EXPERT INSIGHT

python™

Python Object-Oriented Programming

Build robust and maintainable object-oriented Python applications and libraries

Fourth Edition

Steven F. Lott
Dusty Phillips

Packt

Python Object-Oriented Programming

Fourth Edition

Build robust and maintainable object-oriented Python applications and libraries

Steven F. Lott

Dusty Phillips

BIRMINGHAM—MUMBAI

Python Object-Oriented Programming
Fourth Edition

Producer: Dr. Shailesh Jain
Acquisition Editor – Peer Reviews: Saby D'silva
Project Editor: Parvathy Nair
Content Development Editor: Lucy Wan
Copy Editor: Safis Editor
Technical Editor: Aditya Sawant
Proofreader: Safis Editor
Indexer: Tejal Daruwale Soni
Presentation Designer: Pranit Padwal

First published: July 2010
Second edition: August 2015
Third edition: October 2018
Fourth edition: June 2021

Production reference: 2281221

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80107-726-2

www.packt.com

Contributors

About the authors

Steven Lott has been programming since computers were large, expensive, and rare. Working for decades in high tech has given him exposure to a lot of ideas and techniques—some bad, but most are useful and helpful to others.

Steven has been working with Python since the '90s, building a variety of tools and applications. He's written a number of titles for Packt Publishing, including *Mastering Object-Oriented*, *Modern Python Cookbook*, and *Functional Python Programming*.

He's a technomad, and lives on a boat that's usually located on the east coast of the US. He tries to live by the words "Don't come home until you have a story."

Dusty Phillips is a Canadian author and software developer. His storied career has included roles with the world's biggest government, the world's biggest social network, a two person startup, and everything in between. In addition to *Python Object-Oriented Programming*, Dusty wrote *Creating Apps In Kivy* (O'Reilly) and is now focused on writing fiction.

Thank you to Steven Lott, for finishing what I started, to all my readers for appreciating what I write, and to my wife, Jen Phillips, for everything else.

About the reviewer

Bernát Gábor, originally from Transylvania, works as a senior software engineer at Bloomberg in London, UK. During his day job, he primarily focuses on improving the quality of the data ingestion pipeline at Bloomberg with predominant use of the Python programming language and paradigms. He's been working with Python for over ten years and is a major open-source contributor in this domain, with a particular focus on the packaging area. He's also the author and maintainer of high-profile projects such as `virtualenv`, `build`, `and tox`. For more information, see `https://bernat.tech/about`.

I would like to express my very great appreciation to Lisa, my fiancée, for her invaluable support on a daily basis. Love you!

Table of Contents

Preface

The Python programming language is extremely popular and used for a variety of applications. The Python language is designed to make it relatively easy to create small programs. To create more sophisticated software, we need to acquire a number of important programming and software design skills.

This book describes the **object-oriented** approach to creating programs in Python. It introduces the terminology of object-oriented programming, demonstrating software design and Python programming through step-by-step examples. It describes how to make use of inheritance and composition to build software from individual elements. It shows how to use Python's built-in exceptions and data structures, as well as elements of the Python standard library. A number of common design patterns are described with detailed examples.

This book covers how to write automated tests to confirm that our software works. It also shows how to use the various concurrency libraries available as part of Python; this lets us write software that can make use of multiple cores and multiple processors in a modern computer. An extended case study covers a simple machine learning example, showing a number of alternative solutions to a moderately complicated problem.

Who this book is for

This book targets people who are new to object-oriented programming in Python. It assumes basic Python skills. For readers with a background in another object-oriented programming language, this book will expose many distinctive features of Python's approach.

Because of Python's use for data science and data analytics, this book touches on the related math and statistics concepts. Some knowledge in these areas can help to make the applications of the concepts more concrete.

What this book covers

This book is divided into four overall sections. The first six chapters provide the core principles and concepts of object-oriented programming and how these are implemented in Python. The next three chapters take a close look at Python built-in features through the lens of object-oriented programming. *Chapters 10*, *11,* and *12* look at a number of common design patterns and how these can be handled in Python. The final section covers two additional topics: testing and concurrency.

Chapter 1, *Object-Oriented Design*, introduces the core concepts underlying object-oriented design. This provides a road map through the ideas of state and behavior, attributes and methods, and how objects are grouped into classes. This chapter also looks an encapsulation, inheritance, and composition. The case study for this chapter introduces the machine learning problem, which is an implementation of a *k*-nearest neighbors (*k*-NN) classifier.

Chapter 2, *Objects in Python*, shows how class definitions work in Python. This will include the type annotations, called type hints, class definitions, modules, and packages. We'll talk about practical considerations for class definition and encapsulation. The case study will begin to implement some of the classes for the *k*-NN classifier.

Chapter 3, *When Objects Are Alike*, addresses how classes are related to each other. This will include how to make use of inheritance and multiple inheritance. We'll look at the concept of polymorphism among the classes in a class hierarchy. The case study will look at alternative designs for the distance computations used to find the nearest neighbors.

Chapter 4, *Expecting the Unexpected*, looks closely at Python's exceptions and exception handling. We'll look at the built-in exception hierarchy. We'll also look at how unique exceptions can be defined to reflect a unique problem domain or application. In the case study, we'll apply exceptions to data validation.

Chapter 5, *When to Use Object-Oriented Programming*, dives more deeply into design techniques. This chapter will look at how attributes can be implemented via Python's properties. We'll also look at the general concept of a **manager** for working with collections of objects. The case study will apply these ideas to expand on the *k*-NN classifier implementation.

Chapter 6, *Abstract Base Classes and Operator Overloading*, is a deep dive into the idea of abstraction, and how Python supports abstract base classes. This will involve comparing **duck typing** with more formal methods of `Protocol` definition. It will include the techniques for overloading Python's built-in operators. It will also look at metaclasses and how these can be used to modify class construction. The case study will redefine some of the existing classes to show how abstraction must be used carefully to lead to simplification of a design.

Chapter 7, *Python Data Structures*, examines a number of Python's built-in collections. This chapter examines tuples, dictionaries, lists, and sets. It also looks at how dataclasses and named tuples can simplify a design by providing a number of common features of a class. The case study will revise some earlier class definitions to make use of these new techniques.

Chapter 8, *The Intersection of Object-Oriented and Functional Programming*, looks at Python constructs that aren't simply class definitions. While all of Python is object-oriented, function definitions allow us to create callable objects without the clutter of a class definition. We'll also look at Python's context manager construct and the `with` statement. In the case study, we'll look at alternative designs that avoid some class clutter.

Chapter 9, *Strings, Serialization, and File Paths*, covers the way objects are serialized as strings and how strings can be parsed to create objects. We'll look at several physical formats, including Pickle, JSON, and CSV. The case study will revisit how sample data is loaded and processed by the *k*-NN classifier.

Chapter 10, *The Iterator Pattern*, describes the ubiquitous concept of iteration in Python. All of the built-in collections are iterable, and this design pattern is central to a great deal of how Python works. We'll look at Python comprehensions and generator functions, also. The case study will revisit some earlier designs using generator expressions and list comprehensions to partition samples for testing and training.

Chapter 11, *Common Design Patterns*, looks at some common object-oriented design. This will include the Decorator, Observer, Strategy, Command, State, and Singleton design patterns.

Chapter 12, *Advanced Design Patterns*, looks at some more advanced object-oriented design. This will include the Adapter, Façade, Flyweight, Abstract Factory, Composite, and Template patterns.

Chapter 13, *Testing Object-Oriented Programs*, shows how to use `unittest` and `pytest` to provide an automated unit test suite for a Python application. This will look at some more advanced testing techniques, like using mock objects to isolate the unit under test. The case study will show how to create test cases for the distance computations covered in *Chapter 3*.

Chapter 14, *Concurrency*, looks at how we can make use of multi-core and multi-processor computer systems to do computations rapidly and write software that is responsive to external events. We'll look at threads and multiprocessing, as well as Python's `asyncio` module. The case study will show how to use these techniques to do hyperparameter tuning on the *k*-NN model.

To get the most out of this book

All of the examples were tested with Python 3.9.5. The ***mypy*** tool, version 0.812, was used to confirm that the type hints were consistent.

Some of the examples depend on an internet connection to gather data. These interactions with websites generally involve small downloads.

Some of the examples involve packages that are not part of Python's built-in standard library. In the relevant chapters, we note the packages and provide the install instructions. All of these extra packages are in the Python Package Index, at `https://pypi.org`.

Download the example code files

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Python-Object-Oriented-Programming---4th-edition`.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781801077262_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "You can confirm Python is running by importing the `antigravity` module at the `>>>` prompt."

A block of code is set as follows:

```
class Fizz:
    def member(self, v: int) -> bool:
        return v % 5 == 0
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
class Fizz:
    def member(self, v: int) -> bool:
        return v % 5 == 0
```

Any command-line input or output is written as follows:

```
python -m pip install tox
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes. For example: "Formally, an object is a collection of **data** and associated **behaviors**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share Your Thoughts

Once you've read *Python Object-Oriented Programming, Fourth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Object-Oriented Design

In software development, design is often considered as the step that's done *before* programming. This isn't true; in reality, analysis, programming, and design tend to overlap, combine, and interweave. Throughout this book, we'll be covering a mixture of design and programming issues without trying to parse them into separate buckets. One of the advantages of a language like Python is the ability to express the design clearly.

In this chapter, we will talk a little about how we can move from a good idea toward writing software. We'll create some design artifacts – like diagrams – that can help clarify our thinking before we start writing code. We'll cover the following topics:

- What object-oriented means
- The difference between object-oriented design and object-oriented programming
- The basic principles of object-oriented design
- Basic **Unified Modeling Language** (**UML**) and when it isn't evil

We will also introduce this book's object-oriented design case study, using the "4+1" architectural view model. We'll touch on a number of topics here:

- An overview of a classic machine learning application, the famous Iris classification problem
- The general processing context for this classifier
- Sketching out two views of the class hierarchy that look like they'll be adequate to solve the problem

Introducing object-oriented

Everyone knows what an object is: a tangible thing that we can sense, feel, and manipulate. The earliest objects we interact with are typically baby toys. Wooden blocks, plastic shapes, and over-sized puzzle pieces are common first objects. Babies learn quickly that certain objects do certain things: bells ring, buttons are pressed, and levers are pulled.

The definition of an object in software development is not terribly different. Software objects may not be tangible things that you can pick up, sense, or feel, but they are models of something that can do certain things and have certain things done to them. Formally, an object is a collection of **data** and associated **behaviors**.

Considering what an object is, what does it mean to be object-oriented? In the dictionary, *oriented* means *directed toward*. Object-oriented programming means writing code directed toward modeling objects. This is one of many techniques used for describing the actions of complex systems. It is defined by describing a collection of interacting objects via their data and behavior.

If you've read any hype, you've probably come across the terms *object-oriented analysis*, *object-oriented design*, *object-oriented analysis and design*, and *object-oriented programming*. These are all related concepts under the general *object-oriented* umbrella.

In fact, analysis, design, and programming are all stages of software development. Calling them object-oriented simply specifies what kind of software development is being pursued.

Object-oriented analysis (**OOA**) is the process of looking at a problem, system, or task (that somebody wants to turn into a working software application) and identifying the objects and interactions between those objects. The analysis stage is all about *what* needs to be done.

The output of the analysis stage is a description of the system, often in the form of *requirements*. If we were to complete the analysis stage in one step, we would have turned a task, such as *As a botanist, I need a website to help users classify plants so I can help with correct identification*, into a set of required features. As an example, here are some requirements as to what a website visitor might need to do. Each item is an action bound to an object; we've written them with *italics* to highlight the actions, and **bold** to highlight the objects:

- *Browse* **Previous Uploads**
- *Upload new* **Known Examples**

- *Test* for **Quality**
- *Browse* **Products**
- *See* **Recommendations**

In some ways, the term *analysis* is a misnomer. The baby we discussed earlier doesn't analyze the blocks and puzzle pieces. Instead, she explores her environment, manipulates shapes, and sees where they might fit. A better turn of phrase might be *object-oriented exploration*. In software development, the initial stages of analysis include interviewing customers, studying their processes, and eliminating possibilities.

Object-oriented design (**OOD**) is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors on other objects. The design stage is all about transforming *what* should be done into *how* it should be done.

The output of the design stage is an implementation specification. If we were to complete the design stage in a single step, we would have turned the requirements defined during object-oriented analysis into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language.

Object-oriented programming (**OOP**) is the process of converting a design into a working program that does what the product owner originally requested.

Yeah, right! It would be lovely if the world met this ideal and we could follow these stages one by one, in perfect order, like all the old textbooks told us to. As usual, the real world is much murkier. No matter how hard we try to separate these stages, we'll always find things that need further analysis while we're designing. When we're programming, we find features that need clarification in the design.

Most 21st century development recognizes that this cascade (or waterfall) of stages doesn't work out well. What seems to be better is an *iterative* development model. In iterative development, a small part of the task is modeled, designed, and programmed, and then the product is reviewed and expanded to improve each feature and include new features in a series of short development cycles.

The rest of this book is about object-oriented programming, but in this chapter, we will cover the basic object-oriented principles in the context of design. This allows us to understand concepts without having to argue with software syntax or Python tracebacks.

Objects and classes

An **object** is a collection of data with associated behaviors. How do we differentiate between types of objects? Apples and oranges are both objects, but it is a common adage that they cannot be compared. Apples and oranges aren't modeled very often in computer programming, but let's pretend we're doing an inventory application for a fruit farm. To facilitate this example, we can assume that apples go in barrels and oranges go in baskets.

The problem domain we've uncovered so far has four kinds of objects: apples, oranges, baskets, and barrels. In object-oriented modeling, the term used for a *kind of object* is **class**. So, in technical terms, we now have four classes of objects.

It's important to understand the difference between an object and a class. Classes describe related objects. They are like blueprints for creating an object. You might have three oranges sitting on the table in front of you. Each orange is a distinct object, but all three have the attributes and behaviors associated with one class: the general class of oranges.

The relationship between the four classes of objects in our inventory system can be described using a **Unified Modeling Language** (invariably referred to as **UML**, because three-letter acronyms never go out of style) class diagram. Here is our first *class diagram*:

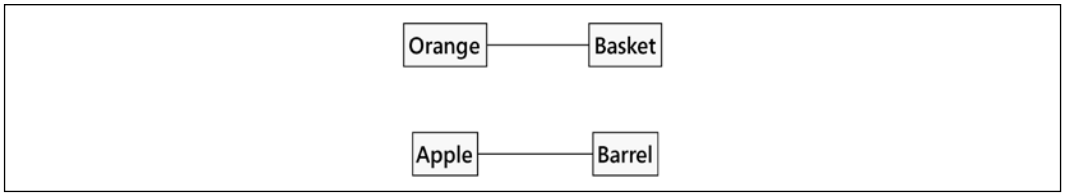

Figure 1.1: Class diagram

This diagram shows that instances of the **Orange** class (usually called "oranges") are somehow associated with a **Basket** and that instances of the **Apple** class ("apples") are also somehow associated with a **Barrel**. *Association* is the most basic way for instances of two classes to be related.

The syntax of a UML diagram is generally pretty obvious; you don't have to read a tutorial to (mostly) understand what is going on when you see one. UML is also fairly easy to draw, and quite intuitive. After all, many people, when describing classes and their relationships, will naturally draw boxes with lines between them. Having a standard based on these intuitive diagrams makes it easy for programmers to communicate with designers, managers, and each other.

Note that the UML diagram generally depicts the class definitions, but we're describing attributes of the objects. The diagram shows the class of Apple and the class of Barrel, telling us that a given apple is in a specific barrel. While we can use UML to depict individual objects, that's rarely necessary. Showing these classes tells us enough about the objects that are members of each class.

Some programmers disparage UML as a waste of time. Citing iterative development, they will argue that formal specifications done up in fancy UML diagrams are going to be redundant before they're implemented, and that maintaining these formal diagrams will only waste time and not benefit anyone.

Every programming team consisting of more than one person will occasionally have to sit down and hash out the details of the components being built. UML is extremely useful for ensuring quick, easy, and consistent communication. Even those organizations that scoff at formal class diagrams tend to use some informal version of UML in their design meetings or team discussions.

Furthermore, the most important person you will ever have to communicate with is your future self. We all think we can remember the design decisions we've made, but there will always be *Why did I do that?* moments hiding in our future. If we keep the scraps of papers we did our initial diagramming on when we started a design, we'll eventually find them to be a useful reference.

This chapter, however, is not meant to be a tutorial on UML. There are many of those available on the internet, as well as numerous books on the topic. UML covers far more than class and object diagrams; it also has a syntax for use cases, deployment, state changes, and activities. We'll be dealing with some common class diagram syntax in this discussion of object-oriented design. You can pick up the structure by example, and then you'll subconsciously choose the UML-inspired syntax in your own team or personal design notes.

Our initial diagram, while correct, does not remind us that apples go in barrels or how many barrels a single apple can go in. It only tells us that apples are somehow associated with barrels. The association between classes is often obvious and needs no further explanation, but we have the option to add further clarification as needed.

The beauty of UML is that most things are optional. We only need to specify as much information in a diagram as makes sense for the current situation. In a quick whiteboard session, we might just draw simple lines between boxes. In a formal document, we might go into more detail.

In the case of apples and barrels, we can be fairly confident that the association is **many apples go in one barrel**, but just to make sure nobody confuses it with **one apple spoils one barrel**, we can enhance the diagram, as shown here:

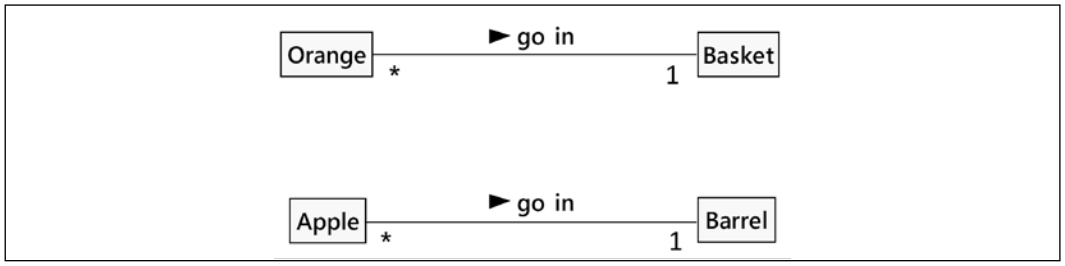

Figure 1.2: Class diagram with more detail

This diagram tells us that oranges **go in** baskets, with a little arrow showing what goes in what. It also tells us the number of that object that can be used in the association on both sides of the relationship. One **Basket** can hold many (represented by a *) **Orange** objects. Any one **Orange** can go in exactly one **Basket**. This number is referred to as the *multiplicity* of the object. You may also hear it described as the *cardinality*; it can help to think of cardinality as a specific number or range, and what we're using here, multiplicity, as a generalized "more-than-one instance".

We may sometimes forget which end of the relationship line is supposed to have which multiplicity number. The multiplicity nearest to a class is the number of objects of that class that can be associated with any one object at the other end of the association. For the apple goes in barrel association, reading from left to right, many instances of the **Apple** class (that is, many **Apple** objects) can go in any one **Barrel**. Reading from right to left, exactly one **Barrel** can be associated with any one **Apple**.

We've seen the basics of classes, and how they specify relationships among objects. Now, we need to talk about the attributes that define an object's state, and the behaviors of an object that may involve state change or interaction with other objects.

Specifying attributes and behaviors

We now have a grasp of some basic object-oriented terminology. Objects are instances of classes that can be associated with each other. A class instance is a specific object with its own set of data and behaviors; a specific orange on the table in front of us is said to be an instance of the general class of oranges.

The orange has a state, for example, ripe or raw; we implement the state of an object via the values of specific attributes. An orange also has behaviors. By themselves, oranges are generally passive. State changes are imposed on them. Let's dive into the meaning of those two words, *state* and *behaviors*.

Data describes object state

Let's start with data. Data represents the individual characteristics of a certain object; its current state. A class can define specific sets of characteristics that are part of all objects that are members of that class. Any specific object can have different data values for the given characteristics. For example, the three oranges on our table (if we haven't eaten any) could each weigh a different amount. The orange class could have a weight attribute to represent that datum. All instances of the orange class have a weight attribute, but each orange has a different value for this attribute. Attributes don't have to be unique, though; any two oranges may weigh the same amount.

Attributes are frequently referred to as **members** or **properties**. Some authors suggest that the terms have different meanings, usually that attributes are settable, while properties are read-only. A Python property can be defined as read-only, but the value will be based on attribute values that are – ultimately – writable, making the concept of *read-only* rather pointless; throughout this book, we'll see the two terms used interchangeably. In addition, as we'll discuss in *Chapter 5*, *When to Use Object-Oriented Programming*, the `property` keyword has a special meaning in Python for a particular kind of attribute.

In Python, we can also call an attribute an **instance variable**. This can help clarify the way attributes work. They are variables with unique values for each instance of a class. Python has other kinds of attributes, but we'll focus on the most common kind to get started.

In our fruit inventory application, the fruit farmer may want to know what orchard the orange came from, when it was picked, and how much it weighs. They might also want to keep track of where each **Basket** is stored. Apples might have a color attribute, and barrels might come in different sizes.

Some of these properties may also belong to multiple classes (we may want to know when apples are picked, too), but for this first example, let's just add a few different attributes to our class diagram:

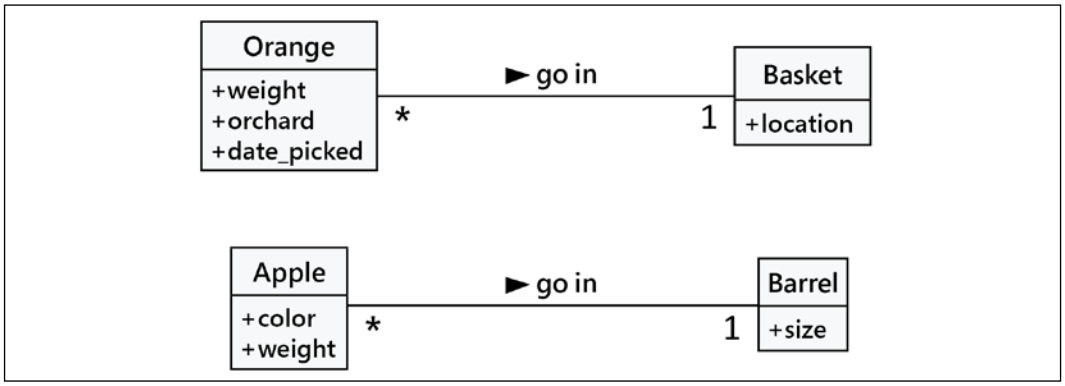

Figure 1.3: Class diagram with attributes

Depending on how detailed our design needs to be, we can also specify the type for each attribute's value. In UML, attribute types are often generic names common to many programming languages, such as integer, floating-point number, string, byte, or Boolean. However, they can also represent generic collections such as lists, trees, or graphs, or most notably, other, non-generic, application-specific classes. This is one area where the design stage can overlap with the programming stage. The various primitives and built-in collections available in one programming language may be different from what is available in another.

Here's a version with (mostly) Python-specific type hints:

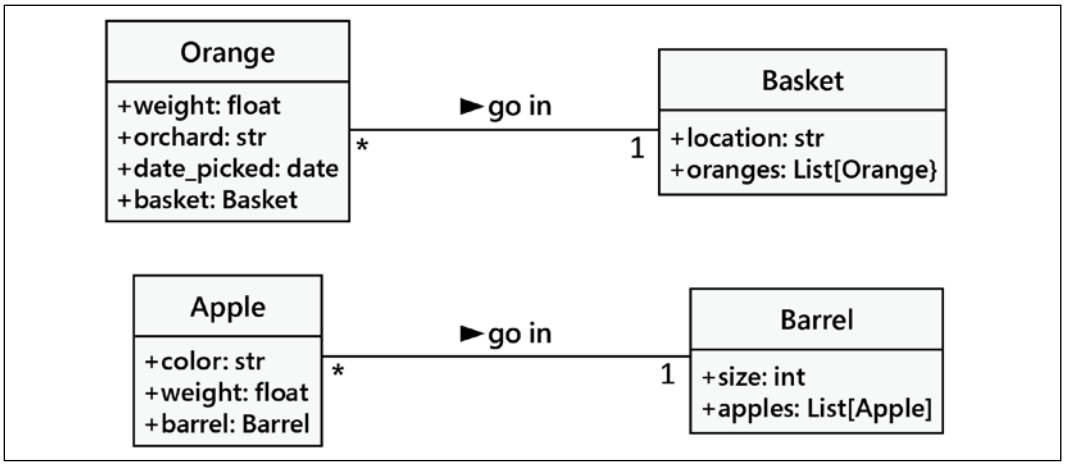

Figure 1.4: Class diagram with attributes and their types

Usually, we don't need to be overly concerned with data types at the design stage, as implementation-specific details are chosen during the programming stage. Generic names are normally sufficient for design; that's why we included `date` as a placeholder for a Python type like `datetime.datetime`. If our design calls for a list container type, Java programmers can choose to use a `LinkedList` or an `ArrayList` when implementing it, while Python programmers (that's us!) might specify `List[Apple]` as a type hint, and use the `list` type for the implementation.

In our fruit-farming example so far, our attributes are all basic primitives. However, there are some implicit attributes that we can make explicit – the associations. For a given orange, we have an attribute referring to the basket that holds that orange, the `basket` attribute, with a type hint of `Basket`.

Behaviors are actions

Now that we know how data defines the object's state, the last undefined term we need to look at is *behaviors*. Behaviors are actions that can occur on an object. The behaviors that can be performed on a specific class of object are expressed as the **methods** of the class. At the programming level, methods are like functions in structured programming, but they have access to the attributes – in particular, the instance variables with the data associated with this object. Like functions, methods can also accept **parameters** and return **values**.

A method's parameters are provided to it as a collection of objects that need to be **passed** into that method. The actual object instances that are passed into a method during a specific invocation are usually referred to as **arguments**. These objects are bound to **parameter** variables in the method body. They are used by the method to perform whatever behavior or task it is meant to do. Returned values are the results of that task. Internal state changes are another possible effect of evaluating a method.

We've stretched our *comparing apples and oranges* example into a basic (if far-fetched) inventory application. Let's stretch it a little further and see whether it breaks. One action that can be associated with oranges is the **pick** action. If you think about implementation, **pick** would need to do two things:

- Place the orange in a basket by updating the **Basket** attribute of the orange.
- Add the orange to the **Orange** list on the given **Basket**.

So, **pick** needs to know what basket it is dealing with. We do this by giving the **pick** method a **Basket** parameter. Since our fruit farmer also sells juice, we can add a **squeeze** method to the **Orange** class. When called, the **squeeze** method might return the amount of juice retrieved, while also removing the **Orange** from the **Basket** it was in.

The class **Basket** can have a **sell** action. When a basket is sold, our inventory system might update some data on as-yet unspecified objects for accounting and profit calculations. Alternatively, our basket of oranges might go bad before we can sell them, so we add a **discard** method. Let's add these methods to our diagram:

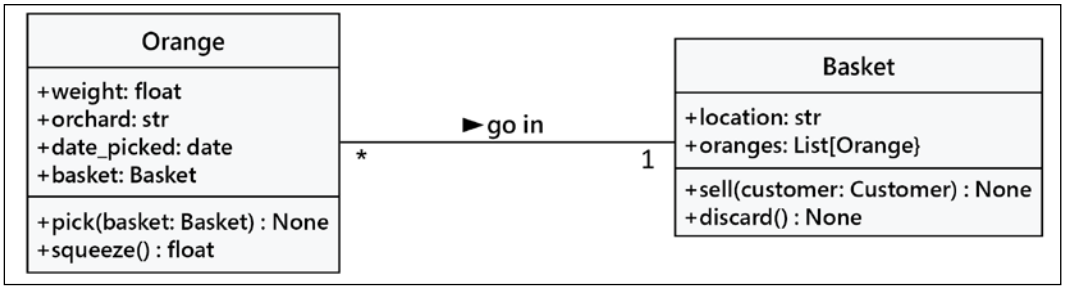

Figure 1.5: Class diagram with attributes and methods

Adding attributes and methods to individual objects allows us to create a **system** of interacting objects. Each object in the system is a member of a certain class. These classes specify what types of data the object can hold and what methods can be invoked on it. The data in each object can be in a different state from other instances of the same class; each object may react to method calls differently because of the differences in state.

Object-oriented analysis and design is all about figuring out what those objects are and how they should interact. Each class has responsibilities and collaborations. The next section describes principles that can be used to make those interactions as simple and intuitive as possible.

Note that selling a basket is not unconditionally a feature of the **Basket** class. It may be that some other class (not shown) cares about the various Baskets and where they are. We often have boundaries around our design. We will also have questions about responsibilities allocated to various classes. The responsibility allocation problem doesn't always have a tidy technical solution, forcing us to draw (and redraw) our UML diagrams more than once to examine alternative designs.

Hiding details and creating the public interface

The key purpose of modeling an object in object-oriented design is to determine what the public **interface** of that object will be. The interface is the collection of attributes and methods that other objects can access to interact with that object. Other objects do not need, and in some languages are not allowed, to access the internal workings of the object.

A common real-world example is the television. Our interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object. When we, as the calling object, access these methods, we do not know or care if the television is getting its signal from a cable connection, a satellite dish, or an internet-enabled device. We don't care what electronic signals are being sent to adjust the volume, or whether the sound is destined for speakers or headphones. If we open the television to access its internal workings, for example, to split the output signal to both external speakers and a set of headphones, we may void the warranty.

This process of hiding the implementation of an object is suitably called **information hiding**. It is also sometimes referred to as **encapsulation**, but encapsulation is actually a more encompassing term. Encapsulated data is not necessarily hidden. Encapsulation is, literally, creating a capsule (or wrapper) on the attributes. The TV's external case encapsulates the state and behavior of the television. We have access to the external screen, the speakers, and the remote. We don't have direct access to the wiring of the amplifiers or receivers within the TV's case.

When we buy a component entertainment system, we change the level of encapsulation, exposing more of the interfaces between components. If we're an Internet of Things maker, we may decompose this even further, opening cases and breaking the information hiding attempted by the manufacturer.

The distinction between encapsulation and information hiding is largely irrelevant, especially at the design level. Many practical references use these terms interchangeably. As Python programmers, we don't actually have or need information hiding via completely private, inaccessible variables (we'll discuss the reasons for this in *Chapter 2*, *Objects in Python*), so the more encompassing definition for encapsulation is suitable.

The public interface, however, is very important. It needs to be carefully designed as it can be difficult to change when other classes depend on it. Changing an interface can break any client objects that depend on it. We can change the internals all we like, for example, to make it more efficient, or to access data over the network as well as locally, and the client objects will still be able to talk to it, unmodified, using the public interface. On the other hand, if we alter the interface by changing publicly accessed attribute names or the order or types of arguments that a method can accept, all client classes will also have to be modified. When designing public interfaces, keep it simple. Always design the interface of an object based on how easy it is to use, not how hard it is to code (this advice applies to user interfaces as well). For this reason, you'll sometimes see Python variables with a leading _ in their name as a warning that these aren't part of the public interface.

Remember, program objects may represent real objects, but that does not make them real objects. They are models. One of the greatest gifts of modeling is the ability to ignore irrelevant details. The model car one of the authors built as a child looked like a real 1956 Thunderbird on the outside, but it obviously didn't run. When they were too young to drive, these details were overly complex and irrelevant. The model is an **abstraction** of a real concept.

Abstraction is another object-oriented term related to encapsulation and information hiding. Abstraction means dealing with the level of detail that is most appropriate to a given task. It is the process of extracting a public interface from the inner details. A car's driver needs to interact with the steering, accelerator, and brakes. The workings of the motor, drive train, and brake subsystem don't matter to the driver. A mechanic, on the other hand, works at a different level of abstraction, tuning the engine and bleeding the brakes. Here's an example of two abstraction levels for a car:

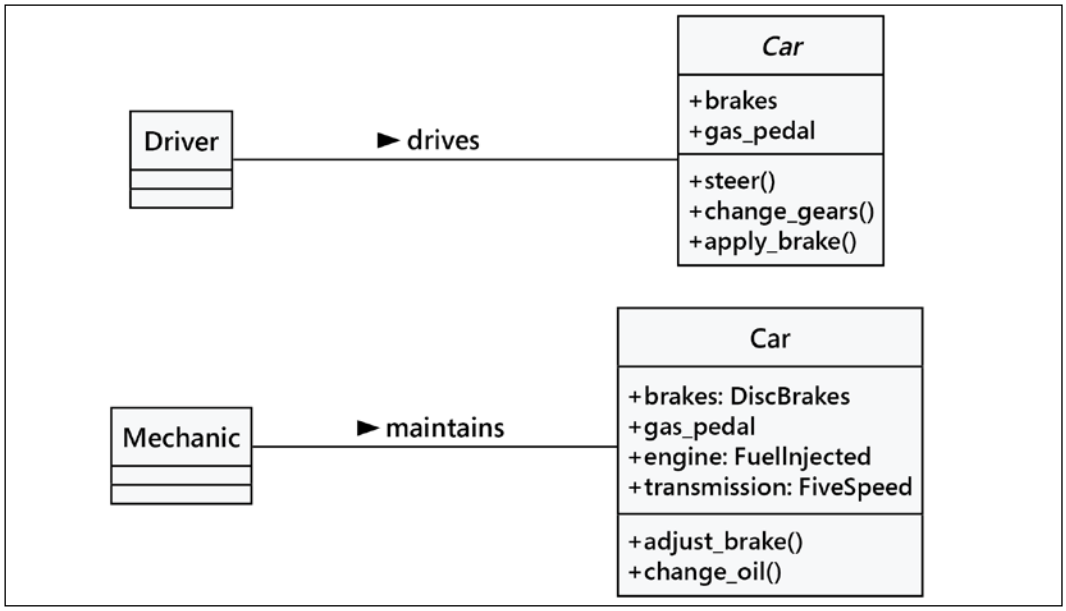

Figure 1.6: Abstraction levels for a car

Now, we have several new terms that refer to similar concepts. Let's summarize all this jargon in a couple of sentences: abstraction is the process of encapsulating information with a separate public interface. Any private elements can be subject to information hiding. In UML diagrams, we might use a leading – instead of a leading + to suggest it's not part of a public interface.

The important lesson to take away from all these definitions is to make our models understandable to other objects that have to interact with them. This means paying careful attention to small details.

Ensure methods and properties have sensible names. When analyzing a system, objects typically represent nouns in the original problem, while methods are normally verbs. Attributes may show up as adjectives or more nouns. Name your classes, attributes, and methods accordingly.

When designing the interface, imagine you are the object; you want clear definitions of your responsibility and you have a very strong preference for privacy to meet those responsibilities. Don't let other objects have access to data about you unless you feel it is in your best interest for them to have it. Don't give them an interface to force you to perform a specific task unless you are certain it's your responsibility to do that.

Composition

So far, we have learned to design systems as a group of interacting objects, where each interaction involves viewing objects at an appropriate level of abstraction. But we don't yet know how to create these levels of abstraction. There are a variety of ways to do this; we'll discuss some advanced design patterns in *Chapters 10*, *11*, and *12*. But even most design patterns rely on two basic object-oriented principles known as **composition** and **inheritance**. Composition is simpler, so let's start with that.

Composition is the act of collecting several objects together to create a new one. Composition is usually a good choice when one object is part of another object. We've already seen a first hint of composition when talking about cars. A fossil-fueled car is composed of an engine, transmission, starter, headlights, and windshield, among numerous other parts. The engine, in turn, is composed of pistons, a crank shaft, and valves. In this example, composition is a good way to provide levels of abstraction. The **Car** object can provide the interface required by a driver, while also giving access to its component parts, which offers the deeper level of abstraction suitable for a mechanic. Those component parts can, of course, be further decomposed into details if the mechanic needs more information to diagnose a problem or tune the engine.

A car is a common introductory example of composition, but it's not overly useful when it comes to designing computer systems. Physical objects are easy to break into component objects. People have been doing this at least since the ancient Greeks originally postulated that atoms were the smallest units of matter (they, of course, didn't have access to particle accelerators). Because computer systems involve a lot of peculiar concepts, identifying the component objects does not happen as naturally as with real-world valves and pistons.

The objects in an object-oriented system occasionally represent physical objects such as people, books, or telephones. More often, however, they represent concepts. People have names, books have titles, and telephones are used to make calls. Calls, titles, accounts, names, appointments, and payments are not usually considered objects in the physical world, but they are all frequently-modeled components in computer systems.

Let's try modeling a more computer-oriented example to see composition in action. We'll be looking at the design of a computerized chess game. This was a very popular pastime in the 80s and 90s. People were predicting that computers would one day be able to defeat a human chess master. When this happened in 1997 (IBM's Deep Blue defeated world chess champion, Gary Kasparov), interest in the problem of chess waned. Nowadays, the descendants of Deep Blue always win.

A *game* of chess is **played** between two *players*, using a chess set featuring a *board* containing 64 *positions* in an 8×8 grid. The board can have two sets of 16 *pieces* that can be **moved**, in alternating *turns* by the two players in different ways. Each piece can **take** other pieces. The board will be required to **draw** itself on the computer *screen* after each turn.

I've identified some of the possible objects in the description using *italics*, and a few key methods using **bold**. This is a common first step in turning an object-oriented analysis into a design. At this point, to emphasize composition, we'll focus on the board, without worrying too much about the players or the different types of pieces.

Let's start at the highest level of abstraction possible. We have two players interacting with a **Chess Set** by taking turns making moves:

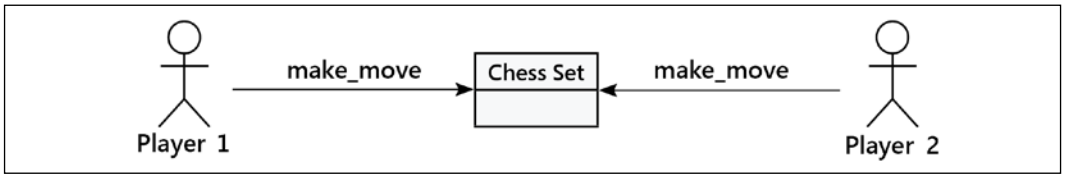

Figure 1.7: Object/instance diagram for a chess game

This doesn't quite look like our earlier class diagrams, which is a good thing since it isn't one! This is an **object diagram**, also called an **instance diagram**. It describes the system at a specific state in time, and is describing specific instances of objects, not the interaction between classes. Remember, both players are members of the same class, so the class diagram looks a little different:

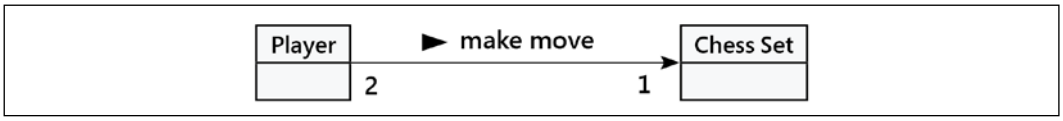

Figure 1.8: Class diagram for a chess game

This diagram shows that exactly two players can interact with one chess set. This also indicates that any one player can be playing with only one **Chess Set** at a time.

However, we're discussing composition, not UML, so let's think about what the **Chess Set** is composed of. We don't care what the player is composed of at this time. We can assume that the player has a heart and brain, among other organs, but these are irrelevant to our model. Indeed, there is nothing stopping said player from being Deep Blue itself, which has neither a heart nor a brain.

The chess set, then, is composed of a board and 32 pieces. The board further comprises 64 positions. You could argue that these pieces are not part of the chess set, because you could replace the pieces of a chess set with a different set of pieces. While this is unlikely or impossible in a computerized version of chess, it introduces us to **aggregation**.

Aggregation is almost exactly like composition. The difference is that aggregate objects can exist independently. It would be impossible for a position to be associated with a different chess board, so we say the board is composed of positions. But the pieces, which might exist independently of the chess set, are said to be in an aggregate relationship with that set.

Another way to differentiate between aggregation and composition is to think about the lifespan of the object:

- If the composite (outside) object controls when the related (inside) objects are created and destroyed, composition is most suitable.
- If the related object is created independently of the composite object, or can outlast that object, an aggregate relationship makes more sense.

Also, keep in mind that composition is aggregation; aggregation is simply a more general form of composition. Any composite relationship is also an aggregate relationship, but not vice versa.

Let's describe our current **Chess Set** composition and add some attributes to the objects to hold the composite relationships:

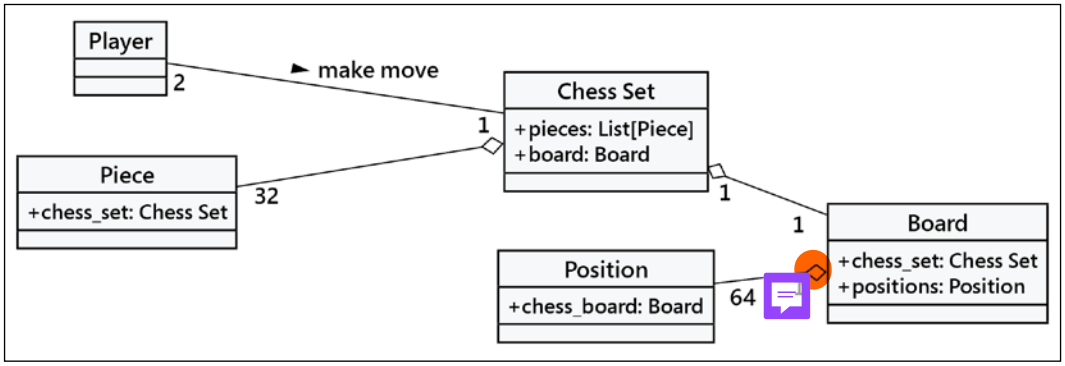

Figure 1.9: Class diagram for a chess game

The composition relationship is represented in UML as a solid diamond. The hollow diamond represents the aggregate relationship. You'll notice that the board and pieces are stored as part of the **Chess Set** in exactly the same way a reference to them is stored as an attribute on the chess set. This shows that, once again, in practice, the distinction between aggregation and composition is often irrelevant once you get past the design stage. When implemented, they behave in much the same way.

This distinction can help you differentiate between the two when your team is discussing how the different objects interact. You'll often need to distinguish between them when talking about how long related objects exist. In many cases, deleting a composite object (like the board) deletes all the locations. The aggregated objects, however, are not deleted automatically.

Inheritance

We discussed three types of relationships between objects: association, composition, and aggregation. However, we have not fully specified our chess set, and these tools don't seem to give us all the power we need. We discussed the possibility that a player might be a human or it might be a piece of software featuring artificial intelligence. It doesn't seem right to say that a player is *associated* with a human, or that the artificial intelligence implementation is *part of* the player object. What we really need is the ability to say that *Deep Blue is a player*, or that *Gary Kasparov is a player*.

The *is a* relationship is formed by **inheritance**. Inheritance is the most famous, well-known, and overused relationship in object-oriented programming. Inheritance is sort of like a family tree. Dusty Phillips is one of this book's authors.

His grandfather's last name was Phillips, and his father inherited that name. Dusty inherited it from him. In object-oriented programming, instead of inheriting features and behaviors from a person, one class can inherit attributes and methods from another class.

For example, there are 32 chess pieces in our chess set, but there are only six different types of pieces (pawns, rooks, bishops, knights, king, and queen), each of which behaves differently when it is moved. All of these classes of piece have properties, such as color and the chess set they are part of, but they also have unique shapes when drawn on the chess board, and make different moves. Let's see how the six types of pieces can inherit from a **Piece** class:

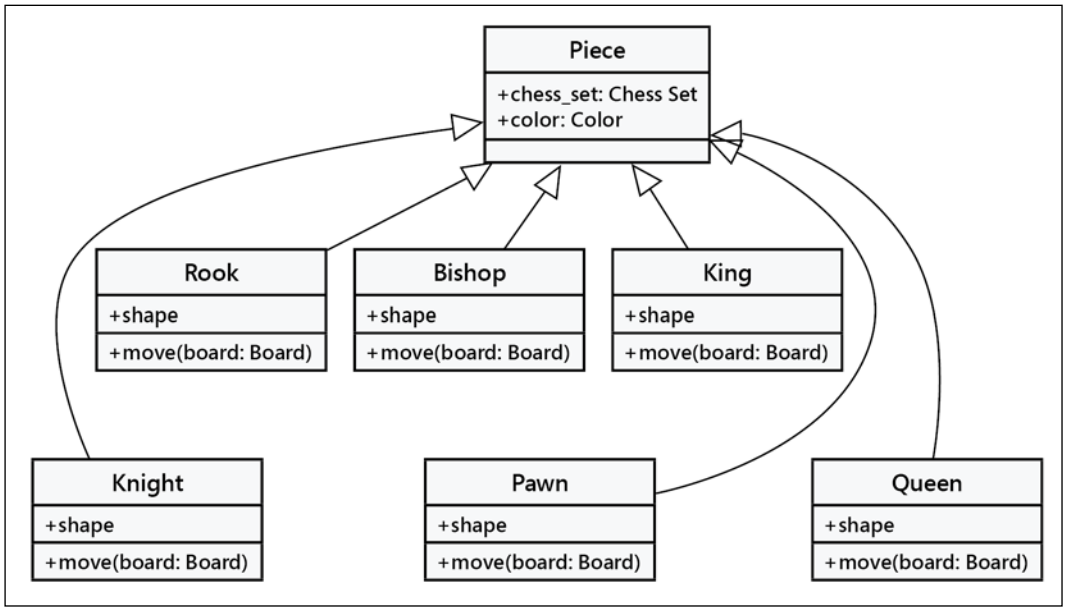

Figure 1.10: How chess pieces inherit from the Piece class

The hollow arrows indicate that the individual classes of pieces inherit from the **Piece** class. All the child classes automatically have a **chess_set** and **color** attribute inherited from the base class. Each piece provides a different shape property (to be drawn on the screen when rendering the board), and a different **move** method to move the piece to a new position on the board at each turn.

We actually know that all subclasses of the **Piece** class need to have a **move** method; otherwise, when the board tries to move the piece, it will get confused. It is possible that we would want to create a new version of the game of chess that has one additional piece (the wizard). Our current design will allow us to design this piece without giving it a **move** method. The board would then choke when it asked the piece to move itself.

We can fix this by creating a dummy move method on the **Piece** class. The subclasses can then **override** this method with a more specific implementation. The default implementation might, for example, pop up an error message that says **That piece cannot be moved**.

Overriding methods in subclasses allows very powerful object-oriented systems to be developed. For example, if we wanted to implement a **Player** class with artificial intelligence, we might provide a **calculate_move** method that takes a **Board** object and decides which piece to move where. A very basic class might randomly choose a piece and direction and move it accordingly. We could then override this method in a subclass with the Deep Blue implementation. The first class would be suitable for play against a raw beginner; the latter would challenge a grand master. The important thing is that other methods in the class, such as the ones that inform the board as to which move was chosen, need not be changed; this implementation can be shared between the two classes.

In the case of chess pieces, it doesn't really make sense to provide a default implementation of the move method. All we need to do is specify that the move method is required in any subclasses. This can be done by making **Piece** an **abstract class** with the **move** method declared as **abstract**. Abstract methods basically say this:

> "*We demand this method exist in any non-abstract subclass, but we are declining to specify an implementation in this class.*"

Indeed, it is possible to make an abstraction that does not implement any methods at all. Such a class would simply tell us what the class *should* do, but provides absolutely no advice on how to do it. In some languages, these purely abstract classes are called **interfaces**. It's possible to define a class with only abstract method placeholders in Python, but it's very rare.

Inheritance provides abstraction

Let's explore the longest word in object-oriented argot. **Polymorphism** is the ability to treat a class differently, depending on which subclass is implemented. We've already seen it in action with the pieces system we've described. If we took the design a bit further, we'd probably see that the **Board** object can accept a move from the player and call the **move** function on the piece. The board need not ever know what type of piece it is dealing with. All it has to do is call the **move** method, and the proper subclass will take care of moving it as a **Knight** or a **Pawn**.

Polymorphism is pretty cool, but it is a word that is rarely used in Python programming. Python goes an extra step past allowing a subclass of an object to be treated like a parent class. A board implemented in Python could take any object that has a **move** method, whether it is a bishop piece, a car, or a duck. When **move** is called, the **Bishop** will move diagonally on the board, the car will drive someplace, and the duck will swim or fly, depending on its mood.

This sort of polymorphism in Python is typically referred to as **duck typing**: *if it walks like a duck or swims like a duck, we call it a duck*. We don't care if it really *is a* duck (*is a* being a cornerstone of inheritance), only that it swims or walks. Geese and swans might easily be able to provide the duck-like behavior we are looking for. This allows future designers to create new types of birds without actually specifying a formal inheritance hierarchy for all possible kinds of aquatic birds. The chess examples, above, use formal inheritance to cover all possible pieces in the chess set. Duck typing also allows a programmer to extend a design, creating completely different drop-in behaviors the original designers never planned for. For example, future designers might be able to make a walking, swimming penguin that works with the same interface without ever suggesting that penguins have a common superclass with ducks.

Multiple inheritance

When we think of inheritance in our own family tree, we can see that we inherit features from more than just one parent. When strangers tell a proud mother that her son has *his father's eyes*, she will typically respond along the lines of, *yes, but he got my nose*.

Object-oriented design can also feature such **multiple inheritance**, which allows a subclass to inherit functionality from multiple parent classes. In practice, multiple inheritance can be a tricky business, and some programming languages (most famously, Java) strictly prohibit it. However, multiple inheritance can have its uses. Most often, it can be used to create objects that have two distinct sets of behaviors. For example, an object designed to connect to a scanner to make an image and send a fax of the scanned image might be created by inheriting from two separate `scanner` and `faxer` objects.

As long as two classes have distinct interfaces, it is not normally harmful for a subclass to inherit from both of them. However, it gets messy if we inherit from two classes that provide overlapping interfaces. The scanner and faxer don't have any overlapping features, so combining features from both is easy. Our counterexample is a motorcycle class that has a `move` method, and a boat class also featuring a `move` method.

If we want to merge them into the ultimate amphibious vehicle, how does the resulting class know what to do when we call `move`? At the design level, this needs to be explained. (As a sailor who lived on a boat, one of the authors really wants to know how this is supposed to work.)

Python has a defined **method resolution order** (**MRO**) to help us understand which of the alternative methods will be used. While the MRO rules are simple, avoiding overlap is even simpler. Multiple inheritance as a "mixin" technique for combining unrelated aspects can be helpful. In many cases, though, a composite object may be easier to design.

Inheritance is a powerful tool for extending behavior and reusing features. It is also one of the most marketable advancements of object-oriented design over earlier paradigms. Therefore, it is often the first tool that object-oriented programmers reach for. However, it is important to recognize that owning a hammer does not turn screws into nails. Inheritance is the perfect solution for obvious *is a* relationships. Beyond this, it can be abused. Programmers often use inheritance to share code between two kinds of objects that are only distantly related, with no *is a* relationship in sight. While this is not necessarily a bad design, it is a terrific opportunity to ask just why they decided to design it that way, and whether a different relationship or design pattern would have been more suitable.

Case study

Our case study will span many of the chapters of this book. We'll be examining a single problem closely from a variety of perspectives. It's very important to look at alternative designs and design patterns; more than once, we'll point out that there's no single right answer: there are a number of good answers. Our intent here is to provide a realistic example that involves realistic depth and complications and leads to difficult trade-off decisions. Our goal is to help the reader apply object-oriented programming and design concepts. This means choosing among the technical alternatives to create something useful.

This first part of the case study is an overview of the problem and why we're tackling it. This background will cover a number of aspects of the problem to set up the design and construction of solutions in later chapters. Part of this overview will include some UML diagrams to capture elements of the problem to be solved. These will evolve in later chapters as we dive into the consequences of design choices and make changes to those design choices.

As with many realistic problems, the authors bring personal bias and assumptions. For information on the consequences of this, consider books like *Technically Wrong*, by Sara Wachter-Boettcher.

Our users want to automate a job often called **classification**. This is the underpinning idea behind product recommendations: last time, a customer bought product X, so perhaps they'd be interested in a similar product, Y. We've classified their desires and can locate other items in that class of products. This problem can involve complex data organization issues.

It helps to start with something smaller and more manageable. The users eventually want to tackle complex consumer products, but recognize that solving a difficult problem is not a good way to learn how to build this kind of application. It's better to start with something of a manageable level of complexity and then refine and expand it until it does everything they need. In this case study, therefore, we'll be building a classifier for iris species. This is a classic problem, and there's a great deal written about approaches to classifying iris flowers.

A training set of data is required, which the classifier uses as examples of correctly classified irises. We will discuss what the training data looks like in the next section.

We'll create a collection of diagrams using the **Unified Modeling Language** (**UML**) to help depict and summarize the software we're going to build.

We'll examine the problem using a technique called **4+1 Views**. The views are:

- A **logical view** of the data entities, their static attributes, and their relationships. This is the heart of object-oriented design.
- A **process view** that describes how the data is processed. This can take a variety of forms, including state models, activity diagrams, and sequence diagrams.
- A **development view** of the code components to be built. This diagram shows relationships among software components. This is used to show how class definitions are gathered into modules and packages.
- A **physical view** of the application to be integrated and deployed. In cases where an application follows a common design pattern, a sophisticated diagram isn't necessary. In other cases, a diagram is essential to show how a collection of components are integrated and deployed.
- A **context view** that provides a unifying context for the other four views. The context view will often describe the actors that use (or interact) with the system to be built. This can involve human actors as well as automated interfaces: both are outside the system, and the system must respond to these external actors.

It's common to start with the context view so that we have a sense of what the other views describe. As our understanding of the users and the problem domain evolves, the context will evolve also.

It's very important to recognize that all of these 4+1 views evolve together. A change to one will generally be reflected in other views. It's a common mistake to think that one view is in some way foundational, and that the other views build on it in a cascade of design steps that always lead to software.

We'll start with a summary of the problem and some background before we start trying to analyze the application or design software.

Introduction and problem overview

As we mentioned previously, we'll be starting with a simpler problem – classifying flowers. We want to implement one popular approach called ***k*-nearest neighbors**, or ***k*-NN** for short. We require a training set of data, which the classifier algorithm uses as examples of correctly classified irises. Each training sample has a number of attributes, reduced to numeric scores, and a final, correct, classification (i.e. iris species). In this iris example, each training sample is an iris, with its attributes, such as petal shape, size, and so on, encoded into a numeric vector that is an overall representation of the iris, along with a correct species label for that iris.

Given an unknown sample, an iris whose species we want to know, we can measure the distance between the unknown sample and any of the known samples in the vector space. For some small group of nearby neighbors, we can take a vote. The unknown sample can be classified into the sub-population selected by the majority of the nearby neighbors.

If we only have two dimensions (or attributes), we can diagram the *k*-NN classification like this:

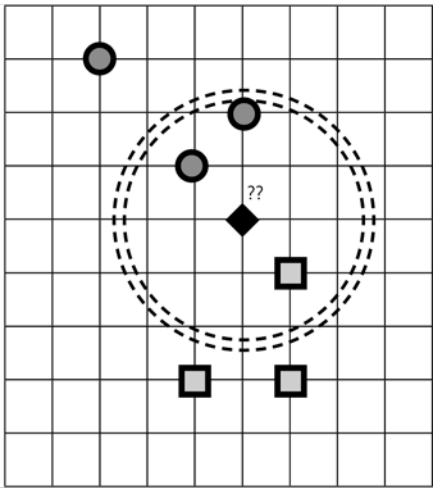

Figure 1.11: k-nearest neighbors

Our unknown sample is a diamond tagged with "??". It's surrounded by known samples of the square and circle species. When we locate the three nearest neighbors, shown inside the dashed circle, we can take a vote and decide that the unknown is most like the Circle species.

One underpinning concept is having tangible, numeric measurements for the various attributes. Converting words, addresses, and other non-ordinal data into an ordinal measurement can be challenging. The good news is that the data we're going to start with data that already has properly ordinal measurements with explicit units of measure.

Another supporting concept is the number of neighbors involved in the voting. This is the *k* factor in *k*-nearest neighbors. In our conceptual diagram, we've shown *k*=3 neighbors; two of the three nearest neighbors are circles, with the third being a square. If we change the *k*-value to 5, this will change the composition of the pool and tip the vote in favor of the squares. Which is right? This is checked by having test data with known right answers to confirm that the classification algorithm works acceptably well. In the preceding diagram, it's clear the diamond was cleverly chosen to be a midway between two clusters, intentionally creating a difficult classification problem.

A popular dataset for learning how this works is the Iris Classification data. See `https://archive.ics.uci.edu/ml/datasets/iris` for some background on this data. This is also available at `https://www.kaggle.com/uciml/iris` and many other places.

More experienced readers may notice some gaps and possible contradictions as we move through the object-oriented analysis and design work. This is intentional. An initial analysis of a problem of any scope will involve learning and rework. This case study will evolve as we learn more. If you've spotted a gap or contradiction, formulate your own design and see if it converges with the lessons learned in subsequent chapters.

Having looked at some aspects of the problem, we can provide a more concrete context with actors and the use cases or scenarios that describe how an actor interacts with the system to be built. We'll start with the context view.

Context view

The context for our application that classifies iris species involves these two classes of actors:

- A "Botanist" who provides the properly classified training data and a properly classified set of test data. The Botanist also runs the test cases to establish the proper parameters for the classification. In the simple case of *k*-NN, they can decide which *k* value should be used.

- A "User" who needs to do classification of unknown data. The user has made careful measurements and makes a request with the measurement data to get a classification from this classifier system. The name "User" seems vague, but we're not sure what's better. We'll leave it for now, and put off changing it until we foresee a problem.

This UML context diagram illustrates the two actors and the three scenarios we will explore:

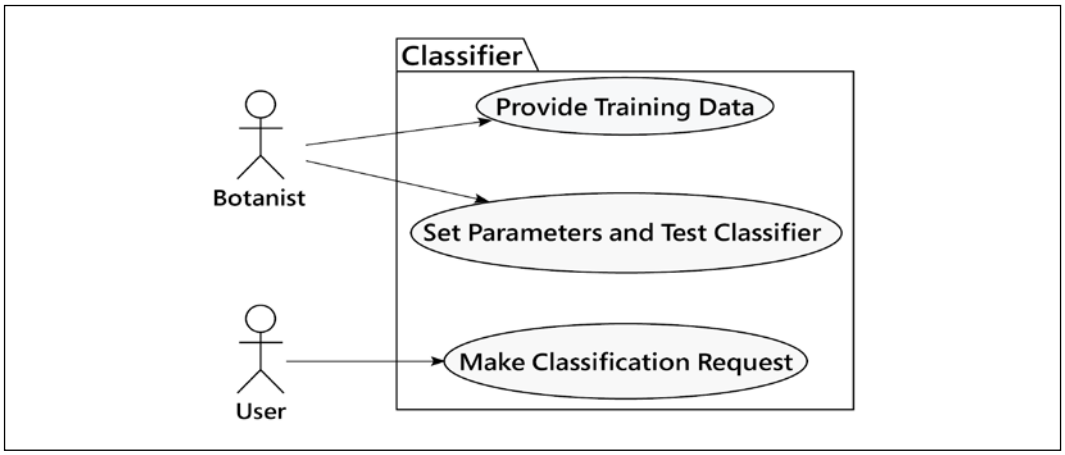

Figure 1.12: UML context diagram

The system as a whole is depicted as a rectangle. It encloses ovals to represent user stories. In the UML, specific shapes have meanings, and we reserve rectangles for objects. Ovals (and circles) are for user stories, which are interfaces to the system.

In order to do any useful processing, we need training data, properly classified. There are two parts to each set of data: a training set and a test set. We'll call the whole assembly "training data" instead of the longer (but more precise) "training and test data."

The tuning parameters are set by the botanist, who must examine the test results to be sure the classifier works. These are the two parameters that can be tuned:

- The distance computation to use
- The number of neighbors to consider for voting

We'll look at these parameters in detail in the *Processing view* section later in this chapter. We'll also revisit these ideas in subsequent case study chapters. The distance computation is an interesting problem.

We can define a set of experiments as a grid of each alternative and methodically fill in the grid with the results of measuring the test set. The combination that provides the best fit will be the recommended parameter set from the botanist. In our case, there are two choices, and the grid is a two-dimensional table, like the one shown below. With more complex algorithms, the "grid" may be a multidimensional space:

		Various k factors		
		k=3	**k=5**	**k=7**
Distance computation algorithms	**Euclidean**	Test results…		
	Manhattan			
	Chebyshev			
	Sorensen			
	Other?			

After the testing, a User can make requests. They provide unknown data to receive classification results from this trained classifier process. In the long run, this "User" won't be a person – they'll be a connection from some website's sales or catalog engine to our clever classifier-based recommendation engine.

We can summarize each of these scenarios with a **use case** or **user story** statement:

- As a Botanist, I want to provide properly classified training and testing data to this system so users can correctly identify plants.
- As a Botanist, I want to examine the test results from the classifier to be sure that new samples are likely to be correctly classified.
- As a User, I want to be able to provide a few key measurements to the classifier and have the iris species correctly classified.

Given the nouns and verbs in the user stories, we can use that information to create a logical view of the data the application will process.

Logical view

Looking at the context diagram, processing starts with training data and testing data. This is properly classified sample data used to test our classification algorithm. The following diagram shows one way to look at a class that contains various training and testing datasets:

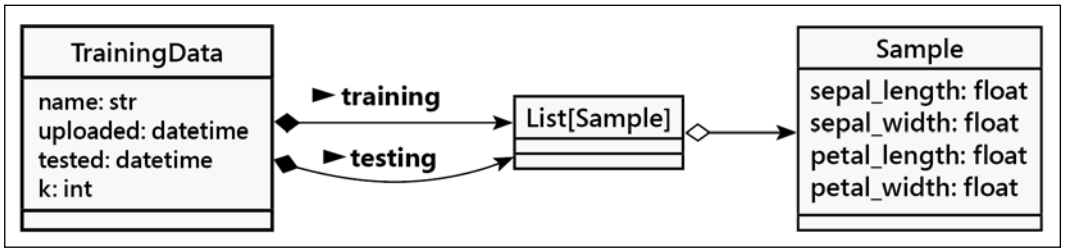

Figure 1.13: Class diagram for training and testing

This shows a `Training Data` class of objects with the attributes of each instance of this class. The `TrainingData` object gives our sample collection a name, and some dates where uploading and testing were completed. For now, it seems like each `TrainingData` object should have a single tuning parameter, `k`, used for the *k*-NN classifier algorithm. An instance also includes two lists of individual samples: a training list and a testing list.

Each class of objects is depicted in a rectangle with a number of individual sections:

- The top-most section provides a name for the class of objects. In two cases, we've used a type hint, `List[Sample]`; the generic class, `list`, is used in a way that ensures the contents of the list are only `Sample` objects.
- The next section of a class rectangle shows the attributes of each object; these attributes are also called the instance variables of this class.
- Later, we'll add "methods" to the bottom section for instances of the class.

Each object of the `Sample` class has a handful of attributes: four floating-point measurement values and a string value, which is the botanist-assigned classification for the sample. In this case, we used the attribute name `class` because that's what it's called in the source data.

The UML arrows show two specific kinds of relationships, highlighted by filled or empty diamonds. A filled diamond shows **composition**: a `TrainingData` object is composed – in part – of two collections. The open diamond shows **aggregation**: a `List[Sample]` object is an aggregate of `Sample` items. To recap what we learned earlier:

- A **composition** is an existential relationship: we can't have `TrainingData` without the two `List[Sample]` objects. And, conversely, a `List[Sample]` object isn't used in our application without being part of a `TrainingData` object.
- An **aggregation**, on the other hand, is a relationship where items can exist independently of each other. In this diagram, a number of `Sample` objects can be part of `List[Sample]` or can exist independently of the list.

It's not clear that the open diamond to show the aggregation of `Sample` objects into a `List` object is relevant. It may be an unhelpful design detail. When in doubt, it's better to omit these kinds of the details until they're clearly required to ensure there's an implementation that meets the user's expectations.

We've shown a `List[Sample]` as a separate class of objects. This is Python's generic `List`, qualified with a specific class of objects, `Sample`, that will be in the list. It's common to avoid this level of detail and summarize the relationships in a diagram like the following:

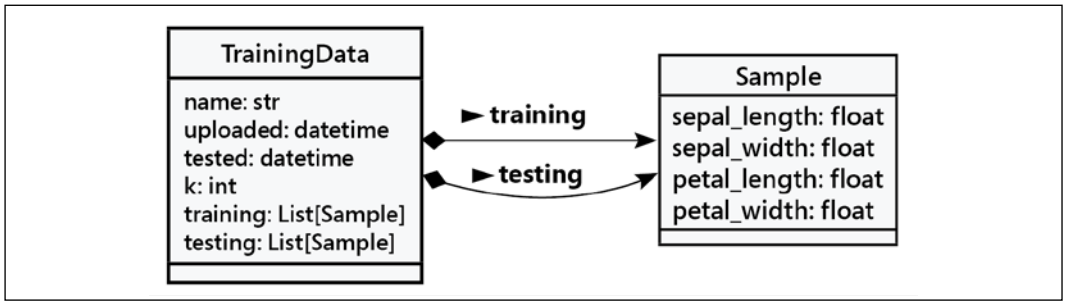

Figure 1.14: Condensed class diagram

This slightly abbreviated form can help with doing analytical work, where the underlying data structures don't matter. It's less helpful for design work, as specific Python class information becomes more important.

Given an initial sketch, we'll compare this logical view with each of the three scenarios mentioned in the context diagram, shown in *Figure 1.12* in the previous section. We want to be sure all of the data and processing in the user stories can be allocated as responsibilities scattered among the classes, attributes, and methods in the diagram.

Walking through the user stories, we uncover these two problems:

- It's not clear how the testing and parameter tuning fit with this diagram. We know there's a *k* factor that's required, but there are no relevant test results to show alternative *k* factors and the consequence of those choices.

- The user's request is not shown at all. Nor is the response to the user. No classes have these items as part of their responsibilities.

The first point suggests we'll need to re-read the user stories and try again to create a better logical view. The second point is a question of boundaries. While the web request and response details are missing, it's more important to describe the essential problem domain – classification and *k*-NN – first. The web services for handling a user's requests is one (of many) solution technologies, and we should set that aside when getting started.

Now, we'll turn our focus to the processing for the data. We're following what seems to be an effective order for creating a description of an application. The data has to be described first; it's the most enduring part, and the thing that is always preserved through each refinement of the processing. The processing can be secondary to the data, because this changes as the context changes and user experience and preferences change.

Process view

There are three separate user stories. This does not necessarily force us to create three process diagrams. For complex processing, there may be more process diagrams than user stories. In some cases, a user story may be too simple to require a carefully designed diagram.

For our application, it seems as though there are at least three unique processes of interest, specifically these:

- Upload the initial set of `Samples` that comprise some `TrainingData`.
- Run a test of the classifier with a given *k* value.
- Make a classification request with a new `Sample` object.

We'll sketch activity diagrams for these use cases. An activity diagram summarizes a number of state changes. The processing begins with a start node and proceeds until an end node is reached. In transaction-based applications, like web services, it's common to omit showing the overall web server engine. This saves us from describing common features of HTTP, including standard headers, cookies, and security concerns. Instead, we generally focus on the unique processing that's performed to create a response for each distinct kind of request.

The activities are shown in round-corner rectangles. Where specific classes of objects or software components are relevant, they can be linked to relevant activities.

What's more important is making sure that the logical view is updated as ideas arise while working on the processing view. It's difficult to get either view done completely in isolation. It's far more important to make incremental changes in each view as new solution ideas arise. In some cases, additional user input is required, and this too will lead to the evolution of these views.

We can sketch a diagram to show how the system responds when the Botanist provides the initial data. Here's the first example:

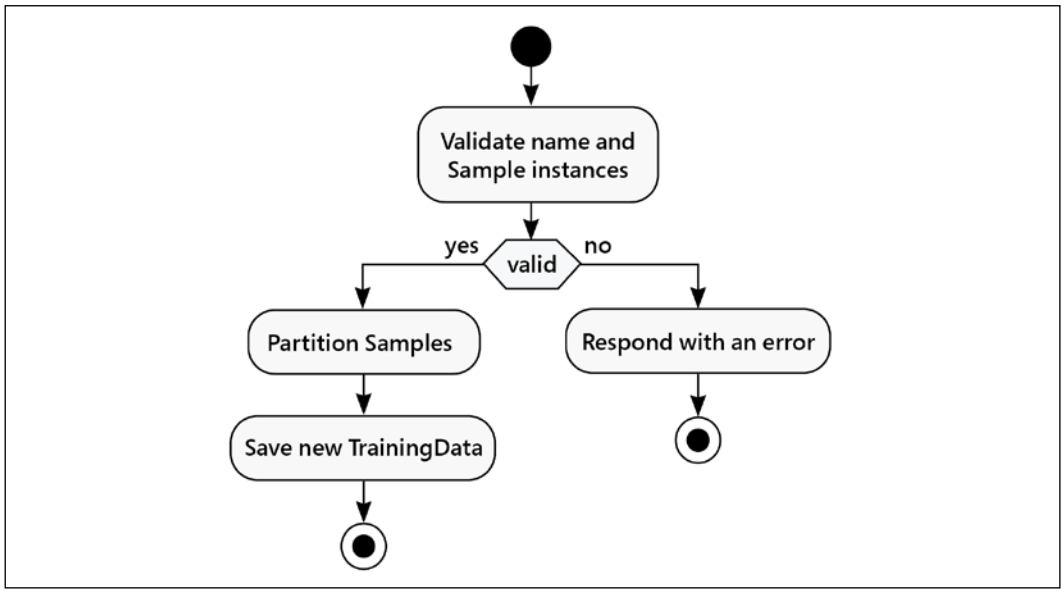

Figure 1.15: Activity diagram

The collection of `KnownSample` values will be partitioned into two subsets: a training subset and a testing subset. There's no rule in our problem summary or user stories for making this distinction; the gap shows we're missing details in the original user story. When details are missing from the user stories, then the logical view may be incomplete, also. For now, we can labor under an assumption that most of the data – say 75% - will be used for training, and the rest, 25%, will be used for testing.

It often helps to create similar diagrams for each of the user stories. It also helps to be sure that the activities all have relevant classes to implement the steps and represent state changes caused by each step.

We've included a verb, `Partition`, in this diagram. This suggests a method will be required to implement the verb. This may lead to rethinking the class model to be sure the processing can be implemented.

We'll turn next to considering some of the components to be built. Since this is a preliminary analysis, our ideas will evolve as we do more detailed design and start creating class definitions.

Development view

There's often a delicate balance between the final deployment and the components to be developed. In rare cases, there are few deployment constraints, and the designer can think freely about the components to be developed. A physical view will evolve from the development. In more common cases, there's a specific target architecture that must be used, and elements of the physical view are fixed.

There are several ways to deploy this classifier as part of a larger application. We might build a desktop application, a mobile application, or a website. Because of the ubiquity of internetworked computers, one common approach is to create a website and connect to it from desktops and mobile apps.

A web services architecture, for example, means requests can be made to a server; the responses could be HTML pages for presentation in a browser, or JSON documents that can be displayed by a mobile application. Some requests will provide whole new sets of training data. Other requests will be seek to classify unknown samples. We'll detail the architecture in the physical view below. We might want to use the Flask framework to build a web service. For more information on Flask, see *Mastering Flask Web Development*, `https://www.packtpub.com/product/mastering-flask-web-development-second-edition/9781788995405`, or *Learning Flask Framework*, `https://www.packtpub.com/product/learning-flask-framework/9781783983360`.

The following diagram shows some of the components we need would need to build for a Flask-based application:

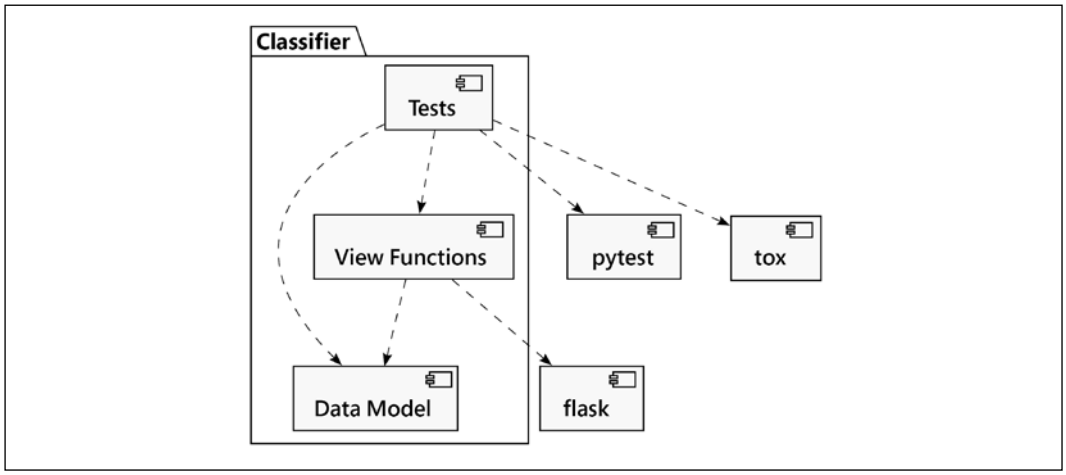

Figure 1.16: Components to be built

This diagram shows a Python package, `Classifier`, that contains a number of modules. The three top-level modules are:

- **Data Model**: (Since this is still analysis time, the name here is not properly Pythonic; we'll change it later as we move into implementation.) It's often helpful to separate the classes that define the problem domain into modules. This makes it possible for us to test them in isolation from any particular application that uses those classes. We'll focus on this part, since it is foundational.
- **View Functions**: (Also an analysis name, not a Pythonic implementation name.) This module will create an instance of the `Flask` class, our application. It will define the functions that handle requests by creating responses that can be displayed by a mobile app or a browser. These functions expose features of the model, and don't involve the same depth and complexity of the model itself; we won't focus on this component in the case study.
- **Tests**: This will have unit tests for the model and view functions. While tests are essential for being sure the software is usable, they are the subject of *Chapter 13, Testing Object-Oriented Programs*.

We have included dependency arrows, using dashed lines. These can be annotated with the Python-specific "imports" label to help clarify how the various packages and modules are related.

As we move through the design in later chapters, we'll expand on this initial view. Having thought about what needs to be built, we can now consider how it's deployed by drawing a physical view of the application. As noted above, there's a delicate dance between development and deployment. The two views are often built together.

Physical view

The physical view shows how the software will be installed into physical hardware. For web services, we often talk about a **continuous integration and continuous deployment** (**CI/CD**) pipeline. This means that a change to the software is tested as a unit, integrated with the existing applications, tested as an integrated whole, then deployed for the users.

While it's common to assume a website, this can also be deployed as a command-line application. It might be run on a local computer. It might also be run on a computer in the cloud. Another choice is to build a web application around the core classifier.

The following diagram shows a view of a web application server:

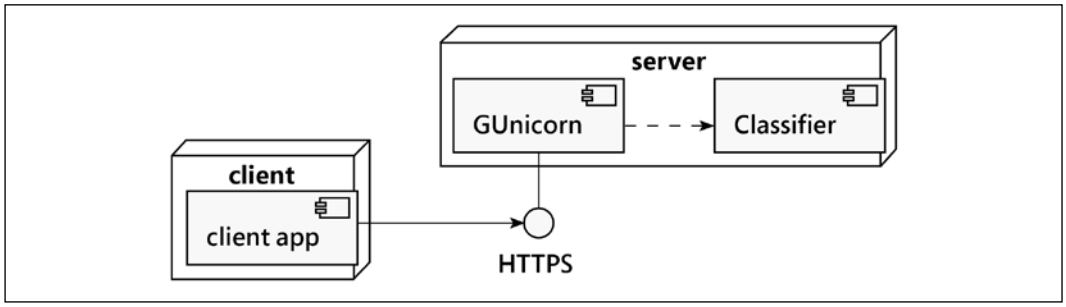

Figure 1.17: Application server diagram

This diagram shows the client and server nodes as three-dimensional "boxes" with "components" installed on them. We've identified three components:

- A Client running the **client app** application. This application connects to the classifier web service and makes RESTful requests. It might be a website, written in JavaScript. It might be a mobile application, written in Kotlin or Swift. All of these frontends have a common **HTTPS** connection to our web server. This secure connection requires some configuration of certificates and encryption key pairs.

- The **GUnicorn** web server. This server can handle a number of details of web service requests, including the important HTTPS protocol. See `https://docs.gunicorn.org/en/stable/index.html` for details.
- Our **Classifier** application. From this view, the complexities have been omitted, and the entire `Classifier` package is reduced to a small component in a larger web services framework. This could be built using the Flask framework.

Of these components, the Client's **client app** is not part of the work being done to develop the classifier. We've included this to illustrate the context, but we're not really going to be building it.

We've used a dotted dependency arrow to show that our `Classifier` application is a dependency from the web server. **GUnicorn** will import our web server object and use it to respond to requests.

Now that we've sketched out the application, we can consider writing some code. As we write, it helps to keep the diagrams up-to-date. Sometimes, they serve as a handy roadmap in a wilderness of code.

Conclusion

There are several key concepts in this case study:

1. Software applications can be rather complicated. There are five views to depict the users, the data, the processing, the components to be built, and the target physical implementation.
2. Mistakes will be made. This overview has some gaps in it. It's important to move forward with partial solutions. One of Python's advantages is the ability to build software quickly, meaning we're not deeply invested in bad ideas. We can (and should) remove and replace code quickly.
3. Be open to extensions. After we implement this, we'll see that setting the *k* parameter is a tedious exercise. An important next step is to automate tuning using a grid search tuning algorithm. It's often helpful to set these things aside and get something that works first, then extend working software later to add this helpful feature.
4. Try to assign clear responsibilities to each class. This has been moderately successful, and some responsibilities are vague or omitted entirely. We'll revisit this as we expand this initial analysis into implementation details.

In later chapters, we'll dive more deeply into these various topics. Because our intent is to present realistic work, this will involve rework. Some design decisions may be revised as the reader is exposed to more and more of the available objected-oriented programming techniques in Python. Additionally, some parts of the solution will evolve as our understanding of design choices and the problem itself evolves. Rework based on lessons learned is a consequence of an agile approach to development.

Recall

Some key points in this chapter:

- Analyzing problem requirements in an object-oriented context
- How to draw **Unified Modeling Language** (**UML**) diagrams to communicate how the system works
- Discussing object-oriented systems using the correct terminology and jargon
- Understanding the distinction between class, object, attribute, and behavior
- Some OO design techniques are used more than others. In our case study example, we focused on a few:
    - Encapsulating features into classes
    - Inheritance to extend a class with new features
    - Composition to build a class from component objects

Exercises

This is a practical book. As such, we're not assigning a bunch of fake object-oriented analysis problems to create designs for you to analyze and design. Instead, we want to give you some ideas that you can apply to your own projects. If you have previous object-oriented experience, you won't need to put much effort into this chapter. However, they are useful mental exercises if you've been using Python for a while, but have never really cared about all that class stuff.

First, think about a recent programming project you've completed. Identify the most prominent object in the design. Try to think of as many attributes for this object as possible. Did it have the following: Color? Weight? Size? Profit? Cost? Name? ID number? Price? Style?

Think about the attribute types. Were they primitives or classes? Were some of those attributes actually behaviors in disguise? Sometimes, what looks like data is actually calculated from other data on the object, and you can use a method to do those calculations. What other methods or behaviors did the object have? Which objects called those methods? What kinds of relationships did they have with this object?

Now, think about an upcoming project. It doesn't matter what the project is; it might be a fun free-time project or a multi-million-dollar contract. It doesn't have to be a complete application; it could just be one subsystem. Perform a basic object-oriented analysis. Identify the requirements and the interacting objects. Sketch out a class diagram featuring the highest level of abstraction on that system. Identify the major interacting objects. Identify minor supporting objects. Go into detail for the attributes and methods of some of the most interesting ones. Take different objects to different levels of abstraction. Look for places where you can use inheritance or composition. Look for places where you should avoid inheritance.

The goal is not to design a system (although you're certainly welcome to do so if inclination meets both ambition and available time). The goal is to think about object-oriented design. Focusing on projects that you have worked on, or are expecting to work on in the future, simply makes it real.

Lastly, visit your favorite search engine and look up some tutorials on UML. There are dozens, so find one that suits your preferred method of study. Sketch some class diagrams or a sequence diagram for the objects you identified earlier. Don't get too hung up on memorizing the syntax (after all, if it is important, you can always look it up again); just get a feel for the language. Something will stay lodged in your brain, and it can make communicating a bit easier if you can quickly sketch a diagram for your next OOP discussion.

Summary

In this chapter, we took a whirlwind tour through the terminology of the object-oriented paradigm, focusing on object-oriented design. We can separate different objects into a taxonomy of different classes and describe the attributes and behaviors of those objects via the class interface. Abstraction, encapsulation, and information hiding are highly-related concepts. There are many different kinds of relationships between objects, including association, composition, and inheritance. UML syntax can be useful for fun and communication.

In the next chapter, we'll explore how to implement classes and methods in Python.

2

Objects in Python

We have a design in hand and are ready to turn that design into a working program! Of course, it doesn't usually happen this way. We'll be seeing examples and hints for good software design throughout the book, but our focus is object-oriented programming. So, let's have a look at the Python syntax that allows us to create object-oriented software.

After completing this chapter, we will understand the following:

- Python's type hints
- Creating classes and instantiating objects in Python
- Organizing classes into packages and modules
- How to suggest that people don't clobber an object's data, invalidating the internal state
- Working with third-party packages available from the Python Package Index, PyPI

This chapter will also continue our case study, moving into the design of some of the classes.

Introducing type hints

Before we can look closely at creating classes, we need to talk a little bit about what a class is and how we're sure we're using it correctly. The central idea here is that everything in Python is an object.

When we write literal values like `"Hello, world!"` or `42`, we're actually creating instances of built-in classes. We can fire up interactive Python and use the built-in `type()` function on the class that defines the properties of these objects:

```
>>> type("Hello, world!")
<class 'str'>
>>> type(42)
<class 'int'>
```

The point of *object-oriented* programming is to solve a problem via the interactions of objects. When we write `6*7`, the multiplication of the two objects is handled by a method of the built-in `int` class. For more complex behaviors, we'll often need to write unique, new classes.

Here are the first two core rules of how Python objects work:

- Everything in Python is an object
- Every object is defined by being an instance of at least one class

These rules have many interesting consequences. A class definition we write, using the `class` statement, creates a new object of class `type`. When we create an **instance** of a class, the class object will be used to create and initialize the instance object.

What's the distinction between class and type? The `class` statement lets us define new types. Because the `class` statement is what we use, we'll call them classes throughout the text. See *Python objects, types, classes, and instances - a glossary* by Eli Bendersky: `https://eli.thegreenplace.net/2012/03/30/python-objects-types-classes-and-instances-a-glossary` for this useful quote:

> *"The terms "class" and "type" are an example of two names referring to the same concept."*

We'll follow common usage and call the annotations **type hints**.

There's another important rule:

- A variable is a reference to an object. Think of a yellow sticky note with a name scrawled on it, slapped on a thing.

This doesn't seem too earth-shattering but it's actually pretty cool. It means the type information – what an object is – is defined by the class(es) associated with the object. This type information is not attached to the *variable* in any way. This leads to code like the following being valid but very confusing Python:

```
>>> a_string_variable = "Hello, world!"
>>> type(a_string_variable)
<class 'str'>
>>> a_string_variable = 42
>>> type(a_string_variable)
<class 'int'>
```

We created an object using a built-in class, `str`. We assigned a long name, `a_string_variable`, to the object. Then, we created an object using a different built-in class, `int`. We assigned this object the same name. (The previous string object has no more references and ceases to exist.)

Here are the two steps, shown side by side, showing how the variable is moved from object to object:

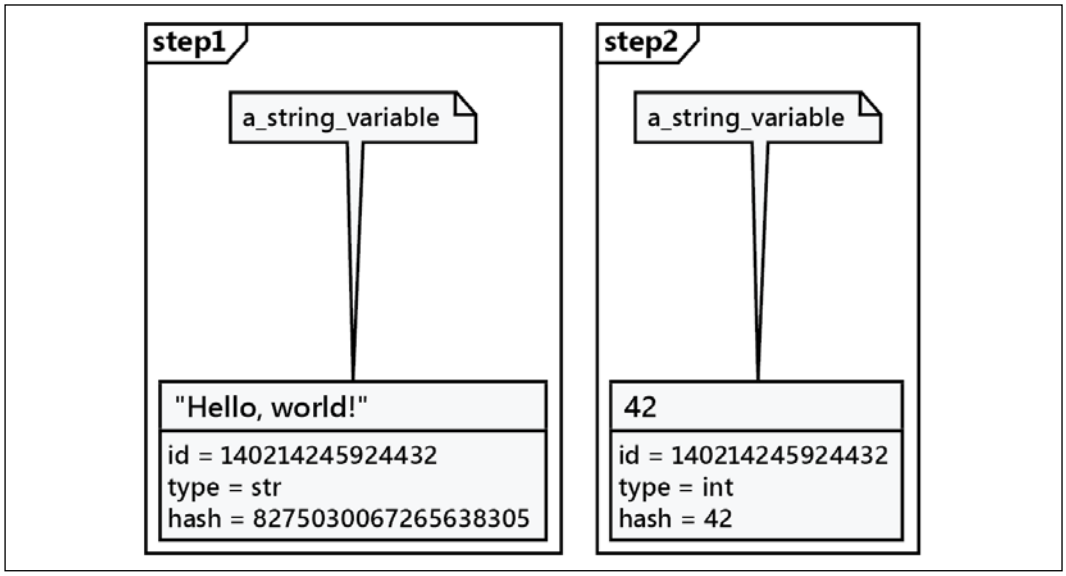

Figure 2.1: Variable names and objects

The various properties are part of the object, not the variable. When we check the type of a variable with `type()`, we see the type of the object the variable currently references. The variable doesn't have a type of its own; it's nothing more than a name. Similarly, asking for the `id()` of a variable shows the ID of the object the variable refers to. So obviously, the name `a_string_variable` is a bit misleading if we assign the name to an integer object.

Type checking

Let's push the relationship between object and type a step further, and look at some more consequences of these rules. Here's a function definition:

```
>>> def odd(n):
...     return n % 2 != 0

>>> odd(3)
True
>>> odd(4)
False
```

This function does a little computation on a parameter variable, `n`. It computes the remainder after division, the modulo. If we divide an odd number by two, we'll have one left over. If we divide an even number by two, we'll have zero left over. This function returns a true value for all odd numbers.

What happens when we fail to provide a number? Well, let's just try it and see (a common way to learn Python!). Entering code at the interactive prompt, we'll get something like this:

```
>>> odd("Hello, world!")
Traceback (most recent call last):
  File "<doctestexamples.md[9]>", line 1, in <module>
odd("Hello, world!")
  File "<doctestexamples.md[6]>", line 2, in odd
    return n % 2 != 0
TypeError: not all arguments converted during string formatting
```

This is an important consequence of Python's super-flexible rules: nothing prevents us from doing something silly that may raise an exception. This is an important tip:

Python doesn't prevent us from attempting to use non-existent methods of objects.

In our example, the `%` operator provided by the `str` class doesn't work the same way as the `%` operator provided by the `int` class, raising an exception. For strings, the `%` operator isn't used very often, but it does interpolation: `"a=%d" % 113` computes a string `'a=113'`; if there's no format specification like `%d` on the left side, the exception is a `TypeError`. For integers, it's the remainder in division: `355 % 113` returns an integer, `16`.

This flexibility reflects an explicit trade-off favoring ease of use over sophisticated prevention of potential problems. This allows a person to use a variable name with little mental overhead.

Python's internal operators check that operands meet the requirements of the operator. The function definition we wrote, however, does not include any runtime type checking. Nor do we want to add code for runtime type checking. Instead, we use tools to examine code as part of testing. We can provide annotations, called **type hints**, and use tools to examine our code for consistency among the type hints.

First, we'll look at the annotations. In a few contexts, we can follow a variable name with a colon, `:`, and a type name. We can do this in the parameters to functions (and methods). We can also do this in assignment statements. Further, we can also add `->` syntax to a function (or a class method) definition to explain the expected return type.

Here's how type hints look:

```
>>> def odd(n: int) -> bool:
...     return n % 2 != 0
```

We've added two type hints to our `odd()` little function definition. We've specified that argument values for the `n` parameter should be integers. We've also specified that the result will be one of the two values of the Boolean type.

While the hints consume some storage, they have no runtime impact. Python politely ignores these hints; this means they're optional. People reading your code, however, will be more than delighted to see them. They are a great way to inform the reader of your intent. You can omit them while you're learning, but you'll love them when you go back to expand something you wrote earlier.

The ***mypy*** tool is commonly used to check the hints for consistency. It's not built into Python, and requires a separate download and install. We'll talk about virtual environments and installation of tools later in this chapter, in the *Third-party libraries* section. For now, you can use `python -m pip install mypy` or `conda install mypy` if you're using ***the conda tool***.

Let's say we had a file, `bad_hints.py`, in a `src` directory, with these two functions and a few lines to call the `main()` function:

```
def odd(n: int) -> bool:
    return n % 2 != 0

def main():
    print(odd("Hello, world!"))

if __name__ == "__main__":
    main()
```

When we run the `mypy` command at the OS's terminal prompt:

```
% mypy -strict src/bad_hints.py
```

The ***mypy*** tool is going to spot a bunch of potential problems, including at least these:

```
src/bad_hints.py:12: error: Function is missing a return type
annotation
src/bad_hints.py:12: note: Use "-> None" if function does not return a
value
src/bad_hints.py:13: error: Argument 1 to "odd" has incompatible type
"str"; expected "int"
```

The `def main():` statement is on *line 12* of our example because our file has a pile of comments not shown above. For your version, the error might be on *line 1*. Here are the two problems:

- The `main()` function doesn't have a return type; ***mypy*** suggests including `-> None` to make the absence of a return value perfectly explicit.
- More important is *line 13*: the code will try to evaluate the `odd()` function using a `str` value. This doesn't match the type hint for `odd()` and indicates another possible error.

Most of the examples in this book will have type hints. We think they're always helpful, especially in a pedagogical context, even though they're optional. Because most of Python is generic with respect to type, there are a few cases where Python behavior is difficult to describe via a succinct, expressive hint. We'll steer clear of these edge cases in this book.

Python Enhancement Proposal (PEP) 585 covers some new language features to make type hints a bit simpler. We've used ***mypy*** version 0.812 to test all of the examples in this book. Any older version will encounter problems with some of the newer syntax and annotation techniques.

Now that we've talked about how parameters and attributes are described with type hints, let's actually build some classes.

Creating Python classes

We don't have to write much Python code to realize that Python is a very *clean* language. When we want to do something, we can just do it, without having to set up a bunch of prerequisite code. The ubiquitous *hello world* in Python, as you've likely seen, is only one line.

Similarly, the simplest class in Python 3 looks like this:

```
class MyFirstClass:
    pass
```

There's our first object-oriented program! The class definition starts with the `class` keyword. This is followed by a name (of our choice) identifying the class and is terminated with a colon.

The class name must follow standard Python variable naming rules (it must start with a letter or underscore, and can only be comprised of letters, underscores, or numbers). In addition, the Python style guide (search the web for *PEP 8*) recommends that classes should be named using what PEP 8 calls **CapWords** notation (start with a capital letter; any subsequent words should also start with a capital).

The class definition line is followed by the class contents, indented. As with other Python constructs, indentation is used to delimit the classes, rather than braces, keywords, or brackets, as many other languages use. Also, in line with the style guide, use four spaces for indentation unless you have a compelling reason not to (such as fitting in with somebody else's code that uses tabs for indents).

Since our first class doesn't actually add any data or behaviors, we simply use the `pass` keyword on the second line as a placeholder to indicate that no further action needs to be taken.

We might think there isn't much we can do with this most basic class, but it does allow us to instantiate objects of that class. We can load the class into the Python 3 interpreter, so we can interactively play with it. To do this, save the class definition mentioned earlier in a file named `first_class.py` and then run the `python -i first_class.py` command. The `-i` argument tells Python to *run the code and then drop to the interactive interpreter*. The following interpreter session demonstrates a basic interaction with this class:

```
>>> a = MyFirstClass()
>>> b = MyFirstClass()
>>> print(a)
<__main__.MyFirstClass object at 0xb7b7faec>
>>> print(b)
<__main__.MyFirstClass object at 0xb7b7fbac>
```

This code instantiates two objects from the new class, assigning the object variable names `a` and `b`. Creating an instance of a class is a matter of typing the class name, followed by a pair of parentheses. It looks much like a function call; **calling** a class will create a new object. When printed, the two objects tell us which class they are and what memory address they live at. Memory addresses aren't used much in Python code, but here, they demonstrate that there are two distinct objects involved.

We can see they're distinct objects by using the `is` operator:

```
>>> a is b
False
```

This can help reduce confusion when we've created a bunch of objects and assigned different variable names to the objects.

Adding attributes

Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?

In fact, we don't have to do anything special in the class definition to be able to add attributes. We can set arbitrary attributes on an instantiated object using dot notation. Here's an example:

```
class Point:
    pass

p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

If we run this code, the two `print` statements at the end tell us the new attribute values on the two objects:

```
5 4
3 6
```

This code creates an empty `Point` class with no data or behaviors. Then, it creates two instances of that class and assigns each of those instances `x` and `y` coordinates to identify a point in two dimensions. All we need to do to assign a value to an attribute on an object is use the `<object>.<attribute> = <value>` syntax. This is sometimes referred to as **dot notation**. The value can be anything: a Python primitive, a built-in data type, or another object. It can even be a function or another class!

Creating attributes like this is confusing to the ***mypy*** tool. There's no easy way to include the hints in the `Point` class definition. We can include hints on the assignment statements, like this: `p1.x: float = 5`. In general, there's a much, much better approach to type hints and attributes that we'll examine in the *Initializing the object* section, later in this chapter. First, though, we'll add behaviors to our class definition.

Making it do something

Now, having objects with attributes is great, but object-oriented programming is really about the interaction between objects. We're interested in invoking actions that cause things to happen to those attributes. We have data; now it's time to add behaviors to our classes.

Let's model a couple of actions on our `Point` class. We can start with a **method** called `reset`, which moves the point to the origin (the origin is the place where `x` and `y` are both zero). This is a good introductory action because it doesn't require any parameters:

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

This `print` statement shows us the two zeros on the attributes:

```
0 0
```

In Python, a method is formatted identically to a function. It starts with the `def` keyword, followed by a space, and the name of the method. This is followed by a set of parentheses containing the parameter list (we'll discuss that `self` parameter, sometimes called the instance variable, in just a moment), and terminated with a colon. The next line is indented to contain the statements inside the method. These statements can be arbitrary Python code operating on the object itself and any parameters passed in, as the method sees fit.

We've omitted type hints in the `reset()` method because it's not the most widely used place for hints. We'll look at the best place for hints in the *Initializing the object* section. We'll look a little more at these instance variables, first, and how the `self` variable works.

Talking to yourself

The one difference, syntactically, between methods of classes and functions outside classes is that methods have one required argument. This argument is conventionally named `self`; I've never seen a Python programmer use any other name for this variable (convention is a very powerful thing). There's nothing technically stopping you, however, from calling it `this` or even `Martha`, but it's best to acknowledge the social pressure of the Python community codified in PEP 8 and stick with `self`.

The `self` argument to a method is a reference to the object that the method is being invoked on. The object is an instance of a class, and this is sometimes called the instance variable.

We can access attributes and methods of that object via this variable. This is exactly what we do inside the `reset` method when we set the `x` and `y` attributes of the `self` object.

Pay attention to the difference between a **class** and an **object** in this discussion. We can think of the **method** as a function attached to a class. The `self` parameter refers to a specific instance of the class. When you call the method on two different objects, you are calling the same method twice, but passing two different **objects** as the `self` parameter.

Notice that when we call the `p.reset()` method, we do not explicitly pass the `self` argument into it. Python automatically takes care of this part for us. It knows we're calling a method on the `p` object, so it automatically passes that object, `p`, to the method of the class, `Point`.

For some, it can help to think of a method as a function that happens to be part of a class. Instead of calling the method on the object, we could invoke the function as defined in the class, explicitly passing our object as the `self` argument:

```
>>> p = Point()
>>> Point.reset(p)
>>> print(p.x, p.y)
```

The output is the same as in the previous example because, internally, the exact same process has occurred. This is not really a good programming practice, but it can help to cement your understanding of the `self` argument.

What happens if we forget to include the `self` argument in our class definition? Python will bail with an error message, as follows:

```
>>> class Point:
...     def reset():
...         pass
...
>>> p = Point()
>>> p.reset()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reset() takes 0 positional arguments but 1 was given
```

The error message is not as clear as it could be ("Hey, silly, you forgot to define the method with a `self` parameter" could be more informative). Just remember that when you see an error message that indicates missing arguments, the first thing to check is whether you forgot the `self` parameter in the method definition.

More arguments

How do we pass multiple arguments to a method? Let's add a new method that allows us to move a point to an arbitrary position, not just to the origin. We can also include a method that accepts another `Point` object as input and returns the distance between them:

```
import math

class Point:
    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self) -> None:
        self.move(0, 0)

    def calculate_distance(self, other: "Point") -> float:
        return math.hypot(self.x - other.x, self.y - other.y)
```

We've defined a class with two attributes, `x`, and `y`, and three separate methods, `move()`, `reset()`, and `calculate_distance()`.

The `move()` method accepts two arguments, `x` and `y`, and sets the values on the `self` object. The `reset()` method calls the `move()` method, since a reset is just a move to a specific known location.

The `calculate_distance()` method computes the Euclidean distance between two points. (There are a number of other ways to look at distance. In the *Chapter 3*, *When Objects Are Alike*, case study, we'll look at some alternatives.) For now, we hope you understand the math. The definition is $\sqrt{(x_s - x_o)^2 + (y_s - y_o)^2}$, which is the `math.hypot()` function. In Python we'll use `self.x`, but mathematicians often prefer to write x_s.

Here's an example of using this class definition. This shows how to call a method with arguments: include the arguments inside the parentheses and use the same dot notation to access the method name within the instance. We just picked some random positions to test the methods. The test code calls each method and prints the results on the console:

```
>>> point1 = Point()
>>> point2 = Point()

>>> point1.reset()
>>> point2.move(5, 0)
>>> print(point2.calculate_distance(point1))
5.0
>>> assert point2.calculate_distance(point1) ==
point1.calculate_distance(
...     point2
... )
>>> point1.move(3, 4)
>>> print(point1.calculate_distance(point2))
4.47213595499958
>>> print(point1.calculate_distance(point1))
0.0
```

The `assert` statement is a marvelous test tool; the program will bail if the expression after `assert` evaluates to `False` (or zero, empty, or `None`). In this case, we use it to ensure that the distance is the same regardless of which point called the other point's `calculate_distance()` method. We'll see a lot more use of `assert` in *Chapter 13*, *Testing Object-Oriented Programs*, where we'll write more rigorous tests.

Initializing the object

If we don't explicitly set the `x` and `y` positions on our `Point` object, either using `move` or by accessing them directly, we'll have a broken `Point` object with no real position. What will happen when we try to access it?

Well, let's just try it and see. *Try it and see* is an extremely useful tool for Python study. Open up your interactive interpreter and type away. (Using the interactive prompt is, after all, one of the tools we used to write this book.)

The following interactive session shows what happens if we try to access a missing attribute. If you saved the previous example as a file or are using the examples distributed with the book, you can load it into the Python interpreter with the `python -i more_arguments.py` command:

```
>>> point = Point()
>>> point.x = 5
>>> print(point.x)
5
>>> print(point.y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'y'
```

Well, at least it threw a useful exception. We'll cover exceptions in detail in *Chapter 4*, *Expecting the Unexpected*. You've probably seen them before (especially the ubiquitous `SyntaxError`, which means you typed something incorrectly!). At this point, simply be aware that it means something went wrong.

The output is useful for debugging. In the interactive interpreter, it tells us the error occurred at *line 1*, which is only partially true (in an interactive session, only one statement is executed at a time). If we were running a script in a file, it would tell us the exact line number, making it easy to find the offending code. In addition, it tells us that the error is an `AttributeError`, and gives a helpful message telling us what that error means.

We can catch and recover from this error, but in this case, it feels like we should have specified some sort of default value. Perhaps every new object should be `reset()` by default, or maybe it would be nice if we could force the user to tell us what those positions should be when they create the object.

Interestingly, ***mypy*** can't determine whether `y` is supposed to be an attribute of a `Point` object. Attributes are – by definition – dynamic, so there's no simple list that's part of a class definition. However, Python has some widely followed conventions that can help name the expected set of attributes.

Most object-oriented programming languages have the concept of a **constructor**, a special method that creates and initializes the object when it is created. Python is a little different; it has a constructor and an initializer. The constructor method, `__new__()`, is rarely used unless you're doing something very exotic. So, we'll start our discussion with the much more common initialization method, `__init__()`.

The Python initialization method is the same as any other method, except it has a special name, `__init__`. The leading and trailing double underscores mean this is a special method that the Python interpreter will treat as a special case.

Never name a method of your own with leading and trailing double underscores. It may mean nothing to Python today, but there's always the possibility that the designers of Python will add a function that has a special purpose with that name in the future. When they do, your code will break.

Let's add an initialization function on our `Point` class that requires the user to supply `x` and `y` coordinates when the `Point` object is instantiated:

```
class Point:
    def __init__(self, x: float, y: float) -> None:
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self) -> None:
        self.move(0, 0)

    def calculate_distance(self, other: "Point") -> float:
        return math.hypot(self.x - other.x, self.y - other.y)
```

Constructing a `Point` instance now looks like this:

```
point = Point(3, 5)
print(point.x, point.y)
```

Now, our `Point` object can never go without both `x` and `y` coordinates! If we try to construct a `Point` instance without including the proper initialization parameters, it will fail with a `not enough arguments` error similar to the one we received earlier when we forgot the `self` argument in a method definition.

Most of the time, we put our initialization statements in an `__init__()` function. It's very important to be sure that all of the attributes are initialized in the `__init__()` method. Doing this helps the ***mypy*** tool by providing all of the attributes in one obvious place. It helps people reading your code, also; it saves them from having to read the whole application to find mysterious attributes set outside the class definition.

While they're optional, it's generally helpful to include type annotations on the method parameters and result values. After each parameter name, we've included the expected type of each value. At the end of the definition, we've included the two-character `->` operator and the type returned by the method.

Type hints and defaults

As we've noted a few times now, hints are optional. They don't do anything at runtime. There are tools, however, that can examine the hints to check for consistency. The ***mypy*** tool is widely used to check type hints.

If we don't want to make the two arguments required, we can use the same syntax Python functions use to provide default arguments. The keyword argument syntax appends an equals sign after each variable name. If the calling object does not provide this argument, then the default argument is used instead. The variables will still be available to the function, but they will have the values specified in the argument list. Here's an example:

```
class Point:
    def __init__(self, x: float = 0, y: float = 0) -> None:
        self.move(x, y)
```

The definitions for the individual parameters can get long, leading to very long lines of code. In some examples, you'll see this single logical line of code expanded to multiple physical lines. This relies on the way Python combines physical lines to match `()`'s. We might write this when the line gets long:

```
class Point:
    def __init__(
        self,
        x: float = 0,
        y: float = 0
    ) -> None:
        self.move(x, y)
```

This style isn't used very often, but it's valid and keeps the lines shorter and easier to read.

The type hints and defaults are handy, but there's even more we can do to provide a class that's easy to use and easy to extend when new requirements arise. We'll add documentation in the form of docstrings.

Explaining yourself with docstrings

Python can be an extremely easy-to-read programming language; some might say it is self-documenting. However, when carrying out object-oriented programming, it is important to write API documentation that clearly summarizes what each object and method does. Keeping documentation up to date is difficult; the best way to do it is to write it right into our code.

Python supports this through the use of **docstrings**. Each class, function, or method header can have a standard Python string as the first indented line inside the definition (the line that ends in a colon).

Docstrings are Python strings enclosed within apostrophes (') or quotation marks ("). Often, docstrings are quite long and span multiple lines (the style guide suggests that the line length should not exceed 80 characters), which can be formatted as multi-line strings, enclosed in matching triple apostrophe (''') or triple quote (""") characters.

A docstring should clearly and concisely summarize the purpose of the class or method it is describing. It should explain any parameters whose usage is not immediately obvious, and is also a good place to include short examples of how to use the API. Any caveats or problems an unsuspecting user of the API should be aware of should also be noted.

One of the best things to include in a docstring is a concrete example. Tools like **doctest** can locate and confirm these examples are correct. All the examples in this book are checked with the doctest tool.

To illustrate the use of docstrings, we will end this section with our completely documented `Point` class:

```
class Point:
    """
    Represents a point in two-dimensional geometric coordinates

    >>> p_0 = Point()
    >>> p_1 = Point(3, 4)
    >>> p_0.calculate_distance(p_1)
    5.0
    """

    def __init__(self, x: float = 0, y: float = 0) -> None:
        """
        Initialize the position of a new point. The x and y
```

```
        coordinates can be specified. If they are not, the
        point defaults to the origin.

        :param x: float x-coordinate
        :param y: float x-coordinate
        """
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        """
        Move the point to a new location in 2D space.

        :param x: float x-coordinate
        :param y: float x-coordinate
        """
        self.x = x
        self.y = y

    def reset(self) -> None:
        """
        Reset the point back to the geometric origin: 0, 0
        """
        self.move(0, 0)

    def calculate_distance(self, other: "Point") -> float:
        """
        Calculate the Euclidean distance from this point
        to a second point passed as a parameter.

        :param other: Point instance
        :return: float distance
        """
        return math.hypot(self.x - other.x, self.y - other.y)
```

Try typing or loading (remember, it's `python -i point.py`) this file into the interactive interpreter. Then, enter `help(Point)<enter>` at the Python prompt.

You should see nicely formatted documentation for the class, as shown in the following output:

```
Help on class Point in module point_2:

class Point(builtins.object)
 |  Point(x: float = 0, y: float = 0) -> None
 |
 |  Represents a point in two-dimensional geometric coordinates
 |
 |  >>> p_0 = Point()
 |  >>> p_1 = Point(3, 4)
 |  >>> p_0.calculate_distance(p_1)
 |  5.0
 |
 |  Methods defined here:
 |
 |  __init__(self, x: float = 0, y: float = 0) -> None
 |      Initialize the position of a new point. The x and y
 |      coordinates can be specified. If they are not, the
 |      point defaults to the origin.
 |
 |      :param x: float x-coordinate
 |      :param y: float x-coordinate
 |
 |  calculate_distance(self, other: 'Point') -> float
 |      Calculate the Euclidean distance from this point
 |      to a second point passed as a parameter.
 |
 |      :param other: Point instance
 |      :return: float distance
 |
 |  move(self, x: float, y: float) -> None
 |      Move the point to a new location in 2D space.
 |
 |      :param x: float x-coordinate
 |      :param y: float x-coordinate
 |
 |  reset(self) -> None
```

```
|      Reset the point back to the geometric origin: 0, 0
|
|  ----------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

Not only is our documentation every bit as polished as the documentation for built-in functions, but we can run `python -m doctest point_2.py` to confirm the example shown in the docstring.

Further, we can run ***mypy*** to check the type hints, also. Use `mypy --strict src/*.py` to check all of the files in the `src` folder. If there are no problems, the ***mypy*** application doesn't produce any output. (Remember, ***mypy*** is not part of the standard installation, so you'll need to add it. Check the preface for information on extra packages that need to be installed.)

Modules and packages

Now we know how to create classes and instantiate objects. You don't need to write too many classes (or non-object-oriented code, for that matter) before you start to lose track of them. For small programs, we generally put all our classes into one file and add a little script at the end of the file to start them interacting. However, as our projects grow, it can become difficult to find the one class that needs to be edited among the many classes we've defined. This is where **modules** come in. Modules are Python files, nothing more. The single file in our small program is a module. Two Python files are two modules. If we have two files in the same folder, we can load a class from one module for use in the other module.

The Python module name is the file's *stem*; the name without the `.py` suffix. A file named `model.py` is a module named `model`. Module files are found by searching a path that includes the local directory and the installed packages.

The `import` statement is used for importing modules or specific classes or functions from modules. We've already seen an example of this in our `Point` class in the previous section. We used the `import` statement to get Python's built-in `math` module and use its `hypot()` function in the `distance` calculation. Let's start with a fresh example.

If we are building an e-commerce system, we will likely be storing a lot of data in a database. We can put all the classes and functions related to database access into a separate file (we'll call it something sensible: `database.py`). Then, our other modules (for example, customer models, product information, and inventory) can import classes from the `database` module in order to access the database.

Let's start with a module called `database`. It's a file, `database.py`, containing a class called `Database`. A second module called `products` is responsible for product-related queries. The classes in the `products` module need to instantiate the `Database` class from the `database` module so that they can execute queries on the product table in the database.

There are several variations on the `import` statement syntax that can be used to access the `Database` class. One variant is to import the module as a whole:

```
>>> import database
>>> db = database.Database("path/to/data")
```

This version imports the `database` module, creating a `database` namespace. Any class or function in the `database` module can be accessed using the `database.<something>` notation.

Alternatively, we can import just the one class we need using the `from...import` syntax:

```
>>> from database import Database
>>> db = Database("path/to/data")
```

This version imported only the `Database` class from the `database` module. When we have a few items from a few modules, this can be a helpful simplification to avoid using longer, fully qualified names like `database.Database`. When we import a number of items from a number of different modules, this can be a potential source of confusion when we omit the qualifiers.

If, for some reason, `products` already has a class called `Database`, and we don't want the two names to be confused, we can rename the class when used inside the `products` module:

```
>>> from database import Database as DB
>>> db = DB("path/to/data")
```

We can also import multiple items in one statement. If our `database` module also contains a `Query` class, we can import both classes using the following code:

```
from database import Database, Query
```

We can import all classes and functions from the `database` module using this syntax:

```
from database import *
```

Don't do this. Most experienced Python programmers will tell you that you should never use this syntax (a few will tell you there are some very specific situations where it is useful, but we can disagree). One way to learn why to avoid this syntax is to use it and try to understand your code two years later. We can save some time and two years of poorly written code with a quick explanation now!

We've got several reasons for avoiding this:

- When we explicitly import the `database` class at the top of our file using `from database import Database`, we can easily see where the `Database` class comes from. We might use `db = Database()` 400 lines later in the file, and we can quickly look at the imports to see where that `Database` class came from. Then, if we need clarification as to how to use the `Database` class, we can visit the original file (or import the module in the interactive interpreter and use the `help(database.Database)` command). However, if we use the `from database import *` syntax, it takes a lot longer to find where that class is located. Code maintenance becomes a nightmare.
- If there are conflicting names, we're doomed. Let's say we have two modules, both of which provide a class named `Database`. Using `from module_1 import *` and `from module_2 import *` means the second import statement overwrites the `Database` name created by the first import. If we used `import module_1` and `import module_2`, we'd use the module names as qualifiers to disambiguate `module_1.Database` from `module_2.Database`.
- In addition, most code editors are able to provide extra functionality, such as reliable code completion, the ability to jump to the definition of a class, or inline documentation, if normal imports are used. The `import *` syntax can hamper their ability to do this reliably.

- Finally, using the `import *` syntax can bring unexpected objects into our local namespace. Sure, it will import all the classes and functions defined in the module being imported from, but unless a special `__all__` list is provided in the module, this `import` will also import any classes or modules that were themselves imported into that file!

Every name used in a module should come from a well-specified place, whether it is defined in that module, or explicitly imported from another module. There should be no magic variables that seem to come out of thin air. We should *always* be able to immediately identify where the names in our current namespace originated. We promise that if you use this evil syntax, you will one day have extremely frustrating moments of *where on earth can this class be coming from?*

For fun, try typing `import this` into your interactive interpreter. It prints a nice poem (with a couple of inside jokes) summarizing some of the idioms that Pythonistas tend to practice. Specific to this discussion, note the line "Explicit is better than implicit." Explicitly importing names into your namespace makes your code much easier to navigate than the implicit `from module import *` syntax.

Organizing modules

As a project grows into a collection of more and more modules, we may find that we want to add another level of abstraction, some kind of nested hierarchy on our modules' levels. However, we can't put modules inside modules; one file can hold only one file after all, and modules are just files.

Files, however, can go in folders, and so can modules. A **package** is a collection of modules in a folder. The name of the package is the name of the folder. We need to tell Python that a folder is a package to distinguish it from other folders in the directory. To do this, place a (normally empty) file in the folder named `__init__.py`. If we forget this file, we won't be able to import modules from that folder.

Let's put our modules inside an `ecommerce` package in our working folder, which will also contain a `main.py` file to start the program. Let's additionally add another package inside the `ecommerce` package for various payment options.

We need to exercise some caution in creating deeply nested packages. The general advice in the Python community is "flat is better than nested." In this example, we need to create a nested package because there are some common features to all of the various payment alternatives.

The folder hierarchy will look like this, rooted under a directory in the project folder, commonly named `src`:

```
src/
 +-- main.py
 +-- ecommerce/
     +-- __init__.py
     +-- database.py
     +-- products.py
     +-- payments/
     |   +-- __init__.py
     |   +-- common.py
     |   +-- square.py
     |   +-- stripe.py
     +-- contact/
         +-- __init__.py
         +-- email.py
```

The `src` directory will be part of an overall project directory. In addition to `src`, the project will often have directories with names like `docs` and `tests`. It's common for the project parent directory to also have configuration files for tools like ***mypy*** among others. We'll return to this in *Chapter 13*, *Testing Object-Oriented Programs*.

When importing modules or classes between packages, we have to be cautious about the structure of our packages. In Python 3, there are two ways of importing modules: absolute imports and relative imports. We'll look at each of them separately.

Absolute imports

Absolute imports specify the complete path to the module, function, or class we want to import. If we need access to the `Product` class inside the `products` module, we could use any of these syntaxes to perform an absolute import:

```
>>> import ecommerce.products
>>> product = ecommerce.products.Product("name1")
```

Or, we could specifically import a single class definition from the module within a package:

```
>>> from ecommerce.products import Product
>>> product = Product("name2")
```

Or, we could import an entire module from the containing package:

```
>>> from ecommerce import products
>>> product = products.Product("name3")
```

The `import` statements use the period operator to separate packages or modules. A package is a namespace that contains module names, much in the way an object is a namespace containing attribute names.

These statements will work from any module. We could instantiate a `Product` class using this syntax in `main.py`, in the `database` module, or in either of the two payment modules. Indeed, assuming the packages are available to Python, it will be able to import them. For example, the packages can also be installed in the Python `site-packages` folder, or the `PYTHONPATH` environment variable could be set to tell Python which folders to search for packages and modules it is going to import.

With these choices, which syntax do we choose? It depends on your audience and the application at hand. If there are dozens of classes and functions inside the `products` module that we want to use, we'd generally import the module name using the `from ecommerce import products` syntax, and then access the individual classes using `products.Product`. If we only need one or two classes from the `products` module, we can import them directly using the `from ecommerce.products import Product` syntax. It's important to write whatever makes the code easiest for others to read and extend.

Relative imports

When working with related modules inside a deeply nested package, it seems kind of redundant to specify the full path; we know what our parent module is named. This is where **relative imports** come in. Relative imports identify a class, function, or module as it is positioned relative to the current module. They only make sense inside module files, and, further, they only make sense where there's a complex package structure.

For example, if we are working in the `products` module and we want to import the `Database` class from the `database` module next to it, we could use a relative import:

```
from .database import Database
```

The period in front of `database` says *use the database module inside the current package*. In this case, the current package is the package containing the `products.py` file we are currently editing, that is, the `ecommerce` package.

If we were editing the `stripe` module inside the `ecommerce.payments` package, we would want, for example, to *use the database package inside the parent package* instead. This is easily done with two periods, as shown here:

```
from ..database import Database
```

We can use more periods to go further up the hierarchy, but at some point, we have to acknowledge that we have too many packages. Of course, we can also go down one side and back up the other. The following would be a valid import from the `ecommerce.contact` package containing an `email` module if we wanted to import the `send_mail` function into our `payments.stripe` module:

```
from ..contact.email import send_mail
```

This import uses two periods indicating *the parent of the payments.stripe package*, and then uses the normal `package.module` syntax to go back down into the `contact` package to name the `email` module.

Relative imports aren't as useful as they might seem. As mentioned earlier, the *Zen of Python* (you can read it when you run `import this`) suggests "flat is better than nested". Python's standard library is relatively flat, with few packages and even fewer nested packages. If you're familiar with Java, the packages are deeply nested, something the Python community likes to avoid. A relative import is needed to solve a specific problem where module names are reused among packages. They can be helpful in a few cases. Needing more than two dots to locate a common parent-of-a-parent package suggests the design should be flattened out.

Packages as a whole

We can import code that appears to come directly from a package, as opposed to a module inside a package. As we'll see, there's a module involved, but it has a special name, so it's hidden. In this example, we have an `ecommerce` package containing two module files named `database.py` and `products.py`. The `database` module contains a `db` variable that is accessed from a lot of places. Wouldn't it be convenient if this could be imported as `from ecommerce import db` instead of `from ecommerce.database import db`?

Remember the `__init__.py` file that defines a directory as a package? This file can contain any variable or class declarations we like, and they will be available as part of the package. In our example, if the `ecommerce/__init__.py` file contained the following line:

```
from .database import db
```

We could then access the `db` attribute from `main.py` or any other file using the following import:

```
from ecommerce import db
```

It might help to think of the `ecommerce/__init__.py` file as if it were the `ecommerce.py` file. It lets us view the `ecommerce` package as having a module protocol as well as a package protocol. This can also be useful if you put all your code in a single module and later decide to break it up into a package of modules. The `__init__.py` file for the new package can still be the main point of contact for other modules using it, but the code can be internally organized into several different modules or subpackages.

We recommend not putting much code in an `__init__.py` file, though. Programmers do not expect actual logic to happen in this file, and much like with `from x import *`, it can trip them up if they are looking for the declaration of a particular piece of code and can't find it until they check `__init__.py`.

After looking at modules in general, let's dive into what should be inside a module. The rules are flexible (unlike other languages). If you're familiar with Java, you'll see that Python gives you some freedom to bundle things in a way that's meaningful and informative.

Organizing our code in modules

The Python module is an important focus. Every application or web service has at least one module. Even a seemingly "simple" Python script is a module. Inside any one module, we can specify variables, classes, or functions. They can be a handy way to store the global state without namespace conflicts. For example, we have been importing the `Database` class into various modules and then instantiating it, but it might make more sense to have only one `database` object globally available from the `database` module. The `database` module might look like this:

```
class Database:
    """The Database Implementation"""

    def __init__(self, connection: Optional[str] = None) -> None:
        """Create a connection to a database."""
        pass

database = Database("path/to/data")
```

Then we can use any of the import methods we've discussed to access the `database` object, for example:

```
from ecommerce.database import database
```

A problem with the preceding class is that the `database` object is created immediately when the module is first imported, which is usually when the program starts up. This isn't always ideal, since connecting to a database can take a while, slowing down startup, or the database connection information may not yet be available because we need to read a configuration file. We could delay creating the database until it is actually needed by calling an `initialize_database()` function to create a module-level variable:

```
db: Optional[Database] = None

def initialize_database(connection: Optional[str] = None) -> None:
    global db
    db = Database(connection)
```

The `Optional[Database]` type hint signals to the ***mypy*** tool that this may be `None` or it may have an instance of the `Database` class. The `Optional` hint is defined in the `typing` module. This hint can be handy elsewhere in our application to make sure we confirm that the value for the `database` variable is not `None`.

The `global` keyword tells Python that the database variable inside `initialize_database()` is the module-level variable, outside the function. If we had not specified the variable as global, Python would have created a new local variable that would be discarded when the function exits, leaving the module-level value unchanged.

We need to make one additional change. We need to import the `database` module as a whole. We can't import the `db` object from inside the module; it might not have been initialized. We need to be sure `database.initialize_database()` is called before `db` will have a meaningful value. If we wanted direct access to the database object, we'd use `database.db`.

A common alternative is a function that returns the current database object. We could import this function everywhere we needed access to the database:

```
def get_database(connection: Optional[str] = None) -> Database:
    global db
    if not db:
        db = Database(connection)
    return db
```

As these examples illustrate, all module-level code is executed immediately at the time it is imported. The `class` and `def` statements create code objects to be executed later when the function is called. This can be a tricky thing for scripts that perform execution, such as the main script in our e-commerce example. Sometimes, we write a program that does something useful, and then later find that we want to import a function or class from that module into a different program. However, as soon as we import it, any code at the module level is immediately executed. If we are not careful, we can end up running the first program when we really only meant to access a couple of functions inside that module.

To solve this, we should always put our startup code in a function (conventionally, called `main()`) and only execute that function when we know we are running the module as a script, but not when our code is being imported from a different script. We can do this by **guarding** the call to `main` inside a conditional statement, demonstrated as follows:

```
class Point:
    """
    Represents a point in two-dimensional geometric coordinates.
    """
    pass


def main() -> None:
    """
    Does the useful work.

    >>> main()
    p1.calculate_distance(p2)=5.0
    """
    p1 = Point()
    p2 = Point(3, 4)
    print(f"{p1.calculate_distance(p2)=}")


if __name__ == "__main__":
    main()
```

The `Point` class (and the `main()` function) can be reused without worry. We can import the contents of this module without any surprising processing happening. When we run it as a main program, however, it executes the `main()` function.

This works because every module has a `__name__` special variable (remember, Python uses double underscores for special variables, such as a class' `__init__` method) that specifies the name of the module when it was imported. When the module is executed directly with `python module.py`, it is never imported, so the `__name__` is arbitrarily set to the `"__main__"` string.

Make it a policy to wrap all your scripts in an `if __name__ == "__main__":` test, just in case you write a function that you may want to be imported by other code at some point in the future.

So, methods go in classes, which go in modules, which go in packages. Is that all there is to it?

Actually, no. This is the typical order of things in a Python program, but it's not the only possible layout. Classes can be defined anywhere. They are typically defined at the module level, but they can also be defined inside a function or method, like this:

```
from typing import Optional

class Formatter:
    def format(self, string: str) -> str:
        pass

def format_string(string: str, formatter: Optional[Formatter] = None)
-> str:
    """
    Format a string using the formatter object, which
    is expected to have a format() method that accepts
    a string.
    """

    class DefaultFormatter(Formatter):
        """Format a string in title case."""

        def format(self, string: str) -> str:
            return str(string).title()

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)
```

We've defined a `Formatter` class as an abstraction to explain what a formatter class needs to have. We haven't used the abstract base class (abc) definitions (we'll look at these in detail in *Chapter 6*, *Abstract Base Classes and Operator Overloading*). Instead, we've provided the method with no useful body. It has a full suite of type hints, to make sure ***mypy*** has a formal definition of our intent.

Within the `format_string()` function, we created an internal class that is an extension of the `Formatter` class. This formalizes the expectation that our class inside the function has a specific set of methods. This connection between the definition of the `Formatter` class, the `formatter` parameter, and the concrete definition of the `DefaultFormatter` class assures that we haven't accidentally forgotten something or added something.

We can execute this function like this:

```
>>> hello_string = "hello world, how are you today?"
>>> print(f" input: {hello_string}")
 input: hello world, how are you today?
>>> print(f"output: {format_string(hello_string)}")
output: Hello World, How Are You Today?
```

The `format_string` function accepts a string and optional `Formatter` object and then applies the formatter to that string. If no `Formatter` instance is supplied, it creates a formatter of its own as a local class and instantiates it. Since it is created inside the scope of the function, this class cannot be accessed from anywhere outside of that function. Similarly, functions can be defined inside other functions as well; in general, any Python statement can be executed at any time.

These inner classes and functions are occasionally useful for one-off items that don't require or deserve their own scope at the module level, or only make sense inside a single method. However, it is not common to see Python code that frequently uses this technique.

We've seen how to create classes and how to create modules. With these core techniques, we can start thinking about writing useful, helpful software to solve problems. When the application or service gets big, though, we often have boundary issues. We need to be sure that objects respect each other's privacy and avoid confusing entanglements that make complex software into a spaghetti bowl of interrelationships. We'd prefer each class to be a nicely encapsulated ravioli. Let's look at another aspect of organizing our software to create a good design.

Who can access my data?

Most object-oriented programming languages have a concept of **access control**. This is related to abstraction. Some attributes and methods on an object are marked private, meaning only that object can access them. Others are marked protected, meaning only that class and any subclasses have access. The rest are public, meaning any other object is allowed to access them.

Python doesn't do this. Python doesn't really believe in enforcing laws that might someday get in your way. Instead, it provides unenforced guidelines and best practices. Technically, all methods and attributes on a class are publicly available. If we want to suggest that a method should not be used publicly, we can put a note in docstrings indicating that the method is meant for internal use only (preferably, with an explanation of how the public-facing API works!).

We often remind each other of this by saying "We're all adults here." There's no need to declare a variable as private when we can all see the source code.

By convention, we generally prefix an internal attribute or method with an underscore character, `_`. Python programmers will understand a leading underscore name to mean *this is an internal variable, think three times before accessing it directly*. But there is nothing inside the interpreter to stop them from accessing it if they think it is in their best interest to do so. Because, if they think so, why should we stop them? We may not have any idea what future uses our classes might be put to, and it may be removed in a future release. It's a pretty clear warning sign to avoid using it.

There's another thing you can do to strongly suggest that outside objects don't access a property or method: prefix it with a double underscore, `__`. This will perform **name mangling** on the attribute in question. In essence, name mangling means that the method can still be called by outside objects if they really want to do so, but it requires extra work and is a strong indicator that you demand that your attribute remains **private**.

When we use a double underscore, the property is prefixed with `_<classname>`. When methods in the class internally access the variable, they are automatically unmangled. When external classes wish to access it, they have to do the name mangling themselves. So, name mangling does not guarantee privacy; it only strongly recommends it. This is very rarely used, and often a source of confusion when it is used.

Don't create new double-underscore names in your own code, it will only cause grief and heartache. Consider this reserved for Python's internally defined special names.

What's important is that encapsulation – as a design principle – assures that the methods of a class encapsulate the state changes for the attributes. Whether or not attributes (or methods) are private doesn't change the essential good design that flows from encapsulation.

The encapsulation principle applies to individual classes as well as a module with a bunch of classes. It also applies to a package with a bunch of modules. As designers of object-oriented Python, we're isolating responsibilities and clearly encapsulating features.

And, of course, we're using Python to solve problems. It turns out there's a huge standard library available to help us create useful software. The vast standard library is why we describe Python as a "batteries included" language. Right out of the box, you have almost everything you need, no running to the store to buy batteries.

Outside the standard library, there's an even larger universe of third-party packages. In the next section, we'll look at how we extend our Python installation with even more ready-made goodness.

Third-party libraries

Python ships with a lovely standard library, which is a collection of packages and modules that are available on every machine that runs Python. However, you'll soon find that it doesn't contain everything you need. When this happens, you have two options:

- Write a supporting package yourself
- Use somebody else's code

We won't be covering the details about turning your packages into libraries, but if you have a problem you need to solve and you don't feel like coding it (the best programmers are extremely lazy and prefer to reuse existing, proven code, rather than write their own), you can probably find the library you want on the **Python Package Index** (**PyPI**) at `http://pypi.python.org/`. Once you've identified a package that you want to install, you can use a tool called `pip` to install it.

You can install packages using an operating system command such as the following:

```
% python -m pip install mypy
```

If you try this without making any preparation, you'll either be installing the third-party library directly into your system Python directory, or, more likely, will get an error that you don't have permission to update the system Python.

The common consensus in the Python community is that you don't touch any Python that's part of the OS. Older Mac OS X releases had a Python 2.7 installed. This was not really available for end users. It's best to think of it as part of the OS; and ignore it and always install a fresh, new Python.

Python ships with a tool called `venv`, a utility that gives you a Python installation called a **virtual environment** in your working directory. When you activate this environment, commands related to Python will work with your virtual environment's Python instead of the system Python. So, when you run `pip` or `python`, it won't touch the system Python at all. Here's how to use it:

```
cd project_directory
python -m venv env
source env/bin/activate    # on Linux or macOS
env/Scripts/activate.bat   # on Windows
```

(For other OSes, see `https://docs.python.org/3/library/venv.html`, which has all the variations required to activate the environment.)

Once the virtual environment is activated, you are assured that `python -m pip` will install new packages into the virtual environment, leaving any OS Python alone. You can now use the `python -m pip install mypy` command to add the ***mypy*** tool to your current virtual environment.

On a home computer – where you have access to the privileged files – you can sometimes get away with installing and working with a single, centralized system-wide Python. In an enterprise computing environment, where system-wide directories require special privileges, a virtual environment is required. Because the virtual environment approach always works, and the centralized system-level approach doesn't always work, it's generally a best practice to create and use virtual environments.

It's typical to create a different virtual environment for each Python project. You can store your virtual environments anywhere, but a good practice is to keep them in the same directory as the rest of the project files. When working with version control tools like **Git**, the `.gitignore` file can make sure your virtual environments are not checked into the Git repository.

When starting something new, we often create the directory, and then `cd` into that directory. Then, we'll run the `python -m venv env` utility to create a virtual environment, usually with a simple name like `env`, and sometimes with a more complex name like `CaseStudy39`.

Finally, we can use one of the last two lines in the preceding code (depending on the operating system, as indicated in the comments) to activate the environment.

Each time we do some work on a project, we can `cd` to the directory and execute the `source` (or `activate.bat`) line to use that particular virtual environment. When switching projects, a `deactivate` command unwinds the environment setup.

Virtual environments are essential for keeping your third-party dependencies separate from Python's standard library. It is common to have different projects that depend on different versions of a particular library (for example, an older website might run on Django 1.8, while newer versions run on Django 2.1). Keeping each project in separate virtual environments makes it easy to work in either version of Django. Furthermore, it prevents conflicts between system-installed packages and `pip`-installed packages if you try to install the same package using different tools. Finally, it bypasses any OS permission restrictions surrounding the OS Python.

There are several third-party tools for managing virtual environments effectively. Some of these include `virtualenv`, `pyenv`, `virtualenvwrapper`, and `conda`. If you're working in a data science environment, you'll probably need to use `conda` so you can install more complex packages. There are a number of features leading to a lot of different approaches to solving the problem of managing the huge Python ecosystem of third-party packages.

Case study

This section expands on the object-oriented design of our realistic example. We'll start with the diagrams created using the **Unified Modeling Language** (**UML**) to help depict and summarize the software we're going to build.

We'll describe the various considerations that are part of the Python implementation of the class definitions. We'll start with a review of the diagrams that describe the classes to be defined.

Logical view

Here's the overview of the classes we need to build. This is (except for one new method) the previous chapter's model:

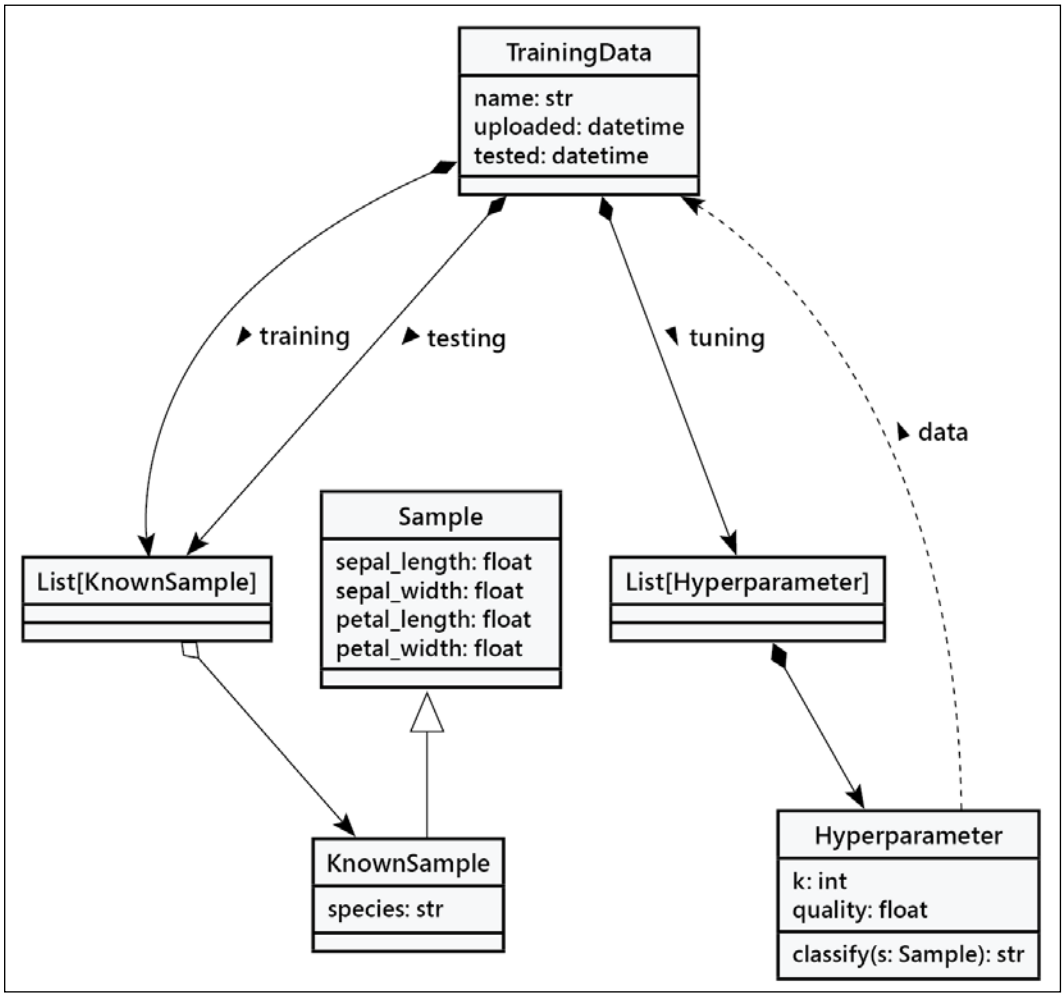

Figure 2.2: Logical view diagram

There are three classes that define our core data model, plus some uses of the generic list class. We've shown it using the type hint of `List`. Here are the four central classes:

- The `TrainingData` class is a container with two lists of data samples, a list used for training our model and a list used for testing our model. Both lists are composed of `KnownSample` instances. Additionally, we'll also have a list of alternative `Hyperparameter` values. In general, these are tuning values that change the behavior of the model. The idea is to test with different hyperparameters to locate the highest-quality model.

We've also allocated a little bit of metadata to this class: the name of the data we're working with, the datetime of when we uploaded the data the first time, and the datetime of when we ran a test against the model.

- Each instance of the `Sample` class is the core piece of working data. In our example, these are measurements of sepal lengths and widths and petal lengths and widths. Steady-handed botany graduate students carefully measured lots and lots of flowers to gather this data. We hope that they had time to stop and smell the roses while they were working.
- A `KnownSample` object is an extended `Sample`. This part of the design foreshadows the focus of *Chapter 3*, *When Objects Are Alike*. A `KnownSample` is a `Sample` with one extra attribute, the assigned species. This information comes from skilled botanists who have classified some data we can use for training and testing.
- The `Hyperparameter` class has the *k* used to define how many of the nearest neighbors to consider. It also has a summary of testing with this value of *k*. The quality tells us how many of the test samples were correctly classified. We expect to see that small values of *k* (like 1 or 3) don't classify well. We expect middle values of *k* to do better, and very large values of *k* to not do as well.

The `KnownSample` class on the diagram may not need to be a separate class definition. As we work through the details, we'll look at some alternative designs for each of these classes.

We'll start with the `Sample` (and `KnownSample`) classes. Python offers three essential paths for defining a new class:

- A `class` definition; we'll focus on this to start.
- A `@dataclass` definition. This provides a number of built-in features. While it's handy, it's not ideal for programmers who are new to Python, because it can obscure some implementation details. We'll set this aside for *Chapter 7*, *Python Data Structures*.
- An extension to the `typing.NamedTuple` class. The most notable feature of this definition will be that the state of the object is immutable; the attribute values cannot be changed. Unchanging attributes can be a useful feature for making sure a bug in the application doesn't mess with the training data. We'll set this aside for *Chapter 7*, also.

Our first design decision is to use Python's `class` statement to write a class definition for `Sample` and its subclass `KnownSample`. This may be replaced in the future (i.e., *Chapter 7*) with alternatives that use data classes as well as `NamedTuple`.

Samples and their states

The diagram in *Figure 2.2* shows the `Sample` class and an extension, the `KnownSample` class. This doesn't seem to be a complete decomposition of the various kinds of samples. When we review the user stories and the process views, there seems to be a gap: specifically, the "make classification request" by a User requires an unknown sample. This has the same flower measurements attributes as a `Sample`, but doesn't have the assigned species attribute of a `KnownSample`. Further, there's no state change that adds an attribute value. The unknown sample will never be formally classified by a Botanist; it will be classified by our algorithm, but it's only an AI, not a Botanist.

We can make a case for two distinct subclasses of `Sample`:

- `UnknownSample`: This class contains the initial four `Sample` attributes. A User provides these objects to get them classified.
- `KnownSample`: This class has the `Sample` attributes plus the classification result, a species name. We use these for training and testing the model.

Generally, we consider class definitions as a way to encapsulate state and behavior. An `UnknownSample` instance provided by a user starts out with no species. Then, after the classifier algorithm computes a species, the `Sample` changes state to have a species assigned by the algorithm.

A question we must always ask about class definitions is this:

Is there any change in behavior that goes with the change in state?

In this case, it doesn't seem like there's anything new or different that can happen. Perhaps this can be implemented as a single class with some optional attributes.

We have another possible state change concern. Currently, there's no class that owns the responsibility of partitioning `Sample` objects into the training or testing subsets. This, too, is a kind of state change.

This leads to a second important question:

What class has responsibility for making this state change?

In this case, it seems like the `TrainingData` class should own the discrimination between testing and training data.

One way to help look closely at our class design is to enumerate all of the various states of individual samples. This technique helps uncover a need for attributes in the classes. It also helps to identify the methods to make state changes to objects of a class.

Sample state transitions

Let's look at the life cycles of `Sample` objects. An object's life cycle starts with object creation, then state changes, and (in some cases) the end of its processing life when there are no more references to it. We have three scenarios:

1. **Initial load**: We'll need a `load()` method to populate a `TrainingData` object from some source of raw data. We'll preview some of the material in *Chapter 9, Strings, Serialization, and File Paths*, by saying that reading a CSV file often produces a sequence of dictionaries. We can imagine a `load()` method using a CSV reader to create `Sample` objects with a species value, making them `KnownSample` objects. The `load()` method splits the `KnownSample` objects into the training and testing lists, which is an important state change for a `TrainingData` object.
2. **Hyperparameter testing**: We'll need a `test()` method in the `Hyperparameter` class. The body of the `test()` method works with the test samples in the associated `TrainingData` object. For each sample, it applies the classifier and counts the matches between Botanist-assigned species and the best guess of our AI algorithm. This points out the need for a `classify()` method for a single sample that's used by the `test()` method for a batch of samples. The `test()` method will update the state of the `Hyperparameter` object by setting the quality score.
3. **User-initiated classification**: A RESTful web application is often decomposed into separate view functions to handle requests. When handling a request to classify an unknown sample, the view function will have a `Hyperparameter` object used for classification; this will be chosen by the Botanist to produce the best results. The user input will be an `UnknownSample` instance. The view function applies the `Hyperparameter.classify()` method to create a response to the user with the species the iris has been classed as. Does the state change that happens when the AI classifies an `UnknownSample` really matter? Here are two views:
    - Each `UnknownSample` can have a `classified` attribute. Setting this is a change in the state of the `Sample`. It's not clear that there's any behavior change associated with this state change.
    - The classification result is not part of the `Sample` at all. It's a local variable in the view function. This state change in the function is used to respond to the user, but has no life within the `Sample` object.

There's a key concept underlying this detailed decomposition of these alternatives:

There's no "right" answer.

Some design decisions are based on non-functional and non-technical considerations. These might include the longevity of the application, future use cases, additional users who might be enticed, current schedules and budgets, pedagogical value, technical risk, the creation of intellectual property, and how cool the demo will look in a conference call.

In *Chapter 1*, *Object-Oriented Design*, we dropped a hint that this application is the precursor to a consumer product recommender. We noted: "The users eventually want to tackle complex consumer products, but recognize that solving a difficult problem is not a good way to learn how to build this kind of application. It's better to start with something of a manageable level of complexity and then refine and expand it until it does everything they need."

Because of that, we'll consider a change in state from `UnknownSample` to `ClassifiedSample` to be very important. The `Sample` objects will live in a database for additional marketing campaigns or possibly reclassification when new products are available and the training data changes.

We'll decide to keep the classification and the species data in the `UnknownSample` class.

This analysis suggests we can coalesce all the various `Sample` details into the following design:

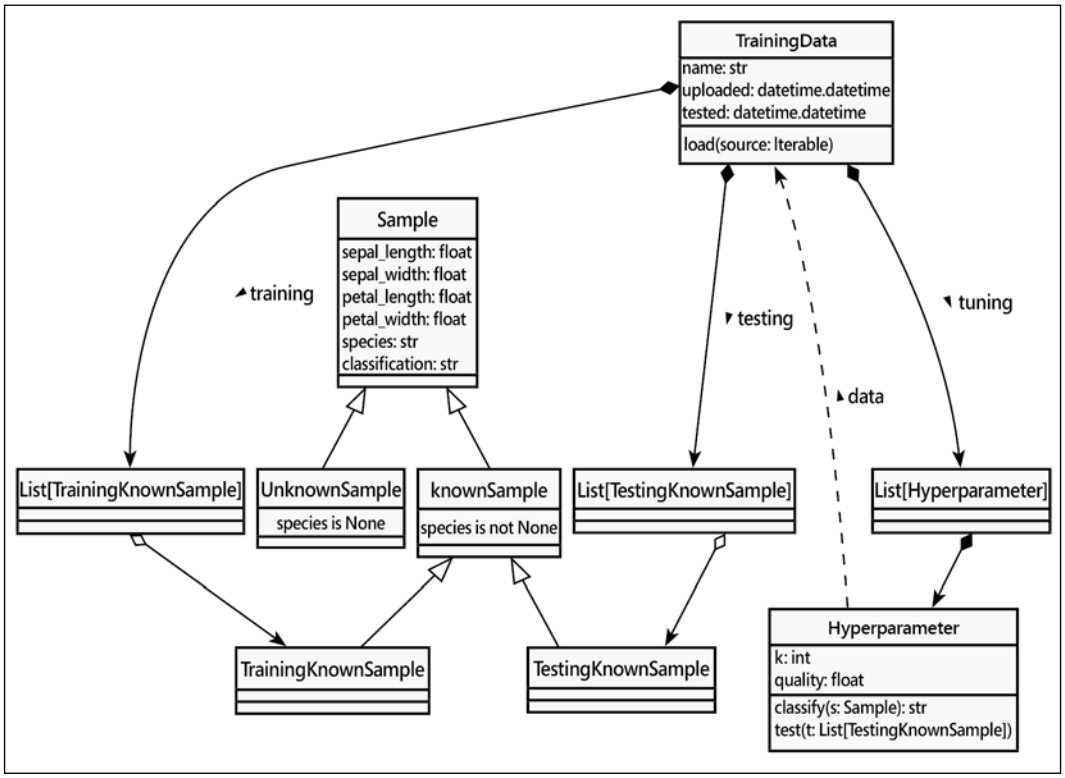

Figure 2.3: The updated UML diagram

This view uses the open arrowhead to show a number of subclasses of `Sample`. We won't directly implement these as subclasses. We've included the arrows to show that we have some distinct use cases for these objects. Specifically, the box for `KnownSample` has a condition **species is not None** to summarize what's unique about these `Sample` objects. Similarly, the `UnknownSample` has a condition, **species is None**, to clarify our intent around `Sample` objects with the species attribute value of `None`.

In these UML diagrams, we have generally avoided showing Python's "special" methods. This helps to minimize visual clutter. In some cases, a special method may be absolutely essential, and worthy of showing in a diagram. An implementation almost always needs to have an `__init__()` method.

There's another special method that can really help: the `__repr__()` method is used to create a representation of the object. This representation is a string that generally has the syntax of a Python expression to rebuild the object. For simple numbers, it's the number. For a simple string, it will include the quotes. For more complex objects, it will have all the necessary Python punctuation, including all the details of the class and state of the object. We'll often use an f-string with the class name and the attribute values.

Here's the start of a class, `Sample`, which seems to capture all the features of a single sample:

```
class Sample:

    def __init__(
        self,
        sepal_length: float,
        sepal_width: float,
        petal_length: float,
        petal_width: float,
        species: Optional[str] = None,
    ) -> None:
        self.sepal_length = sepal_length
        self.sepal_width = sepal_width
        self.petal_length = petal_length
        self.petal_width = petal_width
        self.species = species
        self.classification: Optional[str] = None

    def __repr__(self) -> str:
        if self.species is None:
            known_unknown = "UnknownSample"
        else:
            known_unknown = "KnownSample"
        if self.classification is None:
            classification = ""
        else:
            classification = f", {self.classification}"
```

```
        return (
            f"{known_unknown}("
            f"sepal_length={self.sepal_length}, "
            f"sepal_width={self.sepal_width}, "
            f"petal_length={self.petal_length}, "
            f"petal_width={self.petal_width}, "
            f"species={self.species!r}"
            f"{classification}"
            f")"
        )
```

The `__repr__()` method reflects the fairly complex internal state of this `Sample` object. The states implied by the presence (or absence) of a species and the presence (or absence) of a classification lead to small behavior changes. So far, any changes in object behavior are limited to the `__repr__()` method used to display the current state of the object.

What's important is that the state changes do lead to a (tiny) behavioral change.

We have two application-specific methods for the `Sample` class. These are shown in the next code snippet:

```
    def classify(self, classification: str) -> None:
        self.classification = classification

    def matches(self) -> bool:
        return self.species == self.classification
```

The `classify()` method defines the state change from unclassified to classified. The `matches()` method compares the results of classification with a Botanist-assigned species. This is used for testing.

Here's an example of how these state changes can look:

```
>>> from model import Sample
>>> s2 = Sample(
...     sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_
width=0.2, species="Iris-setosa")
>>> s2
KnownSample(sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_
width=0.2, species='Iris-setosa')
>>> s2.classification = "wrong"
```

```
>>> s2
KnownSample(sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_
width=0.2, species='Iris-setosa', classification='wrong')
```

We have a workable definition of the `Sample` class. The `__repr__()` method is quite complex, suggesting there may be some improvements possible.

It can help to define responsibilities for each class. This can be a focused summary of the attributes and methods with a little bit of additional rationale to tie them together.

Class responsibilities

Which class is responsible for actually performing a test? Does the `Training` class invoke the classifier on each `KnownSample` in a testing set? Or, perhaps, does it provide the testing set to the `Hyperparameter` class, delegating the testing to the `Hyperparameter` class? Since the `Hyperparameter` class has responsibility for the *k* value, and the algorithm for locating the *k*-nearest neighbors, it seems sensible for the `Hyperparameter` class to run the test using its own *k* value and a list of `KnownSample` instances provided to it.

It also seems clear the `TrainingData` class is an acceptable place to record the various `Hyperparameter` trials. This means the `TrainingData` class can identify which of the `Hyperparameter` instances has a value of *k* that classifies irises with the highest accuracy.

There are multiple, related state changes here. In this case, both the `Hyperparameter` and `TrainingData` classes will do part of the work. The system – as a whole – will change state as individual elements change state. This is sometimes described as **emergent behavior**. Rather than writing a monster class that does many things, we've written smaller classes that collaborate to achieve the expected goals.

This `test()` method of `TrainingData` is something that we didn't show in the UML image. We included `test()` in the `Hyperparameter` class, but, at the time, it didn't seem necessary to add it to `TrainingData`.

Here's the start of the class definition:

```
class Hyperparameter:
    """A hyperparameter value and the overall quality of the
classification."""

    def __init__(self, k: int, training: "TrainingData") -> None:
```

```
        self.k = k
        self.data: weakref.ReferenceType["TrainingData"] =
weakref.ref(training)
        self.quality: float
```

Note how we write type hints for classes not yet defined. When a class is defined later in the file, any reference to the yet-to-be-defined class is a *forward reference*. The forward references to the not-yet-defined `TrainingData` class are provided as strings, not the simple class name. When ***mypy*** is analyzing the code, it resolves the strings into proper class names.

The testing is defined by the following method:

```
    def test(self) -> None:
        """Run the entire test suite."""
        training_data: Optional["TrainingData"] = self.data()
        if not training_data:
            raise RuntimeError("Broken Weak Reference")
        pass_count, fail_count = 0, 0
        for sample in training_data.testing:
            sample.classification = self.classify(sample)
            if sample.matches():
                pass_count += 1
            else:
                fail_count += 1
        self.quality = pass_count / (pass_count + fail_count)
```

We start by resolving the weak reference to the training data. This will raise an exception if there's a problem. For each testing sample, we classify the sample, setting the sample's `classification` attribute. The `matches` method tells us if the model's classification matches the known species. Finally, the overall quality is measured by the fraction of tests that passed. We can use the integer count, or a floating-point ratio of tests passed out of the total number of tests.

We won't look at the classification method in this chapter; we'll save that for *Chapter 10, The Iterator Pattern*. Instead, we'll finish this model by looking at the `TrainingData` class, which combines the elements seen so far.

The TrainingData class

The `TrainingData` class has lists with two subclasses of `Sample` objects. The `KnownSample` and `UnknownSample` can be implemented as extensions to a common parent class, `Sample`.

We'll look at this from a number of perspectives in *Chapter 7*. The `TrainingData` class also has a list with `Hyperparameter` instances. This class can have simple, direct references to previously defined classes.

This class has the two methods that initiate the processing:

- The `load()` method reads raw data and partitions it into training data and test data. Both of these are essentially `KnownSample` instances with different purposes. The training subset is for evaluating the *k*-NN algorithm; the testing subset is for determining how well the *k* hyperparameter is working.
- The `test()` method uses a `Hyperparameter` object, performs the test, and saves the result.

Looking back at *Chapter 1*'s context diagram, we see three stories: *Provide Training Data*, *Set Parameters and Test Classifier*, and *Make Classification Request*. It seems helpful to add a method to perform a classification using a given `Hyperparameter` instance. This would add a `classify()` method to the `TrainingData` class. Again, this was not clearly required at the beginning of our design work, but seems like a good idea now.

Here's the start of the class definition:

```
class TrainingData:
    """A set of training data and testing data with methods to load and
test the samples."""

    def __init__(self, name: str) -> None:
        self.name = name
        self.uploaded: datetime.datetime
        self.tested: datetime.datetime
        self.training: List[Sample] = []
        self.testing: List[Sample] = []
        self.tuning: List[Hyperparameter] = []
```

We've defined a number of attributes to track the history of the changes to this class. The uploaded time and the tested time, for example, provide some history. The `training`, `testing`, and `tuning` attributes have `Sample` objects and `Hyperparameter` objects.

We won't write methods to set all of these. This is Python and direct access to attributes is a huge simplification to complex applications. The responsibilities are encapsulated in this class, but we don't generally write a lot of getter/setter methods.

In *Chapter 5*, *When to Use Object-Oriented Programming*, we'll look at some clever techniques, like Python's property definitions, additional ways to handle these attributes.

The `load()` method is designed to process data given by another object. We could have designed the `load()` method to open and read a file, but then we'd bind the `TrainingData` to a specific file format and logical layout. It seems better to isolate the details of the file format from the details of managing training data. In *Chapter 5*, we'll look closely at reading and validating input. In *Chapter 9*, *Strings, Serialization, and File Paths*, we'll revisit the file format considerations.

For now, we'll use the following outline for acquiring the training data:

```
    def load(
            self,
            raw_data_source: Iterable[dict[str, str]]
    ) -> None:
        """Load and partition the raw data"""
        for n, row in enumerate(raw_data_source):
            ... filter and extract subsets (See Chapter 6)
            ... Create self.training and self.testing subsets
        self.uploaded = datetime.datetime.now(tz=datetime.timezone.utc)
```

We'll depend on a source of data. We've described the properties of this source with a type hint, `Iterable[dict[str, str]]`. The `Iterable` states that the method's results can be used by a `for` statement or the `list` function. This is true of collections like lists and files. It's also true of generator functions, the subject of *Chapter 10*, *The Iterator Pattern*.

The results of this iterator need to be dictionaries that map strings to strings. This is a very general structure, and it allows us to require a dictionary that looks like this:

```
{
    "sepal_length": 5.1,
    "sepal_width": 3.5,
    "petal_length": 1.4,
    "petal_width": 0.2,
    "species": "Iris-setosa"
}
```

This required structure seems flexible enough that we can build some object that will produce it. We'll look at the details in *Chapter 9*.

The remaining methods delegate most of their work to the `Hyperparameter` class. Rather than do the work of classification, this class relies on another class to do the work:

```
def test(
        self,
        parameter: Hyperparameter) -> None:
    """Test this Hyperparameter value."""
    parameter.test()
    self.tuning.append(parameter)
    self.tested = datetime.datetime.now(tz=datetime.timezone.utc)

def classify(
        self,
        parameter: Hyperparameter,
        sample: Sample) -> Sample:
    """Classify this Sample."""
    classification = parameter.classify(sample)
    sample.classify(classification)
    return sample
```

In both cases, a specific `Hyperparameter` object is provided as a parameter. For testing, this makes sense because each test should have a distinct value. For classification, however, the "best" `Hyperparameter` object should be used for classification.

This part of the case study has built class definitions for `Sample`, `KnownSample`, `TrainingData`, and `Hyperparameter`. These classes capture parts of the overall application. This isn't complete, of course; we've omitted some important algorithms. It's good to start with things that are clear, identify behavior and state change, and define the responsibilities. The next pass of design can then fill in details around this existing framework.

Recall

Some key points in this chapter:

- Python has optional type hints to help describe how data objects are related and what the parameters should be for methods and functions.
- We create Python classes with the `class` statement. We should initialize the attributes in the special `__init__()` method.

- Modules and packages are used as higher-level groupings of classes.
- We need to plan out the organization of module content. While the general advice is "flat is better than nested," there are a few cases where it can be helpful to have nested packages.
- Python has no notion of "private" data. We often say "we're all adults here"; we can see the source code, and private declarations aren't very helpful. This doesn't change our design; it simply removes the need for a handful of keywords.
- We can install third-party packages using PIP tools. We can create a virtual environment, for example, with `venv`.

Exercises

Write some object-oriented code. The goal is to use the principles and syntax you learned in this chapter to ensure you understand the topics we've covered. If you've been working on a Python project, go back over it and see whether there are some objects you can create and add properties or methods to. If it's large, try dividing it into a few modules or even packages and play with the syntax. While a "simple" script may expand when refactored into classes, there's generally a gain in flexibility and extensibility.

If you don't have such a project, try starting a new one. It doesn't have to be something you intend to finish; just stub out some basic design parts. You don't need to fully implement everything; often, just a `print("this method will do something")` is all you need to get the overall design in place. This is called **top-down design**, in which you work out the different interactions and describe how they should work before actually implementing what they do. The converse, **bottom-up design**, implements details first and then ties them all together. Both patterns are useful at different times, but for understanding object-oriented principles, a top-down workflow is more suitable.

If you're having trouble coming up with ideas, try writing a to-do application. It can keep track of things you want to do each day. Items can have a state change from incomplete to completed. You might want to think about items that have an intermediate state of started, but not yet completed.

Now try designing a bigger project. A collection of classes to model playing cards can be an interesting challenge. Cards have a few features, but there are many variations on the rules. A class for a hand of cards has interesting state changes as cards are added. Locate a game you like and create classes to model cards, hands, and play. (Don't tackle creating a winning strategy; that can be hard.)

A game like Cribbage has an interesting state change where two cards from each player's hand are used to create a kind of third hand, called "the crib." Make sure you experiment with the package and module-importing syntax. Add some functions in various modules and try importing them from other modules and packages. Use relative and absolute imports. See the difference, and try to imagine scenarios where you would want to use each one.

Summary

In this chapter, we learned how simple it is to create classes and assign properties and methods in Python. Unlike many languages, Python differentiates between a constructor and an initializer. It has a relaxed attitude toward access control. There are many different levels of scope, including packages, modules, classes, and functions. We understood the difference between relative and absolute imports, and how to manage third-party packages that don't come with Python.

In the next chapter, we'll learn more about sharing implementation using inheritance.

3
When Objects Are Alike

In the programming world, duplicate code is considered evil. We should not have multiple copies of the same, or similar, code in different places. When we fix a bug in one copy and fail to fix the same bug in another copy, we've caused no end of problems for ourselves.

There are many ways to merge pieces of code or objects that have a similar functionality. In this chapter, we'll be covering the most famous object-oriented principle: inheritance. As discussed in *Chapter 1*, *Object-Oriented Design*, inheritance allows us to create "is-a" relationships between two or more classes, abstracting common logic into superclasses and extending the superclass with specific details in each subclass. In particular, we'll be covering the Python syntax and principles for the following:

- Basic inheritance
- Inheriting from built-in types
- Multiple inheritance
- Polymorphism and duck typing

This chapter's case study will expand on the previous chapter. We'll leverage the concepts of inheritance and abstraction to look for ways to manage common code in parts of the *k*-nearest neighbors computation.

We'll start by taking a close look at how inheritance works to factor out common features so we can avoid copy-and-paste programming.

Basic inheritance

Technically, every class we create uses inheritance. All Python classes are subclasses of the special built-in class named `object`. This class provides a little bit of metadata and a few built-in behaviors so Python can treat all objects consistently.

If we don't explicitly inherit from a different class, our classes will automatically inherit from `object`. However, we can redundantly state that our class derives from `object` using the following syntax:

```
class MySubClass(object):
    pass
```

This is inheritance! This example is, technically, no different from our very first example in *Chapter 2, Objects in Python*. In Python 3, all classes automatically inherit from `object` if we don't explicitly provide a different **superclass**. The superclasses, or *parent* classes, in the relationship are the classes that are being inherited from, `object` in this example. A subclass – `MySubClass`, in this example – inherits from a superclass. A subclass is also said to be *derived from* its parent class, or the subclass *extends* the parent class.

As you've probably figured out from the example, inheritance requires a minimal amount of extra syntax over a basic class definition. Simply include the name of the parent class inside parentheses between the class name and the colon that follows. This is all we have to do to tell Python that the new class should be derived from the given superclass.

How do we apply inheritance in practice? The simplest and most obvious use of inheritance is to add functionality to an existing class. Let's start with a contact manager that tracks the names and email addresses of several people. The `Contact` class is responsible for maintaining a global list of all contacts ever seen in a class variable, and for initializing the name and address for an individual contact:

```
class Contact:
   all_contacts: List["Contact"] = []

   def __init__(self, name: str, email: str) -> None:
       self.name = name
       self.email = email
       Contact.all_contacts.append(self)

   def __repr__(self) -> str:
       return (
           f"{self.__class__.__name__}("
```

```
            f"{self.name!r}, {self.email!r}"
            f")"
        )
```

This example introduces us to **class variables**. The `all_contacts` list, because it is part of the class definition, is shared by all instances of this class. This means that there is only one `Contact.all_contacts` list. We can also access it as `self.all_contacts` from within any method on an instance of the `Contact` class. If a field can't be found on the object (via `self`), then it will be found on the class and will thus refer to the same single list.

Be careful with the `self`-based reference. It can only provide access to an existing class-based variable. If you ever attempt to ***set*** the variable using `self.all_contacts`, you will actually be creating a ***new*** instance variable associated just with that object. The class variable will still be unchanged and accessible as `Contact.all_contacts`.

We can see how the class tracks data with the following example:

```
>>> c_1 = Contact("Dusty", "dusty@example.com")
>>> c_2 = Contact("Steve", "steve@itmaybeahack.com")
>>> Contact.all_contacts
[Contact('Dusty', 'dusty@example.com'), Contact('Steve',
'steve@itmaybeahack.com')]
```

We created two instances of the `Contact` class and assigned them to variables `c_1` and `c_2`. When we looked at the `Contact.all_contacts` class variable, we saw that the list has been updated to track the two objects.

This is a simple class that allows us to track a couple of pieces of data about each contact. But what if some of our contacts are also suppliers that we need to order supplies from? We could add an `order` method to the `Contact` class, but that would allow people to accidentally order things from contacts who are customers or family friends. Instead, let's create a new `Supplier` class that acts like our `Contact` class, but has an additional `order` method that accepts a yet-to-be-defined `Order` object:

```
class Supplier(Contact):
    def order(self, order: "Order") -> None:
        print(
            "If this were a real system we would send "
            f"'{order}' order to '{self.name}'"
        )
```

Now, if we test this class in our trusty interpreter, we see that all contacts, including suppliers, accept a name and email address in their `__init__()` method, but that only `Supplier` instances have an `order()` method:

```
>>> c = Contact("Some Body", "somebody@example.net")
>>> s = Supplier("Sup Plier", "supplier@example.net")
>>> print(c.name, c.email, s.name, s.email)
Some Body somebody@example.net Sup Plier supplier@example.net

>>> from pprint import pprint
>>> pprint(c.all_contacts)
[Contact('Dusty', 'dusty@example.com'),
 Contact('Steve', 'steve@itmaybeahack.com'),
 Contact('Some Body', 'somebody@example.net'),
 Supplier('Sup Plier', 'supplier@example.net')]

>>> c.order("I need pliers")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Contact' object has no attribute 'order'
>>> s.order("I need pliers")
If this were a real system we would send 'I need pliers' order to 'Sup
Plier'
```

Our `Supplier` class can do everything a contact can do (including adding itself to the list of `Contact.all_contacts`) and all the special things it needs to handle as a supplier. This is the beauty of inheritance.

Also, note that `Contact.all_contacts` has collected every instance of the `Contact` class as well as the subclass, `Supplier`. If we used `self.all_contacts`, then this would ***not*** collect all objects into the `Contact` class, but would put `Supplier` instances into `Supplier.all_contacts`.

Extending built-ins

One interesting use of this kind of inheritance is adding functionality to built-in classes. In the `Contact` class seen earlier, we are adding contacts to a list of all contacts. What if we also wanted to search that list by name? Well, we could add a method on the `Contact` class to search it, but it feels like this method actually belongs to the list itself.

The following example shows how we can do this using inheritance from a built-in type. In this case, we're using the `list` type. We're going to inform ***mypy*** that our list is only of instances of the `Contact` class by using `list["Contact"]`. For this syntax to work in Python 3.9, we need to also import the `annotations` module from the `__future__` package. The definitions look like this:

```
from __future__ import annotations
class ContactList(list["Contact"]):
    def search(self, name: str) -> list["Contact"]:

        matching_contacts: list["Contact"] = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()

    def __init__(self, name: str, email: str) -> None:
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

    def __repr__(self) -> str:
        return (
            f"{self.__class__.__name__}("
            f"{self.name!r}, {self.email!r}" f")"
        )
```

Instead of instantiating a generic list as our class variable, we create a new `ContactList` class that extends the built-in `list` data type. Then, we instantiate this subclass as our `all_contacts` list. We can test the new search functionality as follows:

```
>>> c1 = Contact("John A", "johna@example.net")
>>> c2 = Contact("John B", "johnb@sloop.net")
>>> c3 = Contact("Jenna C", "cutty@sark.io")
>>> [c.name for c in Contact.all_contacts.search('John')]
['John A', 'John B']
```

We have two ways to create generic list objects. With type hints, we have another way of talking about lists, separate from creating actual list instances.

First, creating a list with `[]` is actually a shortcut for creating a list using `list()`; the two syntaxes behave identically:

```
>>> [] == list()
True
```

The `[]` is short and sweet. We can call it **syntactic sugar**; it is a call to the `list()` constructor, written with two characters instead of six. The `list` name refers to a data type: it is a class that we can extend.

Tools like ***mypy*** can check the body of the `ContactList.search()` method to be sure it really will create a `list` instance populated with `Contact` objects. Be sure you've installed a version that's 0.812 or newer; older versions of ***mypy*** don't handle these annotations based on generic types completely.

Because we provided the `Contact` class definition after the definition of the `ContactList` class, we had to provide the reference to a not-yet-defined class as a string, `list["Contact"]`. It's often more common to provide the individual item class definition first, and the collection can then refer to the defined class by name without using a string.

As a second example, we can extend the `dict` class, which is a collection of keys and their associated values. We can create instances of dictionaries using the `{}` syntax sugar. Here's an extended dictionary that tracks the longest key it has seen:

```
class LongNameDict(dict[str, int]):
    def longest_key(self) -> Optional[str]:
        """In effect, max(self, key=len), but less obscure"""
        longest = None
        for key in self:
            if longest is None or len(key) > len(longest):
                longest = key
        return longest
```

The hint for the class narrowed the generic `dict` to a more specific `dict[str, int]`; the keys are of type `str` and the values are of type `int`. This helps ***mypy*** reason about the `longest_key()` method. Since the keys are supposed to be `str`-type objects, the statement `for key in self:` will iterate over `str` objects. The result will be a `str`, or possibly `None`. That's why the result is described as `Optional[str]`. (Is `None` appropriate? Perhaps not. Perhaps a `ValueError` exception is a better idea; that will have to wait until *Chapter 4, Expecting the Unexpected*.)

We're going to be working with strings and integer values. Perhaps the strings are usernames, and the integer values are the number of articles they've read on a website. In addition to the core username and reading history, we also need to know the longest name so we can format a table of scores with the right size display box. This is easy to test in the interactive interpreter:

```
>>> articles_read = LongNameDict()
>>> articles_read['lucy'] = 42
>>> articles_read['c_c_phillips'] = 6
>>> articles_read['steve'] = 7
>>> articles_read.longest_key()
'c_c_phillips'
>>> max(articles_read, key=len)
'c_c_phillips'
```

What if we wanted a more generic dictionary? Say with either strings ***or*** integers as the values? We'd need a slightly more expansive type hint. We might use `dict[str, Union[str, int]]` to describe a dictionary mapping strings to a union of either strings or integers.

Most built-in types can be similarly extended. These built-in types fall into several interesting families, with separate kinds of type hints:

- Generic collections: `set`, `list`, `dict`. These use type hints like `set[something]`, `list[something]`, and `dict[key, value]` to narrow the hint from purely generic to something more specific that the application will actually use. To use the generic types as annotations, a `from __future__ import annotations` is required as the first line of code.
- The `typing.NamedTuple` definition lets us define new kinds of immutable tuples and provide useful names for the members. This will be covered in *Chapter 7, Python Data Structures*, and *Chapter 8, The Intersection of Object-Oriented and Functional Programming*.
- Python has type hints for file-related I/O objects. A new kind of file can use a type hint of `typing.TextIO` or `typing.BinaryIO` to describe built-in file operations.
- It's possible to create new types of strings by extending `typing.Text`. For the most part, the built-in `str` class does everything we need.
- New numeric types often start with the `numbers` module as a source for built-in numeric functionality.

We'll use the generic collections heavily throughout the book. As noted, we'll look at named tuples in later chapters. The other extensions to built-in types are too advanced for this book. In the next section, we'll look more deeply at the benefits of inheritance and how we can selectively leverage features of the superclass in our subclass.

Overriding and super

So, inheritance is great for *adding* new behavior to existing classes, but what about *changing* behavior? Our `Contact` class allows only a name and an email address. This may be sufficient for most contacts, but what if we want to add a phone number for our close friends?

As we saw in *Chapter 2*, *Objects in Python*, we can do this easily by setting a `phone` attribute on the contact after it is constructed. But if we want to make this third variable available on initialization, we have to override the `__init__()` method. Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass. No special syntax is needed to do this; the subclass's newly created method is automatically called instead of the superclass's method, as shown in the following code:

```
class Friend(Contact):
    def __init__(self, name: str, email: str, phone: str) -> None:
        self.name = name
        self.email = email
        self.phone = phone
```

Any method can be overridden, not just `__init__()`. Before we go on, however, we need to address some problems in this example. Our `Contact` and `Friend` classes have duplicate code to set up the `name` and `email` properties; this can make code maintenance complicated, as we have to update the code in two or more places. More alarmingly, our `Friend` class is neglecting to add itself to the `all_contacts` list we have created on the `Contact` class. Finally, looking forward, if we add a feature to the `Contact` class, we'd like it to also be part of the `Friend` class.

What we really need is a way to execute the original `__init__()` method on the `Contact` class from inside our new class. This is what the `super()` function does; it returns the object as if it was actually an instance of the parent class, allowing us to call the parent method directly:

```
class Friend(Contact):
    def __init__(self, name: str, email: str, phone: str) -> None:
        super().__init__(name, email)
        self.phone = phone
```

This example first binds the instance to the parent class using `super()` and calls `__init__()` on that object, passing in the expected arguments. It then does its own initialization, namely, setting the `phone` attribute, which is unique to the `Friend` class.

The `Contact` class provided a definition for the `__repr__()` method to produce a string representation. Our class did not override the `__repr__()` method inherited from the superclass. Here's the consequence of that:

```
>>> f = Friend("Dusty", "Dusty@private.com", "555-1212")
>>> Contact.all_contacts
[Friend('Dusty', 'Dusty@private.com')]
```

The details shown for a `Friend` instance don't include the new attribute. It's easy to overlook the special method definitions when thinking about class design.

A `super()` call can be made inside any method. Therefore, all methods can be modified via overriding and calls to `super()`. The call to `super()` can also be made at any point in the method; we don't have to make the call as the first line. For example, we may need to manipulate or validate incoming parameters before forwarding them to the superclass.

Multiple inheritance

Multiple inheritance is a touchy subject. In principle, it's simple: a subclass that inherits from more than one parent class can access functionality from both of them. In practice, it requires some care to be sure any method overrides are fully understood.

As a humorous rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you might be right.

The simplest and most useful form of multiple inheritance follows a design pattern called the **mixin**. A mixin class definition is not intended to exist on its own, but is meant to be inherited by some other class to provide extra functionality. For example, let's say we wanted to add functionality to our `Contact` class that allows sending an email to `self.email`.

Sending email is a common task that we might want to use on many other classes. So, we can write a simple mixin class to do the emailing for us:

```
class Emailable(Protocol):
    email: str

class MailSender(Emailable):
    def send_mail(self, message: str) -> None:
        print(f"Sending mail to {self.email=}")
        # Add e-mail logic here
```

The `MailSender` class doesn't do anything special (in fact, it can barely function as a standalone class, since it assumes an attribute it doesn't set). We have two classes because we're describing two things: aspects of the host class for a mixin, and new aspects the mixin provides to the host. We needed to create a hint, `Emailable`, to describe the kinds of classes our `MailSender` mixin expects to work with.

This kind of type hint is called a **protocol**; protocols generally have methods, and can also have class-level attribute names with type hints, but not full assignment statements. A protocol definition is a kind of incomplete class; think of it like a contract for features of a class. A protocol tells ***mypy*** that any class (or subclass) of `Emailable` objects must support an `email` attribute, and it must be a string.

Note that we're relying on Python's name resolution rules. The name `self.email` can be resolved as either an instance variable, or a class-level variable, `Emailable.email`, or a property. The ***mypy*** tool will check all the classes mixed in with `MailSender` for instance- or class-level definitions. We only need to provide the name of the attribute at the class level, with a type hint to make it clear to ***mypy*** that the mixin does not define the attribute – the class into which it's mixed will provide the `email` attribute.

Because of Python's duck typing rules, we can use the `MailSender` mixin with any class that has an `email` attribute defined. A class with which `MailSender` is mixed doesn't have to be a formal subclass of `Emailable`; it only has to provide the required attribute.

For brevity, we didn't include the actual email logic here; if you're interested in studying how it's done, see the `smtplib` module in the Python standard library.

The `MailSender` class does allow us to define a new class that describes both a `Contact` and a `MailSender`, using multiple inheritance:

```
class EmailableContact(Contact, MailSender):
    pass
```

The syntax for multiple inheritance looks like a parameter list in the class definition. Instead of including one base class inside the parentheses, we include two (or more), separated by a comma. When it's done well, it's common for the resulting class to have no unique features of its own. It's a combination of mixins, and the body of the class definition is often nothing more than the `pass` placeholder.

We can test this new hybrid to see the mixin at work:

```
>>> e = EmailableContact("John B", "johnb@sloop.net")
>>> Contact.all_contacts
[EmailableContact('John B', 'johnb@sloop.net')]
>>> e.send_mail("Hello, test e-mail here")
Sending mail to self.email='johnb@sloop.net'
```

The `Contact` initializer is still adding the new contact to the `all_contacts` list, and the mixin is able to send mail to `self.email`, so we know that everything is working.

This wasn't so hard, and you're probably wondering what our dire warnings about multiple inheritance were for. We'll get into the complexities in a minute, but let's consider some other options we had for this example, rather than using a mixin:

- We could have used single inheritance and added the `send_mail` function to a subclass of `Contact`. The disadvantage here is that the email functionality then has to be duplicated for any unrelated classes that need an email. For example, if we had email information in the payments part of our application, unrelated to these contacts, and we wanted a `send_mail()` method, we'd have to duplicate the code.
- We can create a standalone Python function for sending an email, and just call that function with the correct email address supplied as a parameter when the email needs to be sent (this is a very common choice). Because the function is not part of a class, it's harder to be sure that proper encapsulation is being used.
- We could explore a few ways of using composition instead of inheritance. For example, `EmailableContact` could have a `MailSender` object as a property instead of inheriting from it. This leads to a more complex `MailSender` class because it now has to stand alone. It also leads to a more complex `EmailableContact` class because it has to associate a `MailSender` instance with each `Contact`.

- We could try to monkey patch (we'll briefly cover monkey patching in *Chapter 13*, *Testing Object-Oriented Programs*) the `Contact` class to have a `send_mail` method after the class has been created. This is done by defining a function that accepts the `self` argument, and setting it as an attribute on an existing class. This is fine for creating a unit test fixture, but terrible for the application itself.

Multiple inheritance works alright when we're mixing methods from different classes, but it can be messy when we have to call methods on the superclass. When there are multiple superclasses, how do we know which one's methods to call? What is the rule for selecting the appropriate superclass method?

Let's explore these questions by adding a home address to our `Friend` class. There are a few approaches we might take:

- An address is a collection of strings representing the street, city, country, and other related details of the contact. We could pass each of these strings as a parameter into the `Friend` class's `__init__()` method. We could also store these strings in a generic tuple or dictionary. These options work well when the address information doesn't need new methods.
- Another option would be to create our own `Address` class to hold those strings together, and then pass an instance of this class into the `__init__()` method in our `Friend` class. The advantage of this solution is that we can add behavior (say, a method to give directions or to print a map) to the data instead of just storing it statically. This is an example of composition, as we discussed in *Chapter 1*, *Object-Oriented Design*. The "has-a" relationship of composition is a perfectly viable solution to this problem and allows us to reuse `Address` classes in other entities, such as buildings, businesses, or organizations. (This is an opportunity to use a dataclass. We'll discuss dataclasses in *Chapter 7*, *Python Data Structures*.)
- A third course of action is a cooperative multiple inheritance design. While this can be made to work, it doesn't pass muster with ***mypy***. The reason, we'll see, is some potential ambiguity that's difficult to describe with the available type hints.

The objective here is to add a new class to hold an address. We'll call this new class `AddressHolder` instead of `Address` because inheritance defines an "is-a" relationship. It is not correct to say a `Friend` class is an `Address` class, but since a friend can have an `Address` class, we can argue that a `Friend` class is an `AddressHolder` class. Later, we could create other entities (companies, buildings) that also hold addresses. (Convoluted naming and nuanced questions about "is-a" serve as decent indications we should be sticking with composition, rather than inheritance.)

Here's a naïve `AddressHolder` class. We're calling it naïve because it doesn't account for multiple inheritance well:

```
class AddressHolder:
    def __init__(self, street: str, city: str, state: str, code: str)
-> None:
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

We take all the data and toss the argument values into instance variables upon initialization. We'll look at the consequences of this, and then show a better design.

The diamond problem

We can use multiple inheritance to add this new class as a parent of our existing `Friend` class. The tricky part is that we now have two parent `__init__()` methods, both of which need to be called. And they need to be called with different arguments. How do we do this? Well, we could start with a naïve approach for the `Friend` class, also:

```
class Friend(Contact, AddressHolder):
    def __init__(
        self,
        name: str,
        email: str,
        phone: str,
        street: str,
        city: str,
        state: str,
        code: str,
    ) -> None:
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

In this example, we directly call the `__init__()` function on each of the superclasses and explicitly pass the `self` argument. This example technically works; we can access the different variables directly on the class. But there are a few problems.

First, it is possible for a superclass to remain uninitialized if we neglect to explicitly call the initializer. That wouldn't break this example, but it could cause hard-to-debug program crashes in common scenarios. We would get a lot of strange-looking `AttributeError` exceptions in classes where there's clearly an `__init__()` method. It's rarely obvious the `__init__()` method wasn't actually used.

A more insidious possibility is a superclass being called multiple times because of the organization of the class hierarchy. Look at this inheritance diagram:

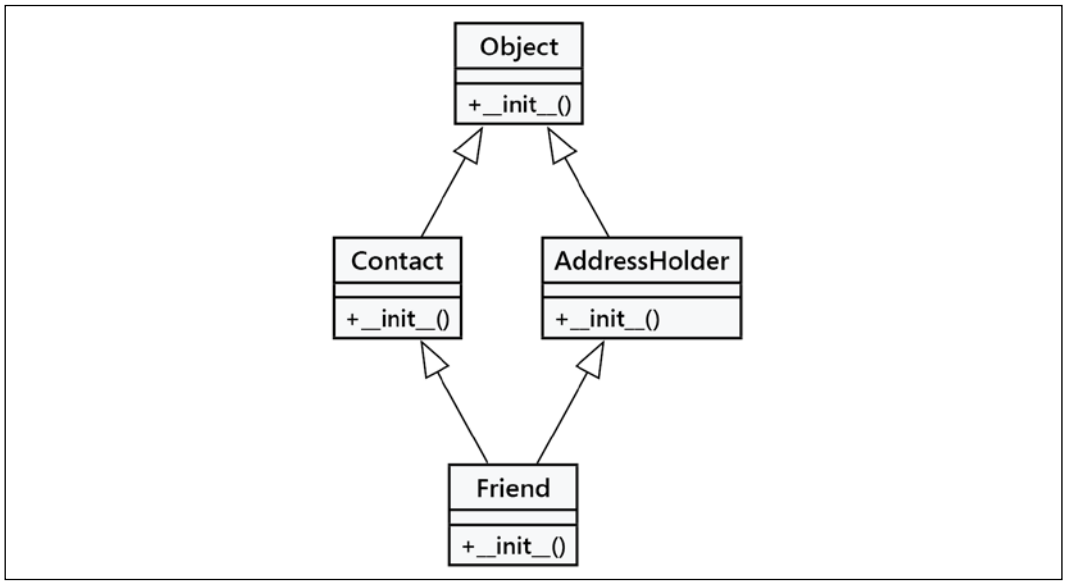

Figure 3.1: Inheritance diagram for our multiple inheritance implementation

The `__init__()` method from the `Friend` class first calls `__init__()` on the `Contact` class, which implicitly initializes the `object` superclass (remember, all classes derive from `object`). The `Friend` class then calls `__init__()` on `AddressHolder`, which implicitly initializes the `object` superclass *again*. This means the parent class has been set up twice. With the `object` class, that's relatively harmless, but in some situations, it could spell disaster. Imagine trying to connect to a database twice for every request!

The base class should only be called once. Once, yes, but when? Do we call `Friend`, then `Contact`, then `Object`, and then `AddressHolder`? Or `Friend`, then `Contact`, then `AddressHolder`, and then `Object`?

Let's contrive an example to illustrate this problem more clearly. Here, we have a base class, `BaseClass`, that has a method named `call_me()`. Two subclasses, `LeftSubclass` and `RightSubclass`, extend the `BaseClass` class, and each overrides the `call_me()` method with different implementations.

Then, *another* subclass extends both of these using multiple inheritance with a fourth, distinct implementation of the `call_me()` method. This is called **diamond inheritance** because of the diamond shape of the class diagram:

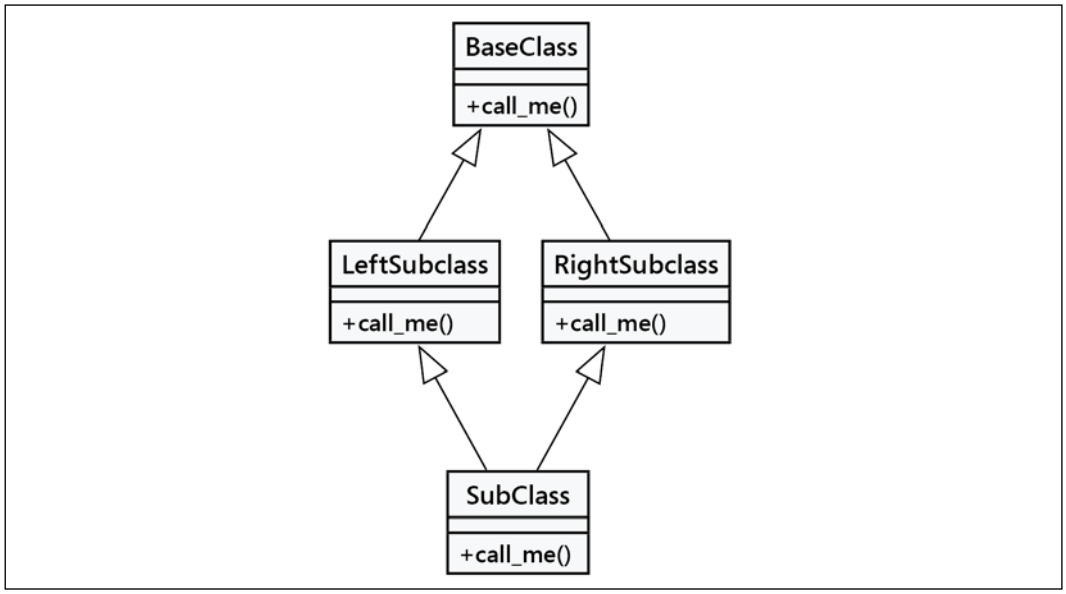

Figure 3.2: Diamond inheritance

Let's convert this diagram into code. This example shows when the methods are called:

```
class BaseClass:
    num_base_calls = 0

    def call_me(self) -> None:
        print("Calling method on BaseClass")
        self.num_base_calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0

    def call_me(self) -> None:
        BaseClass.call_me(self)
        print("Calling method on LeftSubclass")
        self.num_left_calls += 1

class RightSubclass(BaseClass):
    num_right_calls = 0
```

```
    def call_me(self) -> None:
        BaseClass.call_me(self)
        print("Calling method on RightSubclass")
        self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

    def call_me(self) -> None:
        LeftSubclass.call_me(self)
        RightSubclass.call_me(self)
        print("Calling method on Subclass")
        self.num_sub_calls += 1
```

This example ensures that each overridden `call_me()` method directly calls the parent method with the same name. It lets us know each time a method is called by printing the information to the screen. It also creates a distinct instance variable to show how many times it has been called.

The `self.num_base_calls += 1` line requires a little sidebar explanation.

This is effectively `self.num_base_calls = self.num_base_calls + 1`. When Python resolves `self.num_base_calls` on the right side of the `=`, it will first look for an instance variable, then look for the class variable; we've provided a class variable with a default value of zero. After the `+1` computation, the assignment statement will create a new instance variable; it will not update the class-level variable.

Each time after the first call, the instance variable will be found. It's pretty cool for the class to provide default values for instance variables.

If we instantiate one `Subclass` object and call the `call_me()` method on it once, we get the following output:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on BaseClass
Calling method on LeftSubclass
Calling method on BaseClass
```

```
Calling method on RightSubclass
Calling method on Subclass
>>> print(
... s.num_sub_calls,
... s.num_left_calls,
... s.num_right_calls,
... s.num_base_calls)
1 1 1 2
```

Thus, we can see the base class's `call_me()` method being called twice. This could lead to some pernicious bugs if that method is doing actual work, such as depositing into a bank account, twice.

Python's **Method Resolution Order** (**MRO**) algorithm transforms the diamond into a flat, linear tuple. We can see the results of this in the `__mro__` attribute of a class. The linear version of this diamond is the sequence `Subclass`, `LeftSubclass`, `RightSubClass`, `BaseClass`, `object`. What's important here is that `Subclass` lists `LeftSubclass` before `RightSubClass`, imposing an ordering on the classes in the diamond.

The thing to keep in mind with multiple inheritance is that we often want to call the next method in the MRO sequence, not necessarily a method of the parent class. The `super()` function locates the name in the MRO sequence. Indeed, `super()` was originally developed to make complicated forms of multiple inheritance possible.

Here is the same code written using `super()`. We've renamed some of the classes, adding an `_S` to make it clear this is the version using `super()`:

```
class BaseClass:
    num_base_calls = 0

    def call_me(self):
        print("Calling method on Base Class")
        self.num_base_calls += 1

class LeftSubclass_S(BaseClass):
    num_left_calls = 0

    def call_me(self) -> None:
        super().call_me()
        print("Calling method on LeftSubclass_S")
        self.num_left_calls += 1

class RightSubclass_S(BaseClass):
```

```
    num_right_calls = 0

    def call_me(self) -> None:
        super().call_me()
        print("Calling method on RightSubclass_S")
        self.num_right_calls += 1

class Subclass_S(LeftSubclass_S, RightSubclass_S):
    num_sub_calls = 0

    def call_me(self) -> None:
        super().call_me()
        print("Calling method on Subclass_S")
        self.num_sub_calls += 1
```

The change is pretty minor; we only replaced the naive direct calls with calls to `super()`. The `Subclass_S` class, at the bottom of the diamond, only calls `super()` once rather than having to make the calls for both the left and right. The change is easy enough, but look at the difference when we execute it:

```
>>> ss = Subclass_S()
>>> ss.call_me()
Calling method on BaseClass
Calling method on RightSubclass_S
Calling method on LeftSubclass_S
Calling method on Subclass_S
>>> print(
... ss.num_sub_calls,
... ss.num_left_calls,
... ss.num_right_calls,
... ss.num_base_calls)
1 1 1 1
```

This output looks good: our base method is only being called once. We can see how this works by looking at the `__mro__` attribute of the class:

```
>>> from pprint import pprint
>>> pprint(Subclass_S.__mro__)
(<class 'commerce_naive.Subclass_S'>,
 <class 'commerce_naive.LeftSubclass_S'>,
 <class 'commerce_naive.RightSubclass_S'>,
 <class 'commerce_naive.BaseClass'>,
 <class 'object'>)
```

The order of the classes shows what order `super()` will use. The last class in the tuple is generally the built-in `object` class. As noted earlier in this chapter, it's the implicit superclass of all classes.

This shows what `super()` is actually doing. Since the `print` statements are executed after the `super` calls, the printed output is in the order each method is actually executed. Let's look at the output from back to front to see who is calling what:

1. We start with the `Subclass_S.call_me()` method. This evaluates `super().call_me()`. The MRO shows `LeftSubclass_S` as next.
2. We begin evaluation of the `LeftSubclass_S.call_me()` method. This evaluates `super().call_me()`. The MRO puts `RightSubclass_S` as next. This is not a superclass; it's adjacent in the class diamond.
3. The evaluation of the `RightSubclass_S.call_me()` method, `super().call_me()`. This leads to `BaseClass`.
4. The `BaseClass.call_me()` method finishes its processing: printing a message and setting an instance variable, `self.num_base_calls`, to `BaseClass.num_base_calls + 1`.
5. Then, the `RightSubclass_S.call_me()` method can finish, printing a message and setting an instance variable, `self.num_right_calls`.
6. Then, the `LeftSubclass_S.call_me()` method will finish by printing a message and setting an instance variable, `self.num_left_calls`.
7. This serves to set the stage for `Subclass_S` to finish its `call_me()` method processing. It writes a message, sets an instance variable, and rests, happy and successful.

Pay particular attention to this: The `super` call is *not* calling the method on the superclass of `LeftSubclass_S` (which is `BaseClass`). Rather, it is calling `RightSubclass_S`, even though it is not a direct parent of `LeftSubclass_S`! This is the *next* class in the MRO, not the parent method. `RightSubclass_S` then calls `BaseClass` and the `super()` calls have ensured each method in the class hierarchy is executed once.

Different sets of arguments

This is going to make things complicated as we return to our `Friend` cooperative multiple inheritance example. In the `__init__()` method for the `Friend` class, we were originally delegating initialization to the `__init__()` methods of both parent classes, *with different sets of arguments*:

```
Contact.__init__(self, name, email)
AddressHolder.__init__(self, street, city, state, code)
```

How can we manage different sets of arguments when using `super()`? We only really have access to the next class in the MRO sequence. Because of this, we need a way to pass the ***extra*** arguments through the constructors so that subsequent calls to super(), from other mixin classes, receive the right arguments.

It works like this. The first call to `super()` provides arguments to the first class of the MRO, passing the `name` and `email` arguments to `Contact.__init__()`. Then, when `Contact.__init__()` calls `super()`, it needs to be able to pass the address-related arguments to the method of the next class in the MRO, which is `AddressHolder.__init__()`.

This problem often manifests itself anytime we want to call superclass methods with the same name, but with different sets of arguments. Collisions often arise around the special method names. Of these, the most common example is having a different set of arguments to various `__init__()` methods, as we're doing here.

There's no magical Python feature to handle cooperation among classes with divergent `__init__()` parameters. Consequently, this requires some care to design our class parameter lists. The cooperative multiple inheritance approach is to accept keyword arguments for any parameters that are not required by every subclass implementation. A method must pass the unexpected arguments on to its `super()` call, in case they are necessary to later methods in the MRO sequence of classes.

While this works and works well, it's difficult to describe with type hints. Instead, we have to silence ***mypy*** in a few key places.

Python's function parameter syntax provides a tool we can use to do this, but it makes the overall code look cumbersome. Have a look at a version of the `Friend` multiple inheritance code:

```
class Contact:
    all_contacts = ContactList()

    def __init__(self, /, name: str = "", email: str = "", **kwargs:
Any) -> None:
        super().__init__(**kwargs)  # type: ignore [call-arg]
        self.name = name
        self.email = email
        self.all_contacts.append(self)

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(" f"{self.name!r},
{self.email!r}" f")"
```

```
class AddressHolder:
    def __init__(
        self,
        /,
        street: str = "",
        city: str = "",
        state: str = "",
        code: str = "",
        **kwargs: Any,
    ) -> None:
        super().__init__(**kwargs)  # type: ignore [call-arg]
        self.street = street
        self.city = city
        self.state = state
        self.code = code

class Friend(Contact, AddressHolder):
    def __init__(self, /, phone: str = "", **kwargs: Any) -> None:
        super().__init__(**kwargs)
        self.phone = phone
```

We've added the `**kwargs` parameter, which collects all additional keyword argument values into a dictionary. When called with `Contact(name="this", email="that", street="something")`, the `street` argument is put into the `kwargs` dictionary; these extra parameters are passed up to the next class with the `super()` call. The special parameter `/` separates parameters that could be provided by position in the call from parameters that require a keyword to associate them with an argument value. We've given all string parameters an empty string as a default value, also.

If you aren't familiar with the `**kwargs` syntax, it basically collects any keyword arguments passed into the method that were not explicitly listed in the parameter list. These arguments are stored in a dictionary named `kwargs` (we can call the variable whatever we like, but convention suggests `kw` or `kwargs`). When we call a method, for example, `super().__init__()`, with `**kwargs` as an argument value, it unpacks the dictionary and passes the results to the method as keyword arguments. We'll look at this in more depth in *Chapter 8, The Intersection of Object-Oriented and Functional Programming*.

We've introduced two comments that are addressed to ***mypy*** (and any person scrutinizing the code). The `# type: ignore` comments provide a specific error code, `call-arg`, on a specific line to be ignored. In this case, we need to ignore the `super().__init__(**kwargs)` calls because it isn't obvious to ***mypy*** what the MRO really will be at runtime. As someone reading the code, we can look at the `Friend` class and see the order: `Contact` and `AddressHolder`. This order means that inside the `Contact` class, the `super()` function will locate the next class, `AddressHolder`.

The ***mypy*** tool, however, doesn't look this deeply; it goes by the explicit list of parent classes in the `class` statement. Since there's no parent class named, ***mypy*** is convinced the `object` class will be located by `super()`. Since `object.__init__()` cannot take any arguments, the `super().__init__(**kwargs)` in both `Contact` and `AddressHolder` appears incorrect to ***mypy***. Practically, the chain of classes in the MRO will consume all of the various parameters and there will be nothing left over for the `AddressHolder` class's `__init__()` method.

For more information on type hint annotations for cooperative multiple inheritance, see `https://github.com/python/mypy/issues/8769`. The longevity of this issue suggests how hard the solution can be.

The previous example does what it is supposed to do. But it's supremely difficult to answer the question: *What arguments do we need to pass into* `Friend.__init__()`? This is the foremost question for anyone planning to use the class, so a docstring should be added to the method to explain the entire list of parameters from all the parent classes.

The error message in the event of a misspelled or extraneous parameter can be confusing, also. The message `TypeError: object.__init__() takes exactly one argument (the instance to initialize)` isn't too informative on how an extra parameter came to be provided to `object.__init__()`.

We have covered many of the caveats involved with cooperative multiple inheritance in Python. When we need to account for all possible situations, we have to plan for them, and our code can get messy.

Multiple inheritance following the mixin pattern often works out very nicely. The idea is to have additional methods defined in mixin classes, but to keep all of the attributes centralized in a host class hierarchy. This can avoid the complexity of cooperative initialization.

Design using composition also often works better than complex multiple inheritance. Many of the design patterns we'll be covering in *Chapter 11*, *Common Design Patterns,* and *Chapter 12*, *Advanced Design Patterns*, are examples of composition-based design.

The inheritance paradigm depends on a clear "is-a" relationship between classes. Multiple inheritance folds in other relationships that aren't as clear. We can say that an "Email is a kind of Contact," for example. But it doesn't seem as clear that we can say "A Customer is an Email." We might say "A Customer has an Email address" or "A Customer is contacted via Email," using "has an" or "is contacted by" instead of a direct "is-a" relationship.

Polymorphism

We were introduced to polymorphism in *Chapter 1*, *Object-Oriented Design*. It is a showy name describing a simple concept: different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is. It is also sometimes called the Liskov Substitution Principle, honoring Barbara Liskov's contributions to object-oriented programming. We should be able to substitute any subclass for its superclass.

As an example, imagine a program that plays audio files. A media player might need to load an `AudioFile` object and then play it. We can put a `play()` method on the object, which is responsible for decompressing or extracting the audio and routing it to the sound card and speakers. The act of playing an `AudioFile` could feasibly be as simple as:

```
audio_file.play()
```

However, the process of decompressing and extracting an audio file is very different for different types of files. While `.wav` files are stored uncompressed, `.mp3`, `.wma`, and `.ogg` files all utilize totally different compression algorithms.

We can use inheritance with polymorphism to simplify the design. Each type of file can be represented by a different subclass of `AudioFile`, for example, `WavFile` and `MP3File`. Each of these would have a `play()` method that would be implemented differently for each file to ensure that the correct extraction procedure is followed. The media player object would never need to know which subclass of `AudioFile` it is referring to; it just calls `play()` and polymorphically lets the object take care of the actual details of playing. Let's look at a quick skeleton showing how this might work:

```
from pathlib import Path

class AudioFile:
    ext: str
```

```
    def __init__(self, filepath: Path) -> None:
        if not filepath.suffix == self.ext:
            raise ValueError("Invalid file format")
        self.filepath = filepath

class MP3File(AudioFile):
    ext = ".mp3"

    def play(self) -> None:
        print(f"playing {self.filepath} as mp3")

class WavFile(AudioFile):
    ext = ".wav"

    def play(self) -> None:
        print(f"playing {self.filepath} as wav")

class OggFile(AudioFile):
    ext = ".ogg"

    def play(self) -> None:
        print(f"playing {self.filepath} as ogg")
```

All audio files check to ensure that a valid extension was given upon initialization. If the filename doesn't end with the correct name, it raises an exception (exceptions will be covered in detail in *Chapter 4*, *Expecting the Unexpected*).

But did you notice how the `__init__()` method in the parent class is able to access the `ext` class variable from different subclasses? That's polymorphism at work. The `AudioFile` parent class merely has a type hint explaining to ***mypy*** that there will be an attribute named `ext`. It doesn't actually store a reference to the `ext` attribute. When the inherited method is used by a subclass, then the subclass' definition of the `ext` attribute is used. The type hint can help ***mypy*** spot a class missing the attribute assignment.

In addition, each subclass of `AudioFile` implements `play()` in a different way (this example doesn't actually play the music; audio compression algorithms really deserve a separate book!). This is also polymorphism in action. The media player can use the exact same code to play a file, no matter what type it is; it doesn't care what subclass of `AudioFile` it is looking at. The details of decompressing the audio file are *encapsulated*. If we test this example, it works as we would hope:

```
>>> p_1 = MP3File(Path("Heart of the Sunrise.mp3"))
>>> p_1.play()
playing Heart of the Sunrise.mp3 as mp3
>>> p_2 = WavFile(Path("Roundabout.wav"))
>>> p_2.play()
playing Roundabout.wav as wav
>>> p_3 = OggFile(Path("Heart of the Sunrise.ogg"))
>>> p_3.play()
playing Heart of the Sunrise.ogg as ogg
>>> p_4 = MP3File(Path("The Fish.mov"))
Traceback (most recent call last):
...
ValueError: Invalid file format
```

See how `AudioFile.__init__()` can check the file type without actually knowing which subclass it is referring to?

Polymorphism is actually one of the coolest things about object-oriented programming, and it makes some programming designs obvious that weren't possible in earlier paradigms. However, Python makes polymorphism seem less awesome because of duck typing. Duck typing in Python allows us to use *any* object that provides the required behavior without forcing it to be a subclass. The dynamic nature of Python makes this trivial. The following example does not extend `AudioFile`, but it can be interacted with in Python using the exact same interface:

```
class FlacFile:
    def __init__(self, filepath: Path) -> None:
        if not filepath.suffix == ".flac":
            raise ValueError("Not a .flac file")
        self.filepath = filepath

    def play(self) -> None:
        print(f"playing {self.filepath} as flac")
```

Our media player can play objects of the `FlacFile` class just as easily as objects of classes that extend `AudioFile`.

Polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts. Because any objects that supply the correct interface can be used interchangeably in Python, it reduces the need for polymorphic common superclasses. Inheritance can still be useful for sharing code, but if all that is being shared is the public interface, duck typing is all that is required.

This reduced need for inheritance also reduces the need for multiple inheritance; often, when multiple inheritance appears to be a valid solution, we can just use duck typing to mimic one of the multiple superclasses.

In some cases, we can formalize this kind of duck typing using a `typing.Protocol` hint. To make ***mypy*** aware of the expectations, we'll often define a number of functions or attributes (or a mixture) as a formal `Protocol` type. This can help clarify how classes are related. We might, for example, have this kind of definition to define the common features between the `FlacFile` class and the `AudioFile` class hierarchy:

```
class Playable(Protocol):
    def play(self) -> None:
        ...
```

Of course, just because an object satisfies a particular protocol (by providing required methods or attributes) does not mean it will simply work in all situations. It has to fulfill that interface in a way that makes sense in the overall system. Just because an object provides a `play()` method does not mean it will automatically work with a media player. The methods must also have the same meaning, or semantics, in addition to having the same syntax.

Another useful feature of duck typing is that the duck-typed object only needs to provide those methods and attributes that are actually being accessed. For example, if we needed to create a fake file object to read data from, we can create a new object that has a `read()` method; we don't have to override the `write()` method if the code that is going to interact with the fake object will not be calling it. More succinctly, duck typing doesn't need to provide the entire interface of an object that is available; it only needs to fulfill the protocol that is actually used.

Case study

This section expands on the object-oriented design of our example, iris classification. We've been building on this in the previous chapters, and we'll continue building on it in later chapters. In this chapter, we'll review the diagrams created using the **Unified Modeling Language** (**UML**) to help depict and summarize the software we're going to build. We'll move on from the previous chapter to add features for the various ways of computing "nearest" for the *k*-nearest neighbors algorithm. There are a number of variations for this, and it demonstrates how class hierarchies work.

There are several design principles that we'll be exploring as this design becomes more and more complete. One popular set of principles is the **SOLID** principles, which are:

- **S**. Single Responsibility Principle. A class should have one responsibility. This can mean one reason to change when the application's requirements change.
- **O**. Open/Closed. A class should be open to extension but closed to modification.
- **L**. Liskov Substitution. (Named after Barbara Liskov, who created one of the first object-oriented programming languages, CLU.) Any subclass can be substituted for its superclass. This tends to focus a class hierarchy on classes that have very similar interfaces, leading to *polymorphism* among the objects. This the essence of inheritance.
- **I**. Interface Segregation. A class should have the smallest interface possible. This is, perhaps, the most important of these principles. Classes should be relatively small and isolated.
- **D**. Dependency Inversion. This has a peculiar name. We need to know what a bad dependency relationship is so we know how to invert it to have a good relationship. Pragmatically, we'd like classes to be independent, so a Liskov Substitution doesn't involve a lot of code changes. In Python, this often means referring to superclasses in type hints to be sure we have the flexibility to make changes. In some cases, it also means providing parameters so that we can make global class changes without revising any of the code.

We won't look at all of these principles in this chapter. Because we're looking at inheritance, our design will tend to follow the Liskov Substitution design principle. Other chapters will touch on other design principles.

Logical view

Here's the overview of some of the classes shown in the previous chapter's case study. An important omission from those definitions was the `classify` algorithm of the `Hyperparameter` class:

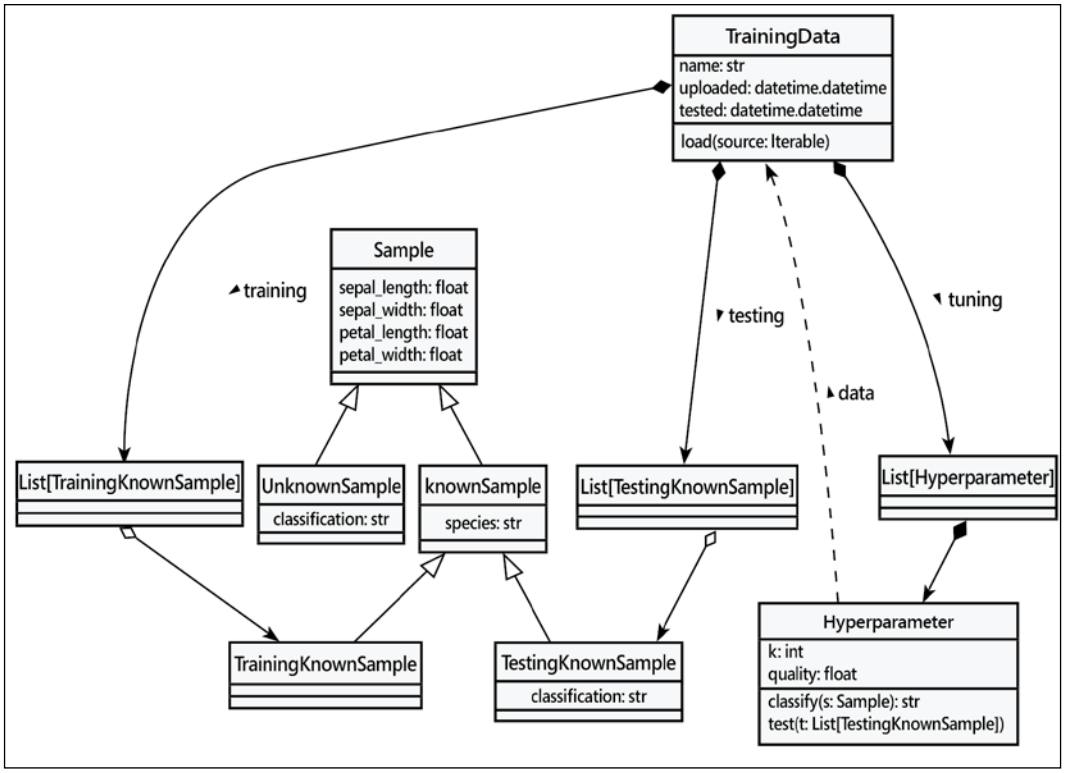

Figure 3.3: Class overview

In the previous chapter, we avoided delving into the classification algorithm. This reflects a common design strategy, sometimes called "*Hard Part, Do Later*," also called "*Do The Easy Part First*." This strategy encourages following common design patterns where possible to isolate the hard part. In effect, the easy parts define a number of fences that enclose and constrain the novel and unknown parts.

The classification we're doing is based on the *k*-nearest neighbors algorithm, *k*-NN. Given a set of known samples, and an unknown sample, we want to find neighbors near the unknown sample; the majority of the neighbors tells us how to classify the newcomer. This means *k* is usually an odd number, so the majority is easy to compute. We've been avoiding the question, "What do we mean by nearest?"

In a conventional, two-dimensional geometric sense, we can use the "Euclidean" distance between samples. Given an Unknown sample located at (u_x, u_y) and a Training sample at (t_x, t_y), the Euclidean distance between these samples, $ED2(t,u)$, is:

$$ED2(t,u) = \sqrt{(t_x - u_x)^2 + \left(t_y - u_y\right)^2}$$

We can visualize it like this:

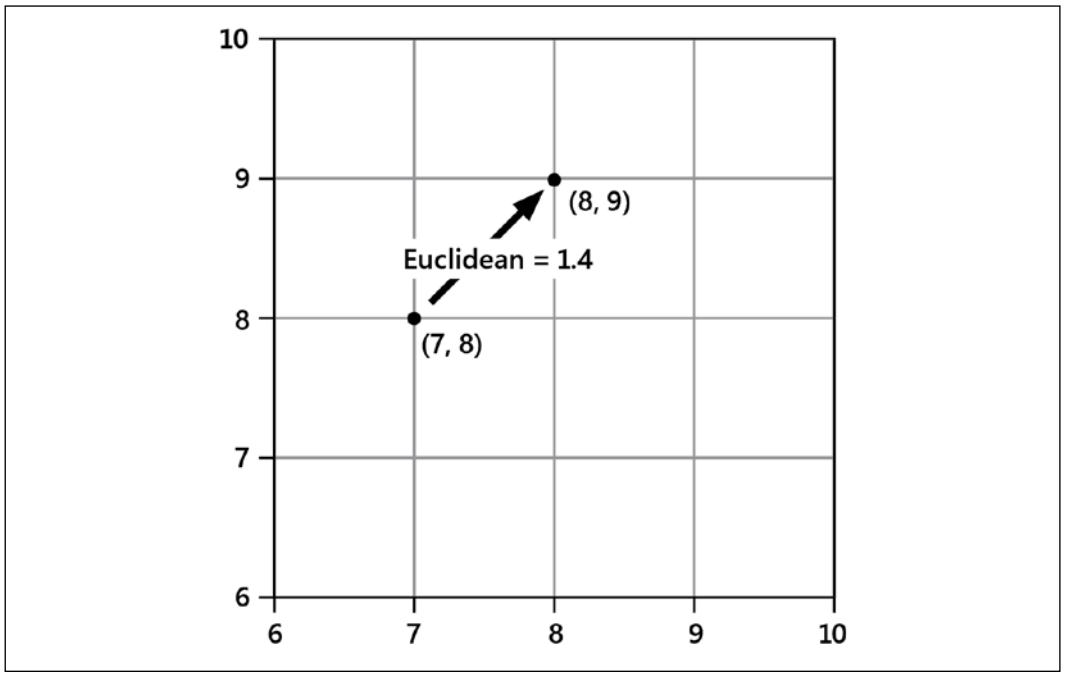

Figure 3.4: Euclidean distance

We've called this ED2 because it's only two-dimensional. In our case study data, we actually have four dimensions: sepal length, sepal width, petal length, and petal width. This is really difficult to visualize, but the math isn't too complex. Even when it's hard to imagine, we can still write it out fully, like so:

$$ED4(t,u) = \sqrt{(t_{sl} - u_{sl})^2 + (t_{sw} - u_{sw})^2 + \left(t_{pl} - u_{pl}\right)^2 + \left(t_{pw} - u_{pw}\right)^2}$$

All of the two-dimensional examples expand to four dimensions, in spite of how hard it is to imagine. We'll stick with the easier to visualize *x*-*y* distance for the diagrams in this section. But we really mean the full four-dimensional computation that includes all of the available measurements.

We can capture this computation as a class definition. An instance of this `ED` class is usable by the `Hyperparameter` class:

```
class ED(Distance):
    def distance(self, s1: Sample, s2: Sample) -> float:
        return hypot(
            s1.sepal_length - s2.sepal_length,
            s1.sepal_width - s2.sepal_width,
            s1.petal_length - s2.petal_length,
            s1.petal_width - s2.petal_width,
        )
```

We've leveraged the `math.hypot()` function to do the square and square root parts of the distance computation. We've used a superclass, `Distance`, that we haven't defined yet. We're pretty sure it's going to be needed, but we'll hold off a bit on defining it.

The Euclidean distance is one of many alternative definitions of distance between a known and unknown sample. There are two relatively simple ways to compute a distance that are similar, and they often produce consistently good results without the complexity of a square root:

- **Manhattan distance**: This is the distance you would walk in a city with square blocks (somewhat like parts of the city of Manhattan.)
- **Chebyshev distance**: This counts a diagonal step as 1. A Manhattan computation would rank this as 2. The Euclidean distance would be $\sqrt{2} \approx 1.41$, as depicted in *Figure 3.4*.

With a number of alternatives, we're going to need to create distinct subclasses. That means we'll need a base class to define the general idea of distances. Looking over the definitions at hand, it seems like the base class can be the following:

```
class Distance:
    """Definition of a distance computation"""
    def distance(self, s1: Sample, s2: Sample) -> float:
        pass
```

This seems to capture the essence of the distance computations we've seen. Let's implement a few more subclasses of this to be sure the abstraction really works.

The Manhattan distance is the total number of steps along the x-axis, plus the total number of steps along the y-axis. The formula uses the absolute values of the distances, written as $|t_x - u_x|$, and looks like this:

$$MD(t,u) = |t_x - u_x| + |t_y - u_y|$$

This can be as much as 41% larger than the direct Euclidean distance. However, it will still parallel the direct distance in a way that can yield a good k-NN result, but with a faster computation because it avoids squaring numbers and computing a square root.

Here's a view of the Manhattan distance:

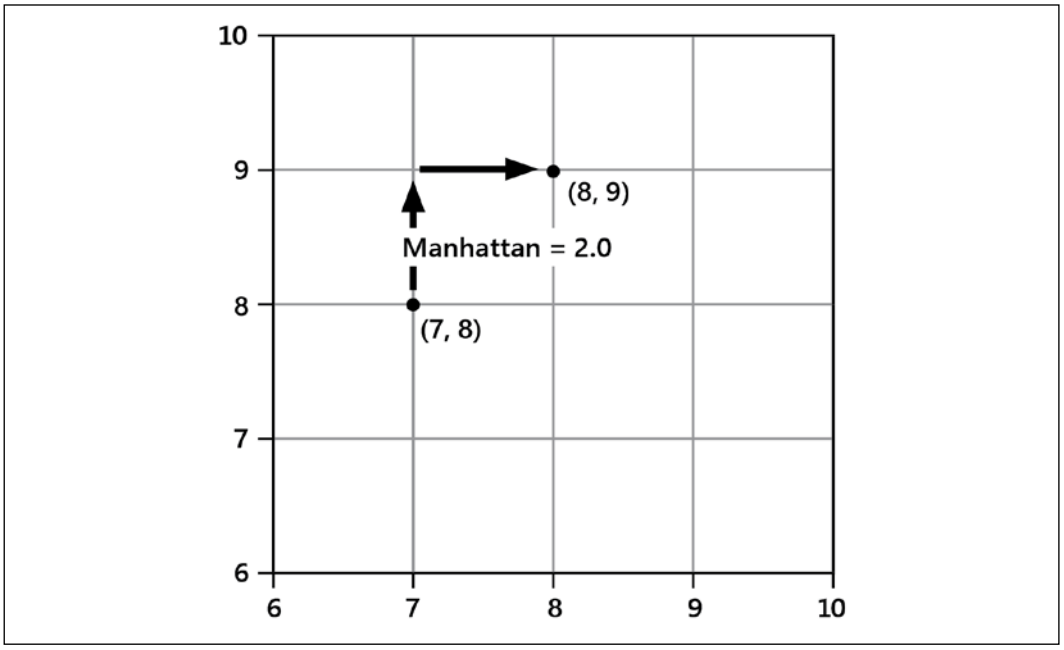

Figure 3.5: Manhattan distance

Here's a subclass of `Distance` that computes this variation:

```
class MD(Distance):
    def distance(self, s1: Sample, s2: Sample) -> float:
        return sum(
            [
```

```
                abs(s1.sepal_length - s2.sepal_length),
                abs(s1.sepal_width - s2.sepal_width),
                abs(s1.petal_length - s2.petal_length),
                abs(s1.petal_width - s2.petal_width),
            ]
        )
```

The Chebyshev distance is the largest of the absolute x or y distances. This tends to minimize the effects of multiple dimensions:

$$CD(k,u) = max\ (|k_x - u_x|, |k_y - u_y|)$$

Here's a view of the Chebyshev distance; it tends to emphasize neighbors that are closer to each other:

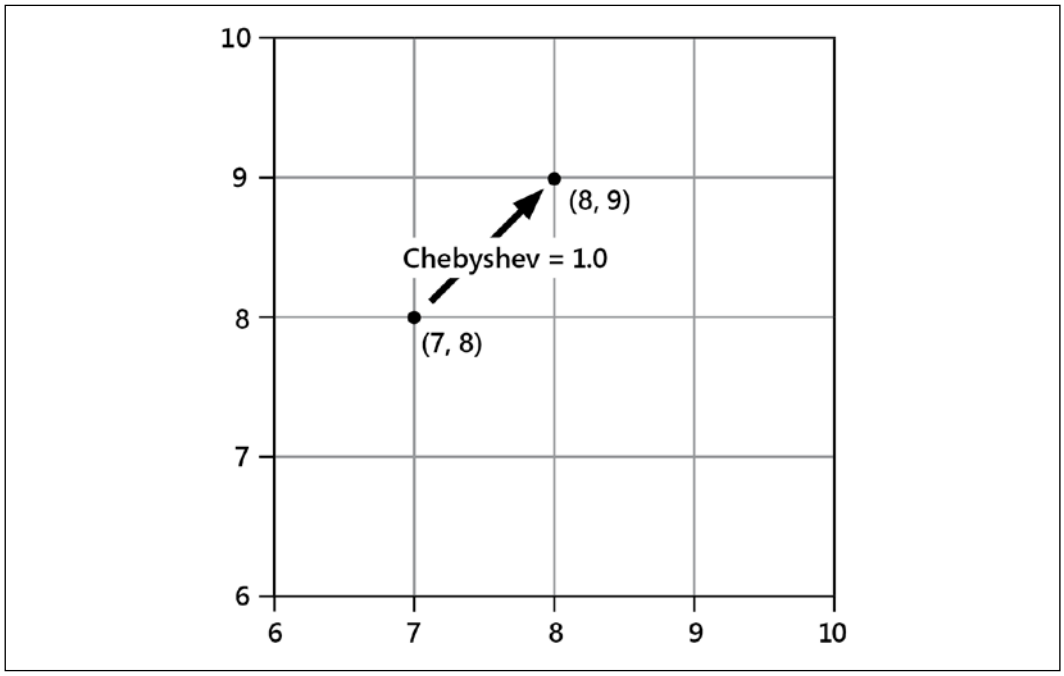

Figure 3.6: Chebyshev distance

Here's a subclass of `Distance` that performs this variant on the distance computation:

```
class CD(Distance())
    def distance(self, s1: Sample, s2: Sample) -> float:
        return sum(
            [
                abs(s1.sepal_length - s2.sepal_length),
                abs(s1.sepal_width - s2.sepal_width),
                abs(s1.petal_length - s2.petal_length),
                abs(s1.petal_width - s2.petal_width),
            ]
        )
```

See *Effects of Distance Measure Choice on KNN Classifier Performance - A Review* (`https://arxiv.org/pdf/1708.04321.pdf`). This paper contains 54 distinct metrics computations. The examples we're looking at are collectively identified as "Minkowski" measures because they're similar and measure each axis equally. Each alternative distance strategy yields different results in the model's ability to classify unknown samples given a set of training data.

This changes the idea behind the `Hyperparameter` class: we now have two distinct hyperparameters. The value of k, to decide how many neighbors to examine, and the distance computation, which tells us how to compute "nearest." These are both changeable parts of the algorithm, and we'll need to test various combinations to see which works best for our data.

How can we have all of these different distance computations available? The short answer is we'll need a lot of subclass definitions of a common distance class. The review paper cited above lets us pare down the domain to a few of the more useful distance computations. To be sure we've got a good design, let's look at one more distance.

Another distance

Just to make it clear how easy it is to add subclasses, we'll define a somewhat more complex distance metric. This is the Sorensen distance, also known as Bray-Curtis. If our distance class can handle these kinds of more complex formulas, we can be confident it's capable of handling others:

$$SD(k,u) = \frac{|k_x - u_x| + |k_y - u_y|}{(k_x + u_x) + (k_y + u_y)}$$

We've effectively standardized each component of the Manhattan distance by dividing by the possible range of values.

Here's a diagram to illustrate how the Sorensen distance works:

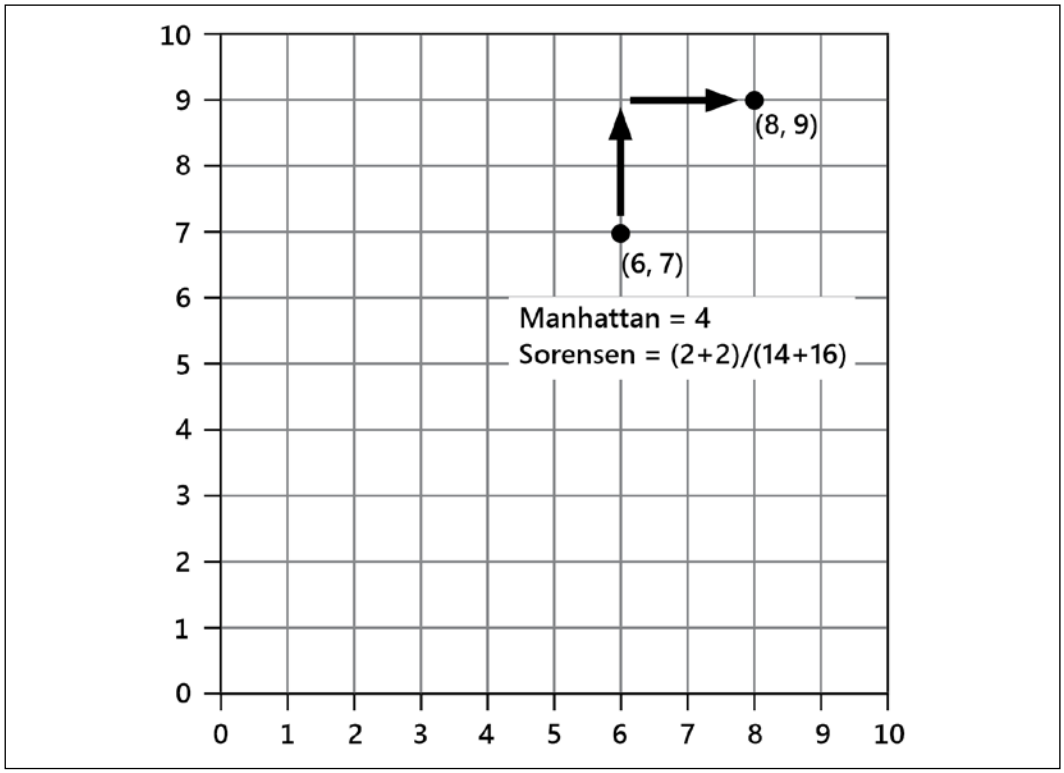

Figure 3.7: Manhattan versus Sorensen distance

The simple Manhattan distance applies no matter how far from the origin we are. The Sorensen distance reduces the importance of measures that are further from the origin so they don't dominate the *k*-NN by virtue of being large-valued outliers.

We can introduce this into our design by adding a new subclass of `Distance`. While this is similar, in some ways, to the Manhattan distance, it's often classified separately:

```
class SD(Distance):
    def distance(self, s1: Sample, s2: Sample) -> float:
        return sum(
            [
```

```
                abs(s1.sepal_length - s2.sepal_length),
                abs(s1.sepal_width - s2.sepal_width),
                abs(s1.petal_length - s2.petal_length),
                abs(s1.petal_width - s2.petal_width),
            ]
        ) / sum(
            [
                s1.sepal_length + s2.sepal_length,
                s1.sepal_width + s2.sepal_width,
                s1.petal_length + s2.petal_length,
                s1.petal_width + s2.petal_width,
            ]
        )
```

This design approach lets us leverage object-oriented inheritance to build a polymorphic family of distance computation functions. We can build on the first few functions to create a wide family of functions and use these as part of hyperparameter tuning to locate the best way to measure distances and perform the required classification.

We'll need to integrate a `Distance` object into the `Hyperparameter` class. This means providing an instance of one of these subclasses. Because they're all implementing the same `distance()` method, we can replace different alternative distance computations to find which performs best with our unique collection of data and attributes.

For now, we can reference a specific distance subclass in our `Hyperparameter` class definition. In *Chapter 11*, *Common Design Patterns*, we'll look at how we can flexibly plug in any possible distance computation from the hierarchy of `Distance` class definitions.

Recall

Some key points in this chapter:

- A central object-oriented design principle is inheritance: a subclass can inherit aspects of a superclass, saving copy-and-paste programming. A subclass can extend the superclass to add features or specialize the superclass in other ways.
- Multiple inheritance is a feature of Python. The most common form is a host class with mixin class definitions. We can combine multiple classes leveraging the method resolution order to handle common features like initialization.

- Polymorphism lets us create multiple classes that provide alternative implementations for fulfilling a contract. Because of Python's duck typing rules, any classes that have the right methods can substitute for each other.

Exercises

Look around you at some of the physical objects in your workspace and see if you can describe them in an inheritance hierarchy. Humans have been dividing the world into taxonomies like this for centuries, so it shouldn't be difficult. Are there any non-obvious inheritance relationships between classes of objects? If you were to model these objects in a computer application, what properties and methods would they share? Which ones would have to be polymorphically overridden? What properties would be completely different between them?

Now write some code. No, not for the physical hierarchy; that's boring. Physical items have more properties than methods. Just think about a pet programming project you've wanted to tackle in the past year, but never gotten around to. For whatever problem you want to solve, try to think of some basic inheritance relationships and then implement them. Make sure that you also pay attention to the sorts of relationships that you actually don't need to use inheritance for. Are there any places where you might want to use multiple inheritance? Are you sure? Can you see any place where you would want to use a mixin? Try to knock together a quick prototype. It doesn't have to be useful or even partially working. You've seen how you can test code using `python -i` already; just write some code and test it in the interactive interpreter. If it works, write some more. If it doesn't, fix it!

Now, take a look at the various distance computations in the case study. We need to be able to work with testing data as well as unknown samples provided by a user. What do these two kinds of samples have in common? Can you create a common superclass and use inheritance for these two classes with similar behavior? (We haven't looked closely at the *k*-NN classification yet, but you can provide a "mock" classifier that will provide fake answers.)

When we look at the distance computation, we can see how a `Hyperparameter` is a composition that includes a distance algorithm plug-in as one of the parameters. Is this a good candidate for a mixin? Why or why not? What limitations does a mixin have that a plug-in does not have?

Summary

We've gone from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox, all the way through to multiple inheritance – one of the most complicated. Inheritance can be used to add functionality to existing classes and built-in generics. Abstracting similar code into a parent class can help increase maintainability. Methods on parent classes can be called using `super`, and argument lists must be formatted safely for these calls to work when using multiple inheritance.

In the next chapter, we'll cover the subtle art of handling exceptional circumstances.