

MODERN OBJECT-ORIENTED PYTHON



**WRITE CLEAN, PYTHONIC CODE
WITH OOP**

Modern Object-Oriented Python

Real Python

Modern Object-Oriented Python

First Edition

by the *Real Python* Tutorial Team,
curated and edited by Stephen Gruppetta

Copyright © *Real Python* (realpython.com), 2012–2026

For online information and ordering of this and other books by *Real Python*, please visit realpython.com. For more information, please contact us at info@realpython.com.

ISBN: 9781775093367 (paperback)

ISBN: 9781775093374 (electronic)

Cover design by Aldren Santos

Additional review and formatting by Dan Bader and Martin Breuss

“Python” and the Python logos are trademarks or registered trademarks of the Python Software Foundation, used by *Real Python* with permission from the Foundation.

Thank you for downloading this ebook. This ebook is licensed for your personal enjoyment only. This ebook may not be resold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. If you’re reading this book and did not purchase it, or if it was not purchased for your use only, then please return to realpython.com/modern-oop and purchase your own copy. Thank you for respecting the hard work behind it.

This is a sample from “Modern Object-Oriented Python: Write Clean, Pythonic Code with OOP”

Object-oriented programming (OOP) is one of the most powerful tools in Python, but knowing how to use it *well* is what separates clean, maintainable code from a tangled mess.

This book takes you from the core building blocks of OOP all the way to the design principles that professionals rely on to keep large code-bases healthy. At the end of the book you’ll be able to:

- Model real-world entities with classes, data attributes, and methods, and customize object behavior with Python’s special methods
- Reuse and organize code with inheritance and composition, manage attribute access cleanly with properties, and cut boilerplate using data classes
- Apply the SOLID principles to design robust, maintainable object-oriented code that’s a pleasure to read, extend, and maintain

If you enjoyed the sample chapters you can purchase a full version of the book at realpython.com/modern-oop

Contents

Contents	4
Foreword	7
1 Introduction to Object-Oriented Programming	13
1.1 What Is Object-Oriented Programming in Python? .	13
1.2 How Do You Define a Class in Python?	14
1.3 How Do You Instantiate a Class in Python?	19
1.4 Understanding the Benefits of Using Classes in Python	23
1.5 Conclusion	24
2 Data Attributes	26
2.1 Attaching Data to Classes and Instances	26
2.2 Conclusion	37
3 Methods	39
3.1 Instance, Class, and Static Methods	40
3.2 Seeing Them in Action	42
3.3 The Pizza Class	44
3.4 Modifying Pizza Ingredients With Instance Methods .	46
3.5 Delicious Pizza Factories With Class Methods	47
3.6 When To Use Static Methods	49
3.7 Conclusion	52

4	Special Methods	53
4.1	Getting to Know Python's Special Methods	54
4.2	Representing Objects as Strings	56
4.3	Supporting Operators in Custom Classes	63
4.4	Supporting Iteration With Iterators and Iterables . . .	77
4.5	Implementing Custom Sequences and Mappings . . .	82
4.6	Conclusion	84
5	Creation and Initialization of Objects	86
5.1	Python's Class Constructors and Instantiation	87
5.2	Object Initialization With <code>__init__()</code>	92
5.3	Defining Multiple Class Constructors	95
5.4	Object Creation With <code>__new__()</code>	110
5.5	Conclusion	118
6	Inheritance and Composition	120
6.1	What Are Inheritance and Composition?	121
6.2	Inheritance in Python	122
6.3	Composition in Python	137
6.4	Choosing Between Inheritance and Composition . . .	141
6.5	Conclusion	147
7	Properties	148
7.1	Managing Attributes in Your Classes	148
7.2	Getting Started With Python's <code>property()</code>	153
7.3	Deciding When to Use Properties	162
7.4	Providing Read-Only Attributes	162
7.5	Creating Read-Write Attributes	165
7.6	Providing Write-Only Attributes	167
7.7	Putting Python's <code>property()</code> Into Action	168
7.8	Overriding Properties in Subclasses	182
7.9	Conclusion	184

8	Data Classes	186
8.1	What's a Data Class?	187
8.2	Alternatives to Data Classes	189
8.3	Basic Data Classes	192
8.4	More Flexible Data Classes	196
8.5	Immutable Data Classes	206
8.6	Inheritance	208
8.7	Conclusion	210
9	SOLID Principles to Improve OOP Design	212
9.1	The SOLID Principles of Object-Oriented Design . . .	212
9.2	Single-Responsibility Principle (SRP)	213
9.3	Open-Closed Principle (OCP)	216
9.4	Liskov Substitution Principle (LSP)	220
9.5	Interface Segregation Principle (ISP)	223
9.6	Dependency Inversion Principle (DIP)	227
9.7	Conclusion	229
	Concluding Remarks	231

Foreword

“Everything is an object in Python.” This is a phrase you’ll hear often, but what does it really mean? This book answers this question by introducing the fundamentals of Object-Oriented Programming in Python.

Python gives you a lot of freedom when choosing the style of programming you wish to use. You can write useful programs with functions, built-in data types, and modules alone. However, as your code grows, you’ll need to provide structure to ensure your code is readable, maintainable, and as close to bug-free as possible.

Object-oriented programming is one of the coding paradigms you can use to structure your code. It’s a style of coding based on grouping related data and behavior into single units called objects. Remember the phrase “everything is an object in Python”. It’s because the Python language is built using object-oriented principles. As you dive deeper into any Python topic, you’ll find object-oriented programming concepts underpinning the language.

Who This Book Is For

This book is for programmers who already know the Python basics. You should be comfortable with variables, functions, conditionals, loops, and Python’s built-in data types.

You may have used classes before without feeling fully confident, or you learned OOP in another language and found Python’s approach

confusing. Perhaps you avoided classes altogether because they felt unnecessary or overcomplicated.

If any of that sounds familiar, you're the intended audience.

A Pythonic Approach to OOP

Throughout this book, OOP is treated as a practical technique rather than a strict set of rules to follow dogmatically. You'll focus on writing clear, readable code and on understanding Python's object model, instead of memorizing patterns or terminology.

OOP, as this paradigm is referred to, is an essential tool in modern programming. Not everyone in the programming community loves it; however, it's everywhere in Python. Still, it's not a requirement when coding in Python. OOP is one of many programming styles you can choose. You can write many programs without defining classes.

This book will explore what's good about OOP, but it'll also highlight pitfalls you should avoid. You'll learn to make practical choices about when and how to use OOP features in your own projects. You'll also see when not to use classes and what alternatives Python offers.

This book exists to help you use object-oriented programming in Python **when it makes your code better**, and not just because it exists.

How This Book Is Organized

You'll start with the fundamentals: classes, instances, data attributes, and methods. From there, you'll dig deeper into Python's object model by exploring special methods, object creation, and initialization.

Later chapters focus on design choices, such as inheritance versus composition, managing attributes with properties, and using data classes. The book closes with a practical look at the SOLID principles and how they apply to Python code.

You can read this book from start to finish or return to specific chapters as a reference. The examples are designed to be run, modified, and explored. Revisiting earlier sections as your understanding grows is encouraged.

The Real Python Philosophy

This book is based on Real Python tutorials and teaching experience. The focus is on clear explanations and examples, and understanding *why* things work the way they do.

There's no heavy theory for its own sake, and no attempt to force a single “correct” way of writing object-oriented Python. The goal is to help you make informed choices and write code you'll be happy to maintain.

Learning by Doing

This book is all about learning by doing. Be sure to actually type in the code examples you encounter rather than copying and pasting them. You'll learn the concepts better and pick up the syntax faster if you type each line yourself. If you make a mistake, the simple act of correcting it will help you learn how to debug your code.

The code found in this book has been tested with Python 3.14 on Windows, macOS, and Linux.

Formatting Conventions

Code blocks are used to present example code:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Monospace text denotes filenames, such as `dog.py`, and inline code references like `.name` and `__init__()`.

Bold text denotes a new or important term when it's first introduced.

Throughout the book, you'll encounter highlighted boxes that call attention to specific types of information:

Note

Notes provide additional context, clarify terminology, or point to related topics elsewhere in the book.

Tips offer best practices, recommended techniques, or useful shortcuts.

Warnings highlight common pitfalls, security risks, or things that could lead to unexpected behavior.

Example Code License

The example Python scripts in this book are licensed under a [Creative Commons Public Domain \(CCo\) License](#). You're welcome to use any portion of the code for any purpose in your own programs.

Feedback and Errata

Your feedback helps make this book better. If you find an error in the text or code, if a topic is confusing, or if you have suggestions for improvement, we'd love to hear from you:

realpython.com/modern-oop/feedback

Let's Get Started

Learning OOP is a process. Some ideas will click immediately, while others may take time and repetition. That's normal. By the end of this book, you'll have a solid understanding of Python's object-oriented features and the confidence to use them in the right places.

Let's begin.

About the Authors

At *Real Python* you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The realpython.com website launched in 2012 and currently helps more than a million Python developers each month with free programming tutorials and in-depth learning resources.

Modern Object-Oriented Python was curated and edited by Stephen Gruppetta and produced by a small team at *Real Python*, building on in-depth tutorials by the contributors below.

Stephen Gruppetta earned a PhD in physics and spent over a decade in academia before becoming a Python educator. He's always looking for simple ways to explain complex things in Python. At *Real Python*, Stephen designs and runs live cohort courses and workshops, and contributes to tutorials and video courses.

Dan Bader is the owner and editor in chief of *Real Python* and the main developer of the realpython.com learning platform. He's the author of *Python Tricks*, a best-selling programming book for intermediate Python developers. Dan handled the formatting and typesetting of this book, and he also wrote one of its source tutorials.

Martin Breuss reviewed the book for technical accuracy. He's *Real Python*'s Head of Content Strategy, where he leads developer education for a community of more than a million Python programmers, drawing on a background in instructional design, curriculum development, code review, and live teaching.

Sarah Jane Ross coordinated the production of this book. As *Real Python*'s Content Operations Manager, she brings over a decade of remote-workflow expertise to the metrics, processes, and pipelines that keep the team delivering high-quality Python education.

Aldren Santos designed the cover and created the artwork. He holds a degree in Multimedia Arts and Sciences from Mapúa University and brings *Real Python* tutorials to life with original illustrations that make learning more visual and approachable.

The chapters in this book are adapted from tutorials by the following members of the *Real Python* team:

David Amos is a writer, programmer, and mathematician passionate about exploring mathematics through code. He's also the author of Real Python's *Python Basics: A Practical Introduction to Python 3*.

Geir Arne Hjelle is an avid Pythonista and a member of the *Real Python* tutorial team. From his home in Oslo, Norway, he works in machine learning and data science.

Leodanis Pozo Ramos is a self-taught Python developer, educator, and technical writer with over ten years of experience. At *Real Python*, he creates in-depth tutorials covering core language features, GUI programming, and command-line interface development.

Isaac Rodriguez is a software development lead who builds, leads, and mentors development teams. He has been writing code for over twenty years and focuses on cloud services and back-end applications using Python, among other languages.

Kyle Stratis is a self-taught Python developer and the founder of Stratis Data Labs. He works at the intersection of data engineering and data science, especially machine learning and deep learning.

Brenda Weleschuk edited many of the original tutorials behind these chapters. As *Real Python*'s editor, she reviews and refines learning resources for clarity, accuracy, and a learner-first focus, bringing a B.A. in English and a background in instructional design, technical writing, and teaching to the content pipeline.

Chapter 1

Introduction to Object-Oriented Programming

A computer program stores data and performs actions with it. Data is stored in data structures, and functions deal with the actions you need to perform on the data. **Object-oriented programming (OOP)** is a way of structuring a program by bundling related data and behaviors into a single unit. These units are called **objects**.

OOP is a cornerstone of modern software development, providing a structured way to design and organize code that helps you model real-world things and how they interact with each other.

1.1 What Is Object-Oriented Programming in Python?

Python implements the object-oriented programming paradigm through **classes**. They provide an elegant way to define reusable pieces of code that encapsulate data and behavior in a single entity.

For example, an object could represent a person with characteristics like a name, age, and address and behaviors such as walking, talking, breathing, and running. Or it could represent an email with characteristics like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete or abstract things and ideas as well as relations between them. OOP models entities as software objects that have some data associated with them and can perform certain operations.

The key takeaway is that objects are at the center of object-oriented programming in Python. In other programming paradigms, data structures only represent the data, and separate functions deal with behavior. In OOP, objects inform the overall structure of the program by combining data and behavior.

1.2 How Do You Define a Class in Python?

In Python, you define a class by using the `class` keyword followed by a name and a colon. You often proceed by using `__init__()` to declare which attributes each instance of the class should have. The `__init__()` method has a special meaning in Python classes. This method is known as the object **initializer** because it defines and sets the initial values for the object's attributes:

```
class Employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

This code creates an `Employee` class to represent employees in an organization. But what does all of that mean? And why do you even need classes in the first place? Take a step back and consider using built-in, primitive **data structures** as an alternative.

Primitive data structures, like [numbers](#), [strings](#), and [lists](#), are designed to represent straightforward pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors. What if you want to represent something more complex?

For example, you might want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a [list](#):

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where you declared the `kirk` list, will you remember that the element with index 0 is the employee's name?

Second, it can introduce errors if employees don't have the same number of elements in their respective lists. The age is missing in the `mccoy` list above, so `mccoy[1]` will return "Chief Medical Officer" instead of [Dr. McCoy's](#) age.

A great way to make this type of code more manageable and more maintainable is to use classes.

Classes vs Instances

Classes allow you to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object that's created from the class can perform with its data.

In this chapter, you'll create a `Dog` class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how to define something. It doesn't actually contain any data. The `Dog` class you'll create specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an **instance** is an object that's built from a class and contains real data. An instance of the `Dog` class is not a blueprint anymore. It represents an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that you've filled out with information. Just like many people can fill out the same form with their own unique information, you can create many instances from a single class.

Class Definition

You start all class definitions with the `class` keyword, then add the name of the class and a colon. Python will consider any code that you indent below the class definition as part of the class's body.

Here's an example of a `Dog` class defined in a Python file named `dog.py`:

```
# dog.py
class Dog:
    pass
```

The body of the `Dog` class consists of a single statement: the `pass` keyword. Python programmers often use `pass` as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Note

Python class names are written in [CapitalizedWords notation](#) by convention. This notation is also referred to as PascalCase or UpperCamelCase. For example, a class for a specific breed of dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

The `Dog` class isn't very interesting right now, so you'll spruce it up a bit by defining some characteristics that all `Dog` objects should have. There are several that you can choose from, including name, age, coat color, and breed. To keep the example concise, you'll just use name and age.

You define the characteristics that all `Dog` objects must have in a method called `__init__()`. Every time you create a new `Dog` object, `__init__()` sets the initial **state** of the object by assigning the values of the object's characteristics. That is, `__init__()` initializes each new instance of the class.

You can give `__init__()` any number of parameters, but the first parameter will always be `self`. When you create a new class instance, Python automatically passes the instance to the `self` parameter in `__init__()` so that Python can define the new **attributes** on the object.

Update the `Dog` class with an `__init__()` method that creates `.name` and `.age` attributes:

```
# dog.py
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Make sure that you indent the `__init__()` method's signature by four spaces, which is the default spacing for one indentation level in Python. The body of the method has eight spaces. This indentation

is important. It tells Python that the `__init__()` method belongs to the `Dog` class.

In the body of `__init__()`, there are two statements using the `self` variable:

1. `self.name = name` creates an attribute called `.name` and assigns the value of the `name` parameter to it.
2. `self.age = age` creates an attribute called `.age` and assigns the value of the `age` parameter to it.

Note

In this book, all attributes are displayed with a leading dot, such as `.name` and `.age`. In Python code, the dot must be preceded by a reference to the object when accessing attributes, such as `self.name` or `miles.name` if `miles` is the name of a `Dog` instance.

Attributes created in `__init__()` are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All `Dog` objects have a name and an age, but the values for the `.name` and `.age` attributes will vary depending on the `Dog` instance.

On the other hand, **class attributes** are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to an **identifier** outside of `__init__()`.

For example, the following `Dog` class has a class attribute called `.species` with the value `"Canis familiaris"`:

```
# dog.py
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

You define class attributes directly beneath the first line of the class name and indent them by one indentation level. You always need to assign them an initial value. When you create an instance of the class, Python automatically creates and assigns class attributes to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that you have a `Dog` class, it's time to create some dogs.

1.3 How Do You Instantiate a Class in Python?

Creating a new object from a class is called **instantiating** a class. You can create a new object by typing the name of the class, followed by opening and closing parentheses.

Creating New Instances

Create the `Dog` class again in a [REPL](#) session, starting from the most basic version:

```
>>> class Dog:
...     pass
...
>>> Dog()
<__main__.Dog object at 0x106702d30>
```

You first create a new `Dog` class with no attributes or methods, and then you instantiate the `Dog` class to create a `Dog` object.

In the output above, you can see that you now have a new `Dog` object at `0x106702d30`. This funny-looking series of letters and numbers is a **memory address** that indicates where Python stores the `Dog` object

in your computer's memory. Note that the address on your screen will be different.

Now instantiate the `Dog` class a second time to create another `Dog` object:

```
>>> Dog()
<__main__.Dog object at 0x0004ccc90>
```

The new `Dog` instance is located at a different memory address. That's because it's an entirely new instance and is completely unique from the first `Dog` object you created.

To see this another way, type the following:

```
>>> first_dog = Dog()
>>> second_dog = Dog()
>>> first_dog is second_dog
False
```

In this code, you create two new `Dog` objects and assign them to the variables `first_dog` and `second_dog`. When you compare them using the `is` operator, the result is `False`. Even though `first_dog` and `second_dog` are both instances of the `Dog` class, they represent two distinct objects in memory.

Class and Instance Attributes

Now create a new `Dog` class with a class attribute called `.species` and two instance attributes called `.name` and `.age`:

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
... 
```

To instantiate this `Dog` class, you need to provide values for `name` and `age`. If you don't, then Python raises a `TypeError`:

```
>>> Dog()
Traceback (most recent call last):
...
TypeError: Dog.__init__() missing 2 required positional arguments:
      'name' and 'age'
```

To pass arguments to the `name` and `age` parameters, put values into the parentheses after the class name:

```
>>> miles = Dog("Miles", 4)
>>> buddy = Dog("Buddy", 9)
```

This creates two new `Dog` instances, one for a four-year-old dog named Miles and one for a nine-year-old dog named Buddy.

The `Dog` class's `__init__()` method has three parameters, so why are you only passing two arguments to it in the example?

When you instantiate the `Dog` class, Python creates a new instance of `Dog` and passes it to the first parameter of `__init__()`. This takes care of the `self` parameter, so you only need to worry about the `name` and `age` parameters.

Note

Behind the scenes, Python both creates and initializes a new object when you use this syntax. You'll read more about this in the chapter *Creation and Initialization of Objects*.

After you create the `Dog` instances, you can access their instance attributes using **dot notation**:

```
>>> miles.name
'Miles'
>>> miles.age
4

>>> buddy.name
'Buddy'
>>> buddy.age
9
```

Note that the variable names `miles` and `buddy` mirror the dogs' names stored in `.name` in this example, but this isn't necessary. You can use any valid variable name for these objects. You can access class attributes in the same way:

```
>>> buddy.species
'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All `Dog` instances have `.species`, `.name`, and `.age` attributes, so you can use those attributes with confidence, knowing that they'll always return a value.

Although the attributes are guaranteed to exist, their values *can* change dynamically:

```
>>> buddy.age = 10
>>> buddy.age
10

>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

In this example, you change the `.age` attribute of the `buddy` object to 10. Then you change the `.species` attribute of the `miles` object to `"Felis silvestris"`, which is a species of cat. That makes Miles a pretty strange dog, but it's valid Python.

The key takeaway here is that custom objects are mutable by default. An object is mutable if you can alter it dynamically. For example, lists and [dictionaries](#) are mutable, but strings and tuples are [immutable](#).

1.4 Understanding the Benefits of Using Classes in Python

Is it worth using classes in Python? Absolutely. Classes are the building blocks of object-oriented programming. They allow you to use Python's features effectively while writing and organizing your code. By learning about classes, you'll be able to take advantage of all the benefits that they provide. You'll also learn when to use them and when you don't need them.

With classes, you can:

- **Model and solve complex real-world problems:** You'll find many situations where the objects in your code map to real-world objects. Classes can help you think about complex problems, which will result in better solutions to your programming problems.
- **Reuse code and avoid repetition:** You can define hierarchies of related classes. The base classes at the top of a hierarchy provide common functionality that you can reuse later in the subclasses down the hierarchy. This allows you to reduce code duplication and promote code reuse. You'll read more about inheritance in the chapter *Inheritance and Composition*.
- **Encapsulate related data and behaviors in a single entity:** You can use Python classes to bundle together related attributes and methods in a single entity, the object. This helps you better organize your code using modular and autonomous entities that you can even reuse across multiple projects.
- **Abstract away the implementation details of concepts and objects:** You can use classes to abstract away the implementation details of core concepts and objects. This will help you

provide your users with intuitive interfaces to process complex data and behaviors.

- **Enable polymorphism with common interfaces:** You can implement a particular interface in several slightly different classes and use them interchangeably in your code. This will make your code more flexible and adaptable.

In short, Python classes can help you write more organized, maintainable, and reusable code. They're a great tool to have under your belt.

Note

In Python, the **interface** of a class is the set of public attributes, including data attributes and methods, that define how external code can interact with instances of the class, such as accessing or modifying its data and invoking its behaviors.

You'll learn more about interfaces throughout this book.

However, don't be tempted to use classes for everything in Python. In some situations, they can overcomplicate your solutions and they aren't always necessary. For data storage without behavior, tools like `@dataclass` or `namedtuple` may be more appropriate. You'll explore these tools later in this book. For programs that don't involve complex structures or reusable components, functions and modules might be enough.

Classes are great, but don't turn them into a one-size-fits-all type of tool. Start your code as simply as possible. If the need for a class appears, then go for it.

1.5 Conclusion

In this chapter, you learned how object-oriented programming is a paradigm that brings data and behavior together in a single unit called an object. Classes are the blueprint you use to create these objects.

You learned how to define a class using the `class` keyword and how to create instances of the class by using parentheses after the class name.

This chapter also introduced attributes that store data in an object. The next chapter covers data attributes in more detail. Chapter 3 will introduce you to another type of attribute: the method. Methods deal with the behavior within an object.

With the knowledge you've gained in this chapter, you're ready to continue exploring object-oriented programming.

Chapter 2

Data Attributes

As you’ve learned in the previous chapter, classes are great when you must bundle data and behavior together in a single entity. There are two types of attributes: data attributes and methods. The data will come in the form of data attributes, while the behavior will come as methods. You’ll read more about methods in the next chapter. In this chapter, you’ll learn how to add, access, and modify data attributes in your custom classes.

Note

The term *attribute* technically refers to both data attributes and methods. Programmers sometimes loosely use the term as a shortcut for data attributes. You can usually infer the meaning from the context. This book uses the term *data attribute* when it’s important to clarify when something refers only to data attributes and not methods.

2.1 Attaching Data to Classes and Instances

In Chapter 1, you created the `Dog` class, and you added three data attributes: `.name`, `.age`, and `.species`. However, you defined the first two

attributes differently from the third one. You can start by describing the two types of data attributes in Python:

1. **Class attributes:** A variable that you define in the class body directly. Class attributes belong to their containing class. Their data is common to the class and all its instances.
2. **Instance attributes:** A variable that you define inside an instance method using the `self` argument and dot notation, like in `self.attribute = value`. Instance attributes belong to a specific instance of a given class. Their data is only available to that instance and defines its state.

Both types of attributes have their specific use cases. Instance attributes are, by far, the most common type of data attribute that you'll use in your day-to-day coding, but class attributes also come in handy.

Class Attributes

Class attributes are variables that you define directly in the class body but outside of any method. These attributes are tied to the class itself rather than to particular objects of that class.

All the objects that you create from a particular class share the same class attributes with the same original values. Because of this reason, changing a class attribute affects all the derived objects.

As an example, say that you want to create a class that keeps an internal count of the instances you create. In this case, you can use a class attribute:

```
>>> class ObjectCounter:
...     num_instances = 0
...     def __init__(self):
...         ObjectCounter.num_instances += 1
...
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x10392d810>
```

```
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x1039810d0>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x10395b750>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x103959810>

>>> ObjectCounter.num_instances
4

>>> counter = ObjectCounter()
>>> counter.num_instances
5
```

`ObjectCounter` keeps a `.num_instances` class attribute that works as a counter of instances. When Python parses this class, it initializes the counter to zero. Creating instances of this class means automatically calling the `__init__()` method and incrementing `.num_instances` by one.

In the above example, you used the class name to access `.num_instances` inside `__init__()`. However, using the built-in `type()` function is best because it'll make your class more flexible:

```
>>> class ObjectCounter:
...     num_instances = 0
...     def __init__(self):
...         type(self).num_instances += 1
... 
```

The built-in `type()` returns the class or type of `self`, which is `ObjectCounter` in this example. This subtle change makes your class more robust and reliable by avoiding hard-coding the name of the class that provides the attribute.

You can *access* class attributes using either the class or one of its instances. That's why you can use the `counter` object to retrieve the value

of `.num_instances`. However, if you need to *modify* a class attribute, then you must use the class itself rather than one of its instances.

For example, if you use `self` to modify `.num_instances`, then you'll be overriding the original class attribute by creating a new instance attribute:

```
>>> class ObjectCounter:
...     num_instances = 0
...     def __init__(self):
...         self.num_instances += 1
...
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x103987550>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x1039c5890>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x10396a890>
>>> ObjectCounter()
<__main__.ObjectCounter object at 0x1036fa110>

>>> ObjectCounter.num_instances
0

>>> counter = ObjectCounter()
>>> counter.num_instances
1
```

You can't modify class attributes through instances of the containing class. Doing that will create new instance attributes with the same name as the original class attributes. That's why `ObjectCounter.num_instances` returns 0 in this example. You overrode the class attribute in the `.__init__()` method. However, `counter.num_instances` refers to the instance attribute, which has a value of 1.

Note

It's curious what Python does in the example above. It uses the initial value of `.num_instances` as a class attribute to create a new instance attribute with the same name and increments it by one. How come? Well, the line `self.num_instances += 1` is equivalent to the following assignment:

```
self.num_instances = self.num_instances + 1
```

In this statement, the attribute on the left of the equal sign is an instance attribute, while the attribute on the right side of the sign is a class attribute. Otherwise, the `+=` operator would've raised an `AttributeError`.

In general, you should use class attributes for sharing data between instances of a class. Any changes on a class attribute will be visible to all the instances of that class.

Instance Attributes

Instance attributes are variables tied to a particular object of a given class. The value of an instance attribute is attached to the object itself. So, the attribute's value is specific to its containing instance.

Python lets you dynamically attach attributes to existing objects that you've already created. However, you're more likely to define instance attributes inside **instance methods**, which are those methods that receive `self` as their first argument.

Even though you can define instance attributes inside any instance method, it's best to define all of them in the `__init__()` method, which is the instance initializer. This ensures that all the attributes have consistent values when you create a new instance. Additionally, it makes the code more organized and easier to debug.

Consider the following `Car` class, which defines several instance attributes:

```
class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.started = False
        self.speed = 0
        self.max_speed = 200
```

In this class, you define a total of seven instance attributes inside `__init__()`. The attributes `.make`, `.model`, `.year`, and `.color` take values from the arguments to `__init__()`, which are the arguments that you must pass to the class constructor `Car()` to create concrete objects.

Then, you explicitly initialize the attributes `.started`, `.speed`, and `.max_speed` with sensible values that don't come from the user.

Note

Inside a class, you must access all instance attributes through the `self` argument. This argument holds a reference to the **current instance**, which is where the attributes belong and live. The `self` argument plays a fundamental role in Python classes. You'll learn more about `self` in later chapters.

Here's how your `Car` class works in practice:

```
>>> from car import Car
>>> toyota_camry = Car("Toyota", "Camry", 2022, "Red")
>>> toyota_camry.make
'Toyota'
>>> toyota_camry.model
'Camry'
>>> toyota_camry.color
'Red'
```

```
>>> toyota_camry.speed
0
```

```
>>> ford_mustang = Car("Ford", "Mustang", 2022, "Black")
>>> ford_mustang.make
'Ford'
>>> ford_mustang.model
'Mustang'
>>> ford_mustang.year
2022
>>> ford_mustang.max_speed
200
```

In these examples, you create two different instances of `Car`. Each instance takes specific input arguments at instantiation time to initialize some of its attributes. Note how the values of the associated attributes are different and specific to the concrete instance.

Unlike class attributes, you can't access instance attributes through the class. You need to access them through their containing instance:

```
>>> Car.make
Traceback (most recent call last):
...
AttributeError: type object 'Car' has no attribute 'make'
```

Instance attributes are specific to a concrete instance of a given class. So, you can't access them through the class object. If you try to do that, you get an `AttributeError` exception.

The `__dict__` Attribute

In Python, both classes and instances have a special attribute called `__dict__`. This attribute holds a dictionary containing the writable members of the underlying class or instance. Remember, these members can be data attributes or methods. Each key in `__dict__` represents an attribute name. The value associated with a given key represents the value of the corresponding attribute.

In a class, `__dict__` will contain class attributes and methods. In an instance, `__dict__` will hold instance attributes.

When you access a class member through the class object, Python automatically searches for the member's name in the class `__dict__`. If the name isn't there, then you get an `AttributeError`.

Similarly, when you access an instance member through a concrete instance of a class, Python looks for the member's name in the instance `__dict__`. If the name doesn't appear there, then Python looks in the class `__dict__`. If the name isn't found, then you get an `AttributeError`.

Here's a toy class that illustrates how this mechanism works:

```
# sample_dict.py
class SampleClass:
    class_attr = 100

    def __init__(self, instance_attr):
        self.instance_attr = instance_attr

    def method(self):
        print(f"Class attribute: {self.class_attr}")
        print(f"Instance attribute: {self.instance_attr}")
```

In this class, you define a class attribute with a value of 100. In the `__init__()` method, you define an instance attribute that takes its value from the user's input. Finally, you define a method to print both attributes.

Now it's time to check the content of `__dict__` in the class object. Go ahead and run the following code:

```
>>> from sample_dict import SampleClass
>>> SampleClass.class_attr
100
```

```
>>> SampleClass.__dict__
mappingproxy({
    '__module__': '__main__',
    'class_attr': 100,
    '__init__': <function SampleClass.__init__ at 0x1036c62a0>,
    'method': <function SampleClass.method at 0x1036c56c0>,
    '__dict__': <attribute '__dict__' of 'SampleClass' objects>,
    '__weakref__': <attribute '__weakref__' of 'SampleClass' objects>,
    '__doc__': None
})

>>> SampleClass.__dict__["class_attr"]
100
```

Both the class attribute and the method are in the class `__dict__` dictionary. Note how you can use `__dict__` to access the value of class attributes by specifying the attribute's name in square brackets, in the same way you usually access keys in a dictionary.

Note

You can access the same dictionary by calling the built-in `vars()` function and pass your class or instance as an argument.

In instances, the `__dict__` dictionary will contain instance attributes only:

```
>>> instance = SampleClass("Hello!")
>>> instance.instance_attr
'Hello!'
>>> instance.method()
Class attribute: 100
Instance attribute: Hello!

>>> instance.__dict__
{'instance_attr': 'Hello!'}
```

```
>>> instance.__dict__["instance_attr"]
'Hello!'

>>> instance.__dict__["instance_attr"] = "Hello, Pythonista!"
>>> instance.instance_attr
'Hello, Pythonista!'
```

The instance `__dict__` dictionary in this example holds `.instance_attr` and its specific value for the object at hand. Again, you can access any existing instance attribute using `__dict__` and the attribute name in square brackets.

Dynamic Class and Instance Attributes

In Python, you can add new attributes to your classes and instances dynamically. This possibility allows you to attach new data and behavior to your classes and objects in response to changing requirements or contexts. It also allows you to adapt existing classes to specific and dynamic needs.

For example, you can take advantage of this Python feature when you don't know the required attributes of a given class at the time when you're defining that class itself.

Consider the following class, which aims to store a row of data from a database table or a [CSV](#) file:

```
>>> class Record:
...     """Hold a record of data."""
... 
```

In this class, you haven't defined any attributes or methods because you don't know what data the class will store. Fortunately, you can add attributes and even methods to this class dynamically.

For example, say that you've read a row of data from an `employees.csv` file using `csv.DictReader`. This class reads the data and returns it in a

dictionary-like object. Now suppose that you have the following dictionary of data:

```
>>> john = {  
...     "name": "John Doe",  
...     "position": "Python Developer",  
...     "department": "Engineering",  
...     "salary": 80000,  
...     "hire_date": "2020-01-01",  
...     "is_manager": False,  
... }
```

Next, you want to add this data to an instance of your `Record` class, and you need to represent each data field as an instance attribute. Here's how you can do it:

```
>>> john_record = Record()  
>>> for field, value in john.items():  
...     setattr(john_record, field, value)  
...  
>>> john_record.name  
'John Doe'  
>>> john_record.department  
'Engineering'  
  
>>> john_record.__dict__  
{  
    'name': 'John Doe',  
    'position': 'Python Developer',  
    'department': 'Engineering',  
    'salary': 80000,  
    'hire_date': '2020-01-01',  
    'is_manager': False  
}
```

In this code snippet, you first create an instance of `Record` called `john_record`. Then you run a for loop to [iterate over the items of](#)

your `dictionary` of data, `john`. Inside the loop, you use the built-in `setattr()` function to sequentially add each field as an attribute to your `john_record` object. If you inspect `john_record`, then you'll note that it stores all the original data as attributes.

You can also use the dot notation and an assignment to add new attributes to a class dynamically:

```
>>> class User:
...     pass
...
>>> # Add instance attributes dynamically
>>> jane = User()
>>> jane.name = "Jane Doe"
>>> jane.job = "Data Engineer"
>>> jane.__dict__
{'name': 'Jane Doe', 'job': 'Data Engineer'}
```

Here, you first create a minimal `User` class with no custom attributes. You use a `pass` statement as a placeholder to define the class's body.

Then you create an object called `jane`. Note how you can use dot notation and an assignment to add new attributes to the instance. In this example, you add `.name` and `.job` attributes with appropriate values.

Although you can construct an entire Python class dynamically, you must use this technique carefully because it can make your code difficult to understand and use.

2.2 Conclusion

You now have a better understanding of the types of attributes used to store data when using classes and objects. Class attributes belong to the class but can also be accessed by instances, whereas instance attributes are specific to each object of the class.

You've also seen how to define and use these attributes, following best practices to keep your code clear and maintainable. These tools will help you structure your objects more effectively and prepare you for working with behavior-based attributes, like methods, in the next chapter.

Chapter 3

Methods

In earlier chapters, you learned that Python classes allow you to bundle data and behavior together into a single entity. Data attributes define an object's state, while methods operate on that state to perform specific actions.

A **method** is a function defined inside a class, but its placement within the class explicitly ties it to the class or its instances. This relationship gives methods their power in object-oriented programming. They can take arguments and return values just like regular functions, but calling them requires the dot notation, linking them to a specific instance or to the class itself.

Python provides three distinct types of methods in classes:

1. **Instance methods** for interacting with individual objects
2. **Class methods** for managing class-wide behavior
3. **Static methods** for performing tasks unrelated to instance or class state

Understanding these methods and when to use them will help you write clearer, more structured and maintainable object-oriented code. In this chapter, you'll learn how to define and use each method type effectively.

3.1 Instance, Class, and Static Methods

Each type of method has a specific purpose and set of access restrictions. You've already used instance methods in previous chapters. You can begin with a basic example that showcases all three types of methods before looking at each one in detail:

```
# demo_class.py
class DemoClass:
    def instance_method(self):
        return "instance method called", self

    @classmethod
    def class_method(cls):
        return "class method called", cls

    @staticmethod
    def static_method():
        return "static method called"
```

Here's a closer look at these three types of methods.

Instance Methods

The first method in `DemoClass`, descriptively called `instance_method()`, is a regular *instance method*. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, `self`, which points to an instance of `DemoClass` when the method is called. However, instance methods can accept more than just one parameter, as you'll see later in this chapter.

Through the `self` parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.

Not only can they modify object state, but instance methods can also access the class itself through the `self.__class__` attribute. This means instance methods can also modify class state.

Class Methods

The second method, `DemoClass.class_method()`, is marked with a `@classmethod` decorator to flag it as a *class method*.

Instead of accepting a `self` parameter, class methods take a `cls` parameter that points to the class, not the object instance.

Note

The naming of the parameters `self` and `cls` is just a convention. You could just as easily name them `the_object` and `the_class` and get the same result. All that matters is that they're positioned first in the parameter list for the method. However, these are well-established conventions in Python. You should use these parameter names in classes you define.

Because the class method only has access to this `cls` argument, it can't modify object instance state. That would require access to `self`. However, class methods can still modify class state that applies across all instances of the class.

Static Methods

The third method, `DemoClass.static_method()`, is marked with a `@staticmethod` decorator to flag it as a *static method*.

This type of method takes neither a `self` nor a `cls` parameter, but it's free to accept an arbitrary number of other parameters.

Therefore, a static method can neither modify object state nor class state. Static methods are restricted in what data they can access. They're primarily a way to [namespace](#) your methods, since they're similar to regular functions but they're defined within a class.

3.2 Seeing Them in Action

DemoClass was set up in such a way that each method's implementation returns a tuple containing information to help you trace what's going on and which parts of the class or object the method can access.

Here's what happens when you call an **instance method**. This code assumes the file defining the class is named `demo_class.py`:

```
>>> from demo_class import DemoClass
>>> obj = DemoClass()
>>> obj.instance_method()
('instance method called',
 <demo_class.DemoClass object at 0x1049d5f10>)
```

This confirms that `instance_method()` has access to the object instance, which is printed as `<demo_class.DemoClass object>` through the `self` argument.

When you call the method, Python replaces the `self` argument with the instance object, `obj`. To confirm this point, you can pass the instance object *manually* to the class instead of using the dot notation:

```
>>> DemoClass.instance_method(obj)
('instance method called',
 <demo_class.DemoClass object at 0x1049d5f10>)
```

Instance methods can also access the *class itself* through the `self.__class__` attribute. This makes instance methods powerful in terms of access restrictions. They can modify state on the object instance *and* on the class itself.

Try out the **class method** next:

```
>>> obj.class_method()
('class method called', <class 'demo_class.DemoClass'>)
```

Calling `class_method()` shows that the method doesn't have access to the `DemoClass` instance. However, it can access the class, as shown by

`<class 'demo_class.DemoClass'>`, which represents the class itself. Everything in Python is an object, even classes themselves.

Notice how Python automatically passes the class as the first argument to the function when you call `DemoClass.class_method()`. Calling a class method in Python through the *dot syntax* triggers this behavior. The `self` parameter on instance methods works in the same way, since Python automatically passes the instance as the first argument when you call an instance method.

Time to call the **static method** now:

```
>>> obj.static_method()
'static method called'
```

Did you see how you called `static_method()` on the object and were able to do so successfully? Some developers are surprised when they learn that it's possible to call a static method on an object instance.

Behind the scenes, Python simply enforces the access restrictions by not passing in the `self` or the `cls` argument when you call a static method using the dot notation.

This confirms that static methods can neither access the object instance state nor the class state. They work like regular functions but belong to the class's namespace. They also belong to the namespace of each instance.

Now, look at what happens when you attempt to call these methods on the class itself without creating an object instance beforehand. You should start a new REPL session to ensure there's no instance of the class in memory:

```
>>> from demo_class import DemoClass
>>> DemoClass.class_method()
('class method called', <class 'demo_class.DemoClass'>)

>>> DemoClass.static_method()
'static method called'

>>> DemoClass.instance_method()
Traceback (most recent call last):
...
TypeError: DemoClass.instance_method() missing 1 required
positional argument: 'self'
```

You're able to call `class_method()` and `static_method()` just fine, but attempting to call `instance_method()` fails with a `TypeError`.

This is to be expected, since in this case you didn't create an object instance and tried calling an instance function directly on the class blueprint itself. This means there's no way for Python to populate the `self` argument. Therefore, the call fails.

3.3 The Pizza Class

This basic example shows the distinction between these three method types. However, you'll now explore this topic with a more realistic example for when to use these method types.

You can define a bare-bones `Pizza` class in `pizza.py`:

```
# pizza.py
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = list(ingredients)

    def __repr__(self):
        return f"Pizza({self.ingredients})"
```

The class contains the two special methods `__init__()` and `__repr__()`. You already used `__init__()` in previous examples in this book, but you'll learn more about this special method in later chapters. The `__repr__()` special method provides a string representation for when you need to display an object. You'll also learn about this special method and its close relative `__str__()` later in this book.

The class defines a data attribute called `.ingredients`. You can use any iterable containing the ingredients when you create an instance of the class. However, the `__init__()` method casts this into a list as the class definition will assume this attribute is a list.

You can explore this class in a REPL session:

```
>>> from pizza import Pizza
>>> Pizza(["cheese", "tomatoes"])
Pizza(['cheese', 'tomatoes'])
```

In the next three sections, you'll extend this class to include instance methods, class methods, and a static method.

Note

You'll define and update the class in a script named `pizza.py`. However, you'll explore this class in a REPL session. Each time you make changes to the class in `pizza.py`, you'll need to reload the class. However, you can't simply repeat the `import` statement in the same environment. Although there are other solutions, the simplest option is to start a new REPL session each time you make changes to `pizza.py`.

In the next section, you'll start by creating instance methods to deal with fussy customers in a pizza restaurant who want changes to the ingredients on their pizza.

3.4 Modifying Pizza Ingredients With Instance Methods

The first addition you'll make to the `Pizza` class is to add two instance methods. As you learned earlier, an instance method is a function that belongs in a class that has access to a specific instance of the class. Therefore, an instance method can change the data stored in the object. The parameter name `self` is used by convention to refer to the instance within an instance method.

You can add two instance methods called `.add_ingredient()` and `.remove_ingredient()` to the class:

```
# pizza.py
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = list(ingredients)

    def __repr__(self):
        return f"Pizza({self.ingredients})"

    def add_ingredient(self, ingredient):
        self.ingredients.append(ingredient)

    def remove_ingredient(self, ingredient):
        if ingredient in self.ingredients:
            self.ingredients.remove(ingredient)
```

You can now call these instance methods to change the state of the object:

```
>>> from pizza import Pizza
>>> a_pizza = Pizza(["cheese", "tomatoes"])
>>> a_pizza
Pizza(['cheese', 'tomatoes'])
```

```
>>> a_pizza.add_ingredient("onions")
>>> a_pizza
Pizza(['cheese', 'tomatoes', 'onions'])

>>> a_pizza.remove_ingredient("cheese")
>>> a_pizza
Pizza(['tomatoes', 'onions'])
```

The data contained within the attribute `.ingredients` changes when you call these instance methods. Therefore, a `Pizza` object contains the data about the ingredients and the ability to make changes to the data. This is the key principle of objects in object-oriented programming.

3.5 Delicious Pizza Factories With Class Methods

Everyone has their favorite pizza variation, so you can create more delicious pizzas:

```
>>> Pizza(["mozzarella", "tomatoes"])
Pizza(['mozzarella', 'tomatoes'])
>>> Pizza(["mozzarella", "tomatoes", "ham", "mushrooms"])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
>>> Pizza(["mozzarella"] * 4)
Pizza(['mozzarella', 'mozzarella', 'mozzarella', 'mozzarella'])
```

The Italians figured out their pizza taxonomy centuries ago, and so these delicious types of pizzas all have their own names. You'd do well to take advantage of that and give the users of your `Pizza` class a better interface for creating the pizza objects they crave.

A nice and clean way to do that is by using class methods as [factory functions](#) for the different kinds of pizzas you can create:

```
# pizza.py
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = list(ingredients)

    def __repr__(self):
        return f"Pizza({self.ingredients})"

    # instance methods
    # ...

    @classmethod
    def margherita(cls):
        return cls(["mozzarella", "tomatoes"])

    @classmethod
    def prosciutto(cls):
        return cls(["mozzarella", "tomatoes", "ham"])
```

The code above is truncated for display purposes. You should keep the instance methods you defined earlier in your code. Note how the `margherita` and `prosciutto` factory methods use the `cls` argument instead of calling the `Pizza` constructor directly.

This is a trick you can use to follow the [Don't Repeat Yourself \(DRY\)](#) principle. If you decide to rename this class at some point, you won't have to remember updating the constructor name in all of the class method factory functions.

Now, what can you do with these factory methods? It's time to try them out. You need to use a new REPL session since you made changes to `pizza.py`:

```
>>> from pizza import Pizza
>>> Pizza.margherita()
Pizza(['mozzarella', 'tomatoes'])
>>> Pizza.prosciutto()
Pizza(['mozzarella', 'tomatoes', 'ham'])
```

You can use the factory functions to create new `Pizza` objects that are configured the way you want them. They all use the same `__init__()` constructor internally and simply provide a shortcut for remembering all of the various ingredients.

Since these methods still create an instance of the class, you can also use other methods on the instances they create, such as the instance method `.add_ingredient()`:

```
>>> a_pizza = Pizza.prosciutto()
>>> a_pizza
Pizza(['mozzarella', 'tomatoes', 'ham'])

>>> a_pizza.add_ingredient("onions")
>>> a_pizza
Pizza(['mozzarella', 'tomatoes', 'ham', 'onions'])
```

Another way to look at this use of class methods is that they allow you to define alternative constructors for your classes.

Python only allows one `__init__()` method per class. It's possible to add as many alternative constructors as necessary by using class methods. This can make the interface for your classes self-documenting and simplify their usage.

3.6 When To Use Static Methods

You can stretch the pizza analogy thinner and thinner and add a static method to work out the pizza's area:

```
# pizza.py
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = list(ingredients)

    def __repr__(self):
        return f"Pizza({self.radius}, {self.ingredients})"

    # instance methods and class methods
    # ...

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

There are a few changes in this code:

- The `__init__()` special method has an extra parameter `radius`, which is used to create an instance attribute `.radius`.
- The `__repr__()` special method uses the new `.radius` attribute to display the pizza's radius.
- There's a new static method, `.circle_area()`, which calculates the area using the well-known circle area formula.
- There's also a new `.area()` instance method that returns the pizza's area. Instead of calculating the area directly within this method, it calls the static method `.circle_area()`.

You can try this new version of the `Pizza` class in a new REPL session:

```
>>> from pizza import Pizza
>>> a_pizza = Pizza(4, ["mozzarella", "tomatoes"])
>>> a_pizza
Pizza(4, ['mozzarella', 'tomatoes'])

>>> a_pizza.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

This is a simplistic example, but it helps explain some of the benefits that static methods provide.

As you've learned, static methods can't access class or instance state because they don't take a `cls` or `self` argument. That's a big limitation, but it's also a great signal to show that a particular method is independent from everything else around it.

In the above example, it's clear that `circle_area()` can't modify the class or the class instance in any way.

Now, why is that useful?

Flagging a method as a static method is a hint that a method won't modify class or instance state. This restriction is also enforced by the Python runtime. They enable you to communicate clearly about parts of your class architecture so that new development work is naturally guided to happen within these set boundaries.

Of course, it would be possible to defy these restrictions. But in practice, they often help avoid accidental modifications going against the original design.

Put differently, using static methods and class methods are ways to communicate developer intent while enforcing that intent enough to avoid most slip-of-the-mind mistakes and bugs that would break the design. Applied sparingly and when it makes sense, writing some of

your methods that way can provide maintenance benefits and make it less likely that other developers use your classes incorrectly.

Static methods also have benefits when it comes to writing test code since they're completely independent from the rest of the class. You don't have to worry about setting up a complete class instance before you can test the static method in a unit test. You can just fire away like you would testing a regular function. Again, this makes future maintenance easier.

3.7 Conclusion

In this chapter, you explored the three types of methods available in Python classes and how they serve different purposes:

- **Instance methods** interact with object-specific data and rely on `self`.
- **Class methods** provide class-wide functionality using `cls`, often serving as alternative constructors.
- **Static methods** are independent of instance or class state, performing general-purpose tasks.

You've also seen examples of how these methods are used, from manipulating object-specific data to creating flexible class-wide tools. Now you can choose the right method type for any scenario and use it effectively in your code.

Methods are at the core of object-oriented programming in Python, giving your classes the ability to perform meaningful actions. In the next chapter, you'll learn about **special methods**, which let you customize how your objects behave and interact with Python's built-in functionality. These methods will open the door to creating more Pythonic and intuitive classes.

Chapter 4

Special Methods

You’ve encountered methods that add functionality to objects in the previous chapter. However, Python also provides **special methods**, a unique category of methods that allow you to integrate your classes seamlessly into Python’s language features.

Special methods have special meanings. They define how your objects interact with core Python operations. Whether you’re creating objects, adding two numbers, comparing values, or iterating through data, Python often relies on special methods under the hood. Understanding these methods gives you the tools to customize your classes and use Python’s data model to its fullest extent.

A special method’s name starts and ends with a double underscore. For this reason, they’re often informally called **dunder methods**, where *dunder* is a shortened version of *double underscores*.

In this chapter, you’ll explore how to use special methods to make your classes more Pythonic. From customizing object representations to overloading operators, supporting iteration, and defining custom sequences or mappings, you’ll learn practical techniques to enhance your OOP skills and create classes that integrate naturally into Python’s ecosystem.

4.1 Getting to Know Python's Special Methods

Here's how the [Python documentation](#) defines the term *special methods*:

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores.

There's an important detail to highlight in this definition. *Python implicitly calls special methods to execute certain operations in your code*. For example, when you run the addition $5 + 2$, Python internally runs the following code under the hood:

```
>>> (5).__add__(2)
7
```

The `.__add__()` special method in the integer class supports the addition that you typically run as $5 + 2$.

Even though you can directly call special methods, they're not intended for direct use. You shouldn't call them directly in your code. Instead, you should rely on Python to call them automatically in response to a given operation.

Note

Special methods are sometimes also called *magic methods*. However, there's nothing magical about them. The only apparent magic around these methods is that Python calls them implicitly under the hood. As with all magic, once you know how the trick works, it's no longer mysterious. The official documentation refers to them as special methods, and the informal name dunder methods is also commonly used. It's best to stick to these two terms when referring to these methods.

Special methods exist for many purposes. All the available special methods support built-in features and play specific roles in the language. For example, built-in types such as [lists](#), [strings](#), and [dictionaries](#) implement most of their core functionality using special methods. In your custom classes, you can use special methods to make callable objects, define how objects are compared, tweak how you create objects, and more.

Note that because special methods have special meaning in Python, you should avoid naming custom methods using leading and trailing double underscores. Your custom method won't trigger any Python action if its name doesn't match any official special method names, but it'll certainly confuse other programmers. New dunder names may also be introduced in future Python versions.

Special methods are core to Python's [data model](#) and are a fundamental part of object-oriented programming in Python. The most common special method is one you've already used: `__init__()`. This dunder method is called when you create a new instance of a class. You'll learn more about this special method in the next chapter, which focuses on object creation and initialization. In this chapter, you'll learn about some of the other most commonly used special methods. They'll help you write better object-oriented code in your day-to-day programming.

4.2 Representing Objects as Strings

A common task that you'll perform in your Python code is to display data or produce output. In some cases, your programs may need to display useful information to the user. In other cases, you may need to show information to other programmers using your code.

These two types of output can be pretty different because their respective target audiences will have different needs. The information for the user may not need to highlight implementation details. It may just need to be user-friendly and clear.

In contrast, the information for an audience of programmers may need to be technically sound, providing details of your code's implementation.

Python has you covered in both scenarios. If you want to provide user-friendly output, then you can use the `__str__()` dunder method. On the other hand, when you need to provide developer-friendly output, then you can use the `__repr__()` method. These special methods support two different string representations for Python objects.

Along with `__init__()`, the `__str__()` and `__repr__()` methods are arguably the most common special methods in custom classes. In the following sections, you'll learn about these two useful methods.

Using `__repr__()` and `__str__()` in a Class

You can explore string representation with user-defined classes by defining a class named `Book`. You can use a script named `book.py` to define this class:

```
# book.py
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

odyssey = Book("The Odyssey", "Homer")

print(odyssey)
```

You define the class `Book` with an `__init__()` method. There are two required arguments when initializing the class, `title` and `author`.

You create an instance of this class with the title and author of the book *The Odyssey*, and you pass `odyssey` to the `print()` function. You use `print()` because when you evaluate a line that only contains the variable name in a script, no output is printed. When you run this script, you get the following output:

```
$ python book.py
<__main__.Book object at 0x1025c4ed0>
```

This output is the default string representation of an object that's inherited from the `object` class. The `object` class is the base class for all Python classes. It shows:

- **`<__main__.Book>`**: The name of the class and where it's defined
- **`0x1025c4ed0`**: The memory address of the object

This default representation tells you the name of the class and the memory address, which is shown as a [hexadecimal value](#). In CPython, the memory address is the same as the object's identity. You can find an object's identity using the [built-in function `id\(\)`](#), which returns an integer rather than a hex value.

However, the memory address is rarely useful. This default representation doesn't provide any additional information about the object that would be useful for the user or the programmer.

You can use the built-in functions `repr()` and `str()` to view the two string representations. However, in this case, they both return the same default string representation by default:

```
# book.py
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

odyssey = Book("The Odyssey", "Homer")

print(repr(odyssey))
print(str(odyssey))
```

You call `repr(odyssey)` and `str(odyssey)` and print out the values they return. The output shows the same default representation for both:

```
$ python book.py
<__main__.Book object at 0x100d046d0>
<__main__.Book object at 0x100d046d0>
```

This output confirms that only the default representation is available at the moment.

You can define the `__repr__()` special method for this class:

```
# book.py
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __repr__(self):
        class_name = type(self).__name__
        return (
            f'{class_name}('
```

```
        f"title={self.title!r}, "  
        f"author={self.author!r})"  
    )  
  
odyssey = Book("The Odyssey", "Homer")  
  
print(repr(odyssey))  
print(str(odyssey))
```

The `__repr__()` method has no additional parameters other than `self`. This special method, like most special methods, is an instance method. Also, it must always return a string. Note that although you can hard-code the name of the class in the returned string, it's best to obtain this dynamically, such as by using `type(self).__name__`, to make the class more robust and future-compatible.

The f-string includes the `!r` conversion flag, which ensures that Python uses the `__repr__()` representation for the objects within the curly braces.

The output from this code shows that both `repr(odyssey)` and `str(odyssey)` return the string representation that you just defined:

```
$ python book.py  
Book(title='The Odyssey', author='Homer')  
Book(title='The Odyssey', author='Homer')
```

The `repr()` built-in function calls the object's `__repr__()` method. If a class doesn't have a `__str__()` method defined, then `str()` will also default to the object's `__repr__()` method.

You define the return value for `__repr__()` to include the name of the class followed by parentheses `()` and the two arguments required to initialize the class. This format is ideal for the official string representation since it represents a valid Python expression that can re-create an object equal to the original one. Whenever possible, you should use this format for the official string representation.

When to Use `.__repr__()` vs `.__str__()`

Next, you can define the `.__str__()` method for the class:

```
# book.py
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __repr__(self):
        class_name = type(self).__name__
        return (
            f"{class_name}("
            f"title={self.title!r}, "
            f"author={self.author!r})"
        )

    def __str__(self):
        return self.title

odyssey = Book("The Odyssey", "Homer")

print(repr(odyssey))
print(str(odyssey))
```

You define the `.__str__()` special method, which shouldn't have any additional parameters other than `self` and should return a string. The string representation can be any string that you feel would be most beneficial for the program's user. The output from this version of the code shows both the official and informal string representations, in that order:

```
$ python book.py
Book(title='The Odyssey', author='Homer')
The Odyssey
```

The informal string representation returned by `__str__()` only shows the book's title and doesn't have a reference to the class name. You can customize the informal string representation to suit your needs:

```
# book.py
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __repr__(self):
        class_name = type(self).__name__
        return (
            f'{class_name}('
            f'title={self.title!r}, '
            f'author={self.author!r})'
        )

    def __str__(self):
        return f'{self.title}' by {self.author}'

odyssey = Book("The Odyssey", "Homer")

print(repr(odyssey))
print(str(odyssey))
```

The string returned by `__str__()` now has the book's title in quotation marks followed by the author's name. Note that if you want to use double quotation marks within a string, you can use single quotation marks to denote the string. The output from this code shows the official representation first, followed by the modified informal string representation:

```
$ python book.py
Book(title='The Odyssey', author='Homer')
"The Odyssey" by Homer
```

The official string representation includes all the details needed by a programmer, and it allows you to replicate the object to explore it further. This has a couple of major benefits:

- It makes the program more **maintainable**.
- It facilitates **debugging**.

You can use the expression in this string representation to create an object with the same values as the original one. This feature is useful during debugging as you or a fellow programmer replicates an object to explore it further.

However, the informal string representation has a friendlier format that's better suited for a program's user. This representation makes the output more readable for a user.

Take extra precautions when implementing these two special methods to ensure that sensitive information, such as user passwords, doesn't get exposed in any way. It's best to skip such attributes or at least anonymize them before returning the corresponding string.

When you define a class, it's a best practice to include the official string representation by defining `__repr__()`. By including this method, you avoid the default representation, which isn't very useful in most cases. This method will also provide a fallback option for the informal string representation, which comes in handy when you can use the same representation for both use cases.

If you need to display the object to a user, then you can also define `__str__()`. This method will provide an output that's easier for the user to understand.

If you're using Python's [data classes](#), which you'll learn about later in this book, a default official string representation is already included. You don't need to define `__repr__()` yourself unless you want to override the default format.

You're now ready to include string representations in any class that you define.

4.3 Supporting Operators in Custom Classes

Python has multiple types of **operators**, which are special symbols, combinations of symbols, or keywords that designate some type of computation. Internally, Python supports operators with special methods. For example, the `.__add__()` special method supports the plus operator (+) as you already saw.

In the following sections, you'll learn about the specific special methods that support Python operators, including arithmetic, comparison, membership, and augmented operators. To kick things off, you'll start with arithmetic operators, which are arguably the most commonly used operators.

Arithmetic Operators

Arithmetic operators are those that you use to perform *arithmetic operations* on numeric values. In most cases, they come from math, and therefore, Python represents them with the usual math signs.

In the following table, you'll find a list of the arithmetic operators in Python and their supporting special methods:

Operator	Supporting Method
+	<code>.__add__(self, other)</code>
-	<code>.__sub__(self, other)</code>
*	<code>.__mul__(self, other)</code>
/	<code>.__truediv__(self, other)</code>
//	<code>.__floordiv__(self, other)</code>
%	<code>.__mod__(self, other)</code>
**	<code>.__pow__(self, other[, modulo])</code>

Note that all these methods take a second argument, which is called *other* by convention. In most cases, this argument should be the same type as `self` or a compatible type. If that's not the case, then you can get an error.

To illustrate how you can overload some of these operators, create a `Distance` class to deal with distances using different units. You can name the script `distance.py`:

```
# distance.py
class Distance:
    _multiples = {
        "mm": 0.001,
        "cm": 0.01,
        "m": 1,
        "km": 1_000,
    }

    def __init__(self, value, unit="m"):
        self.value = value
        self.unit = unit.lower()
```

In `Distance`, you have a class attribute called `._multiples`. This attribute contains a dictionary of length units and their corresponding conversion factors. The leading underscore in `._multiples` is a Python convention that indicates that this attribute is not meant for access by a user of the class but is only used within the class definition.

In the `.__init__()` method, you initialize the `.value` and `.unit` attributes according to the user's definition. Note that you've used the `str.lower()` method to enforce the use of lowercase letters within the class.

Try to create two distances and add them in a REPL session:

```
>>> from distance import Distance
>>> distance_1 = Distance(200, "m")
>>> distance_2 = Distance(1, "km")
>>> total = distance_1 + distance_2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +:
      'Distance' and 'Distance'
```

Python doesn't know how to add two instances of the `Distance` class. When you use the `+` operator, the program looks for the `__add__()` dunder method in `Distance` but doesn't find it. It raises a `TypeError` stating that the operand is not supported.

You can include this special method and a few more methods to complete this class:

```
# distance.py
class Distance:
    _multiples = {
        "mm": 0.001,
        "cm": 0.01,
        "m": 1,
        "km": 1_000,
    }

    def __init__(self, value, unit="m"):
        self.value = value
        self.unit = unit.lower()

    def to_meter(self):
        return self.value * type(self)._multiples[self.unit]

    def __add__(self, other):
        return self._compute(other, "+")
```

```
def __sub__(self, other):
    return self._compute(other, "-")

def _compute(self, other, operator):
    expr = f"{self.to_meter()} {operator} {other.to_meter()}"
    operation = eval(expr)
    cls = type(self)
    return cls(operation / cls._multiples[self.unit], self.unit)
```

You define `.to_meter()`, and you use `.multiples` to express the current distance in meters. You'll use this method as a helper in the `.__add__()` and `.__sub__()` methods. This way, your class will be able to add or subtract distances of different units correctly.

The `.__add__()` and `.__sub__()` methods support the addition and subtraction operators. As with all the special methods associated with arithmetic operators, they also include a second parameter named `other`. In both methods, you use the `._compute()` helper method to run the computation. Note that `._compute()` also has a leading underscore in its name, showing it's only meant to be used within the class definition.

In `._compute()`, you take `other` and `operator` as arguments. Then, you use the built-in `eval()` function to evaluate the expression and obtain the intended result for the current operation. Next, you get the class name using the built-in `type()` function. Finally, you create and return an instance of `Distance` with the computed value converted into the unit of the current instance.

The built-in `eval()` function implies some security risks since a user can input anything and your code will execute it. It's fine to use `eval()` in an example such as this, but normally you should avoid it if your code has untrusted inputs.

You can try out your `Distance` class again in a new REPL session:

```
>>> from distance import Distance
>>> distance_1 = Distance(200, "m")
>>> distance_2 = Distance(1, "km")
>>> total = distance_1 + distance_2
>>> total.value
1200.0
>>> total.unit
'm'

>>> displacement = distance_2 - distance_1
>>> displacement.value
0.8
>>> displacement.unit
'km'
```

Your class works as expected. It supports the addition and subtraction operators since it includes the `.__add__()` and `.__sub__()` special methods.

More on Arithmetic Operators

The special methods that support operators are affected by the relative position of each object in the containing [expression](#). That's why `displacement` in the above section is in kilometers, while `total` is in meters.

Take `.__add__()`, for example. Python calls this method on the left-hand operand. It's possible to have operands of different data types when using operators. If the left-hand operand doesn't implement the method, then the operation will fail. If this operand implements the method, but its implementation doesn't behave as you want with the right-hand operand, then you can face issues. Thankfully, Python has the right-hand version of operator methods, such as `.__radd__()`, to tackle these issues. The other arithmetic operators also have special methods with an extra leading `r` in their name. Here's how these work.

For example, say that you want to write a `Number` class that supports addition between its object and integer or floating-point numbers. In that situation, you can do something like the following:

```
# number.py
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        print("__add__ called")
        if isinstance(other, Number):
            return Number(self.value + other.value)
        elif isinstance(other, int | float):
            return Number(self.value + other)
        else:
            raise TypeError("unsupported operand type for +")

    def __radd__(self, other):
        print("__radd__ called")
        return self.__add__(other)
```

The `__add__()` method works when you use an instance of `Number` as the left-hand operator in an addition expression. In contrast, `__radd__()` works when you use an instance of `Number` as the right-hand operand. Note that the `r` at the beginning of the method's name stands for *right*.

Here's how your `Number` class works:

```
>>> from number import Number
>>> five = Number(5)
>>> ten = Number(10)
>>> fifteen = five + ten
__add__ called
>>> fifteen.value
15
```

```
>>> six = five + 1
__add__ called
>>> six.value
6

>>> twelve = 2 + ten
__radd__ called
__add__ called
>>> twelve.value
12
```

In this code snippet, you create two instances of `Number`. Then you use them in an addition where Python calls `__add__()` as expected. Next up, you use `five` as the left-hand operator in an expression that mixes your class with the `int` type. In this case, Python calls `Number.__add__()` again.

Finally, you use `ten` as the right-hand operand in an addition. The left-hand operand is an integer, but the `__add__()` special method in the integer class doesn't support adding an instance of your `Number` class. Therefore, Python implicitly calls `Number.__radd__()` instead, which falls back to calling `Number.__add__()`.

Here's a summary of the `__r*__()` methods:

Operator	Right-Hand Method
+	<code>__radd__(self, other)</code>
-	<code>__rsub__(self, other)</code>
*	<code>__rmul__(self, other)</code>
/	<code>__rtruediv__(self, other)</code>
//	<code>__rfloordiv__(self, other)</code>
%	<code>__rmod__(self, other)</code>
**	<code>__rpow__(self, other[, modulo])</code>

Python calls these methods on a right-hand operand when the left-hand operand doesn't support the corresponding operation and the operands are of different types.

Python also has some unary operators. They're called unary because they operate on a single operand:

Operator	Supporting Method	Description
-	<code>.__neg__(self)</code>	Returns the target value with the opposite sign
+	<code>.__pos__(self)</code>	Provides a complement to the negation without performing any transformation

You can also overload these operators in Python's built-in numeric types and support them in your own classes.

Comparison Operator Methods

You'll also find that special methods are behind [comparison operators](#). For example, when you execute `5 < 2`, Python calls the `.__lt__()` special method. The letters in the special method name stand for *less than*. Here's a summary of all the comparison operators and their supporting methods:

Operator	Supporting Method
<	<code>.__lt__(self, other)</code>
<=	<code>.__le__(self, other)</code>
==	<code>.__eq__(self, other)</code>
!=	<code>.__ne__(self, other)</code>
>=	<code>.__ge__(self, other)</code>
>	<code>.__gt__(self, other)</code>

To show how you can use some of these methods in your custom classes, say that you're writing a `Rectangle` class to create multiple different rectangles. You want to be able to compare these rectangles.

Here's a `Rectangle` class that supports the equality (`==`), less than (`<`), and greater than (`>`) operators:

```
# rectangle.py
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

    def area(self):
        return self.height * self.width

    def __eq__(self, other):
        return self.area() == other.area()

    def __lt__(self, other):
        return self.area() < other.area()

    def __gt__(self, other):
        return self.area() > other.area()
```

Your class has an `.area()` method that computes the rectangle's area using its height and width. Then you have the three required special methods to support the intended comparison operators. Note that all of them use the rectangle's area to determine the final result.

Here's how your class works in practice:

```
>>> from rectangle import Rectangle
>>> basketball_court = Rectangle(15, 28)
>>> soccer_field = Rectangle(75, 110)
>>> basketball_court < soccer_field
True
```

```
>>> basketball_court > soccer_field
False
>>> basketball_court == soccer_field
False
```

That's great! Your `Rectangle` class now supports equality (`==`), less than (`<`), and greater than (`>`) operations. As an exercise, you can go ahead and implement the rest of the operators using the appropriate special method.

Membership Operators

Python has two operators that allow you to determine whether a given value is in a collection of values. The operators are `in` and `not in`. They support a check that's known as a [membership test](#).

For example, say that you want to determine whether a number appears in a list of numbers. You can use the `in` operator:

```
>>> 2 in [2, 3, 5, 9, 7]
True

>>> 10 in [2, 3, 5, 9, 7]
False
```

In the first example, the number 2 *is* in the list of numbers, so you get `True` as a result. In the second example, the number 10 is not in the list, and you get `False`. The `not in` operator works similarly to `in` but as a negation. With this operator, you find out if a given value *is not* in a collection.

The special method that supports the membership operators `in` and `not in` is `__contains__()`. By implementing this method in a custom class, you can control how its instances will behave in a membership test.

The `__contains__()` method must take an argument that represents the value that you want to check for.

To illustrate how to support membership tests in a custom class, say that you want to implement a basic [stack](#) data structure that supports the *push* and *pop* operations as well as the `in` and `not in` operators.

Here's the code for your `Stack` class:

```
# stack.py
class Stack:
    def __init__(self, items=None):
        self.items = list(items) if items is not None else []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def __contains__(self, item):
        for current_item in self.items:
            if item == current_item:
                return True
        return False
```

In this example, you define `Stack` and provide the `.push()` and `.pop()` methods. The former method appends a new item to the top of the stack, while the latter removes and returns the item at the top of the stack.

Then, you implement the `__contains__()` method, which takes the target item as an argument and uses a `for` loop to determine whether the item is in the list that stores the stack's data.

You can make the above implementation of `__contains__()` more concise by taking advantage of the `in` operator itself. That's possible because Python lists support membership tests by default.

For example, you can update `__contains__()` as follows:

```
# stack.py
class Stack:
    # ...

    def __contains__(self, item):
        return item in self.items
```

In this version of `__contains__()`, you use the `in` operator instead of a `for` loop. You're directly checking for membership without explicitly iterating over the data.

Here's how you can use an instance of `Stack` in a membership test:

```
>>> from stack import Stack
>>> stack = Stack([2, 3, 5, 9, 7])
>>> 2 in stack
True
>>> 10 in stack
False

>>> 2 not in stack
False
>>> 10 not in stack
True
```

By implementing the `__contains__()` method in your custom classes, you can customize how objects of those classes respond to membership checks using the `in` and `not in` operators.

Augmented Assignments

When it comes to arithmetic operations, you'll find a common expression that uses the current value of a variable to update the variable itself. A classic example of this operation is when you need to update a counter or an accumulator:

```
>>> counter = 0
>>> counter = counter + 1
>>> counter = counter + 1
>>> counter
2
```

The second and third lines in this code snippet update the counter's value using the previous value. This type of operation is so common in programming that Python has a shortcut for it, known as **augmented assignment operators**.

For example, you can shorten the above code using the augmented assignment operator for addition:

```
>>> counter = 0
>>> counter += 1
>>> counter += 1
>>> counter
2
```

This code looks more concise and it's more Pythonic. This operator (+=) isn't the only one that Python supports. Here's a summary of the supported operators and the special methods used to define them:

Operator	Supporting Method
+=	<code>.__iadd__(self, other)</code>
-=	<code>.__isub__(self, other)</code>
*=	<code>.__imul__(self, other)</code>
/=	<code>.__itruediv__(self, other)</code>
//=	<code>.__ifloordiv__(self, other)</code>
%=	<code>.__imod__(self, other)</code>
**=	<code>.__ipow__(self, other[, modulo])</code>

When you use these methods in your custom classes, you should attempt to do the underlying operation **in place**, which is possible when you're working with **mutable** types. If the data type is immutable, then

This is a sample from “Modern Object-Oriented Python: Write Clean, Pythonic Code with OOP”

Object-oriented programming (OOP) is one of the most powerful tools in Python, but knowing how to use it *well* is what separates clean, maintainable code from a tangled mess.

This book takes you from the core building blocks of OOP all the way to the design principles that professionals rely on to keep large code-bases healthy. At the end of the book you’ll be able to:

- Model real-world entities with classes, data attributes, and methods, and customize object behavior with Python’s special methods
- Reuse and organize code with inheritance and composition, manage attribute access cleanly with properties, and cut boilerplate using data classes
- Apply the SOLID principles to design robust, maintainable object-oriented code that’s a pleasure to read, extend, and maintain

If you enjoyed the sample chapters you can purchase a full version of the book at realpython.com/modern-oop
