Design patterns are frequently used algorithms that describe convenient ways for classes to communicate.

It has become apparent that you don't just *write* a design pattern off the top of your head. In fact, most such patterns are *discovered* rather than written. The process of looking for these patterns is called *pattern mining* and is worthy of a book of its own.

The 23 design patterns selected for inclusion in the original *Design Patterns* book were patterns that had several known applications and were on a middle level of generality, where they could easily cross application areas and encompass several objects.

The authors divided these patterns into three types: creational, structural, and behavioral.

- *Creational patterns* create objects for you instead of having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- *Behavioral patterns* help you define the communication between objects in your system and control the flow in a complex program.

The Learning Process

We have found that learning design patterns is a multiple-step process:

1. Acceptance
2. Recognition
3. Internalization

First, you accept the premise that design patterns are important in your work. Then you recognize that you need to read about design patterns in order to determine when you might use them. Finally, you internalize the patterns in sufficient detail that you know which ones might help you solve a given design problem.

For some lucky people, design patterns are obvious tools, and they grasp their essential utility just by reading summaries of the patterns. For many of the rest of us, there is a slow induction period after we've read about a pattern, followed by the proverbial "Aha!" when we see how we can apply them in our work. These chapters help take you to that final stage of internalization by providing complete, working programs that you can try out for yourself.

The examples in *Design Patterns* are brief and are written in either C++ or, in some cases, Smalltalk. If you are working in another language, it is helpful to have the pattern examples in your language of choice. This part of the book attempts to fill that need for Python programmers.

Notes on Object-Oriented Approaches

The fundamental reason for using design patterns is to keep classes separated and prevent them from having to know too much about one another. Equally important, using these patterns helps you avoid reinventing the wheel and enables you to describe your programming approach succinctly in terms other programmers can easily understand.

There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance. Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all the methods of that parent class. It also has access to all its variables. However, by starting your inheritance hierarchy with a complete, working class, you might be unduly restricting yourself as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggests that you always

Program to an interface and not to an implementation.

Putting this more succinctly, you should define the top of any class hierarchy with an *abstract* class or an *interface*, which implements no methods but simply defines the methods that class will support. Then in all your derived classes, you have more freedom to implement these methods as best suits your purposes.

Python does not directly support interfaces, but it does let you write abstract classes, where the methods have no implementation. Remember the `comd` interface to the `DButton` class:

```
class DButton(Button):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        super().config(command=self.comd)

    # abstract method to be called by children
    def comd(self): pass
```

This is a good example of an abstract class. Here you fill in the code for the command method in the derived button classes. As you will see, it is also an example of the Command design pattern.

The other major concept you should recognize is *object composition.* We have already seen this approach in the `Statelist` examples. Object composition is simply the construction of objects that contain others—the encapsulation of several objects inside another one. Many beginning OO programmers tend to use inheritance to solve every problem, but as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that works best for what you want to accomplish without having all the methods of the parent classes. Thus, the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

At first this seems contrary to the customs of OO programming, but you will see any number of cases among the design patterns where we find that inclusion of one or more objects inside another is the preferred method.

Python Design Patterns

The following chapters discuss each of the 23 design patterns featured in the *Design Patterns* book, along with at least one working program example for that pattern. The programs have some sort of visual interface as well to make them more immediate to you.

Which design patterns are most useful? This depends on the individual programmer. The ones we use the most are Command, Factory, Decorator, Façade, and Mediator, but we have used nearly every one at some point.

References

1. Erich Gamma, *Object-Oriented Software Development based on ET++,* (in German) (Springer-Verlag, Berlin, 1992).
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995).
3. James Cooper, *Java Design Patterns: A Tutorial* (Boston: Addison-Wesley: 2000).
4. James Cooper, *C# Design Patterns: A Tutorial* (Boston: Addison-Wesley, 2003).
5. Brandon Rhodes, “Python Design Patterns,” https://python-patterns.guide.

Index

Symbols

A

C

D

E

F

G

H

I

J

K

L

N

O

Q

R

S

T

W - X - Y - Z