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:

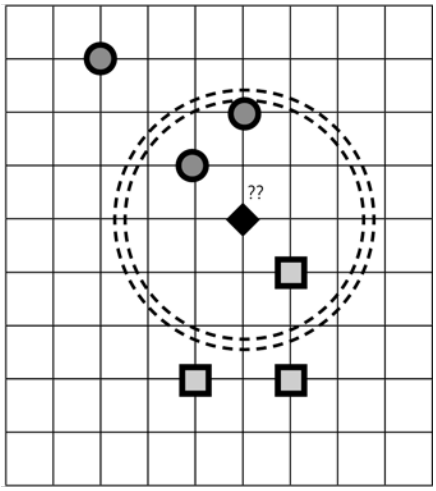

Figure 1.11: k-nearest neighbors

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:

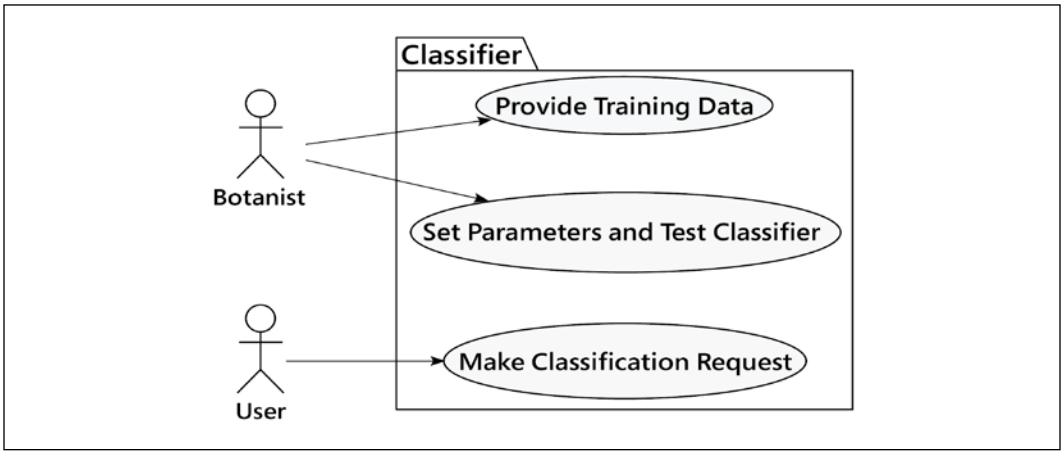

Figure 1.12: UML context diagram

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:

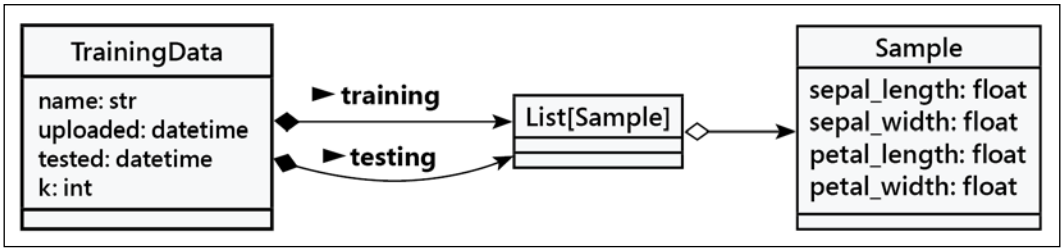

Figure 1.13: Class diagram for training and testing

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:

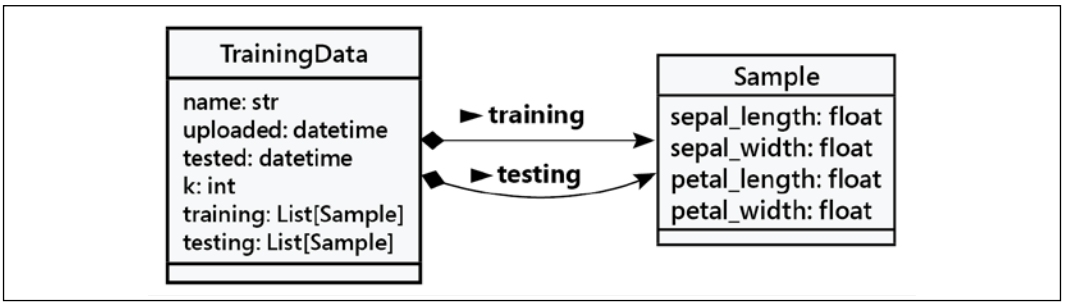

Figure 1.14: Condensed class diagram

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:

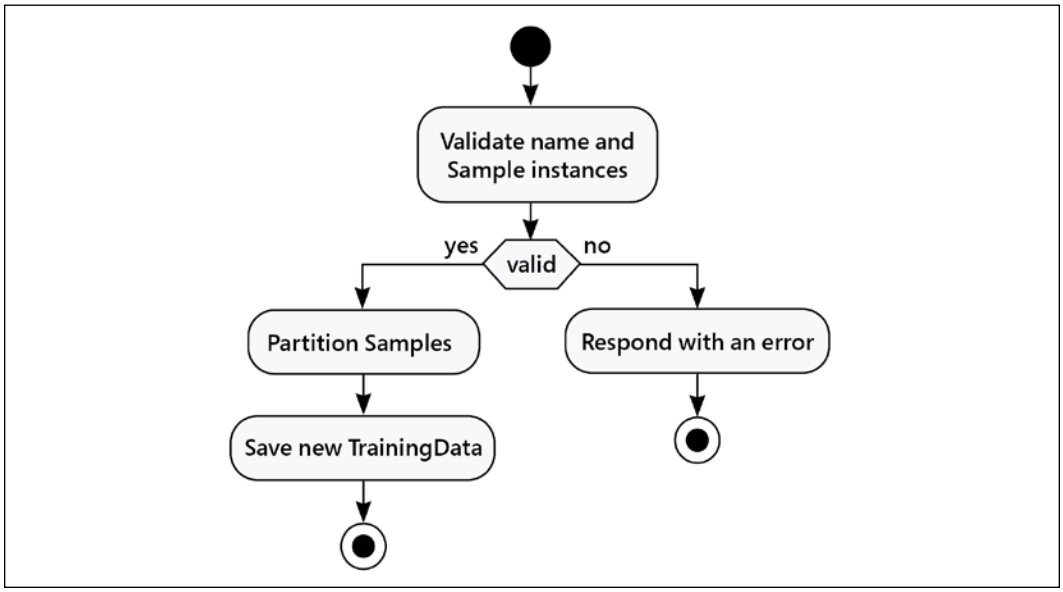

Figure 1.15: Activity diagram

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:

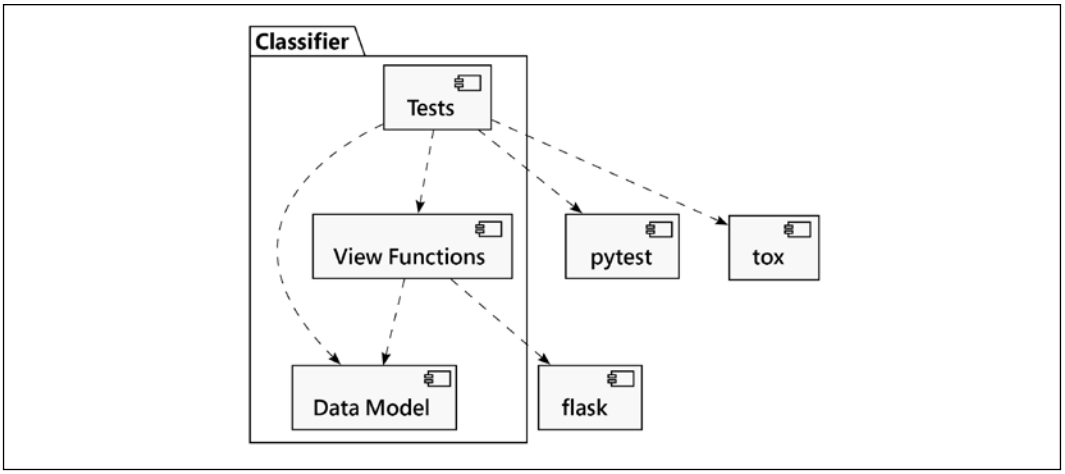

Figure 1.16: Components to be built

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:

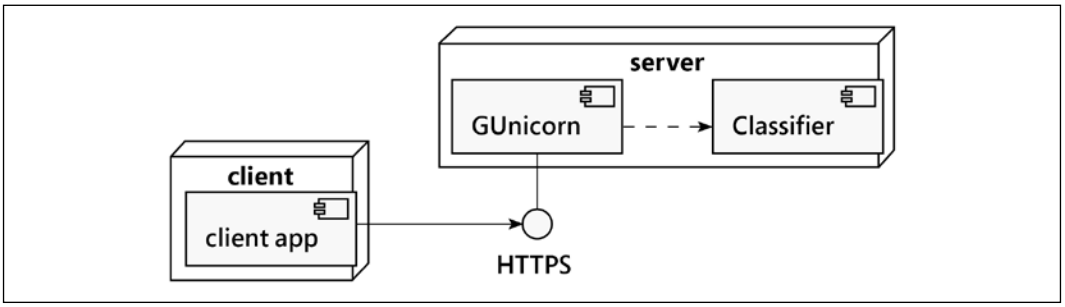

Figure 1.17: Application server diagram

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:

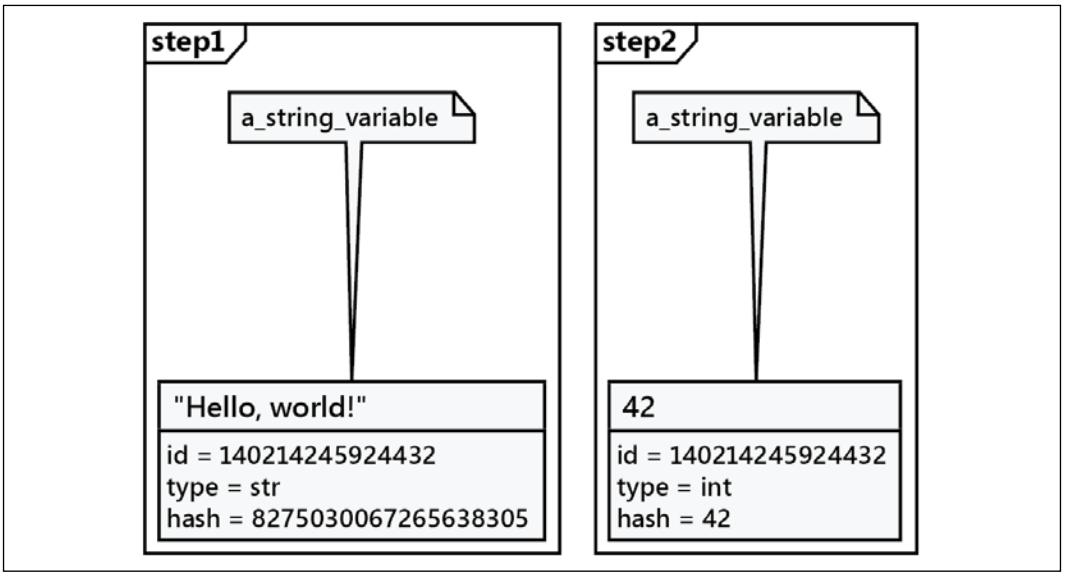

Figure 2.1: Variable names and objects

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:

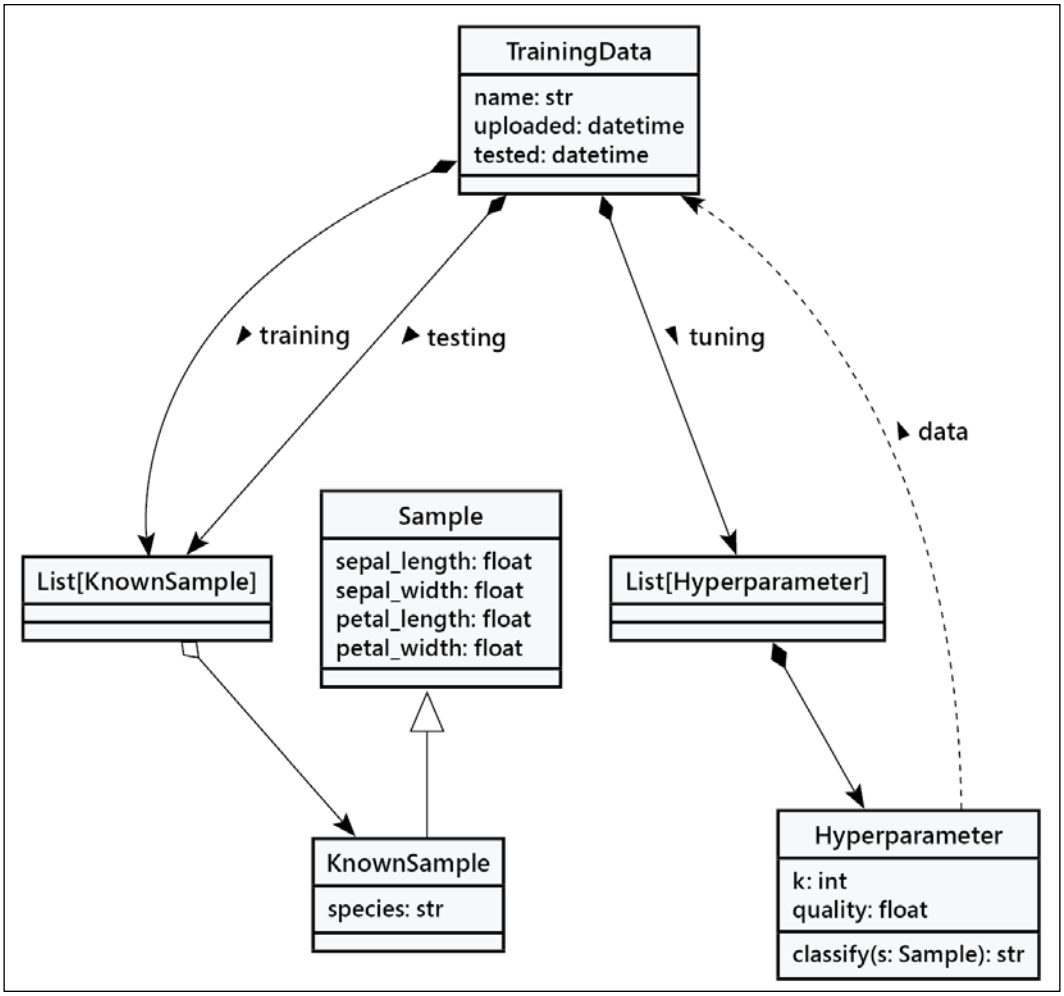

Figure 2.2: Logical view diagram

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:

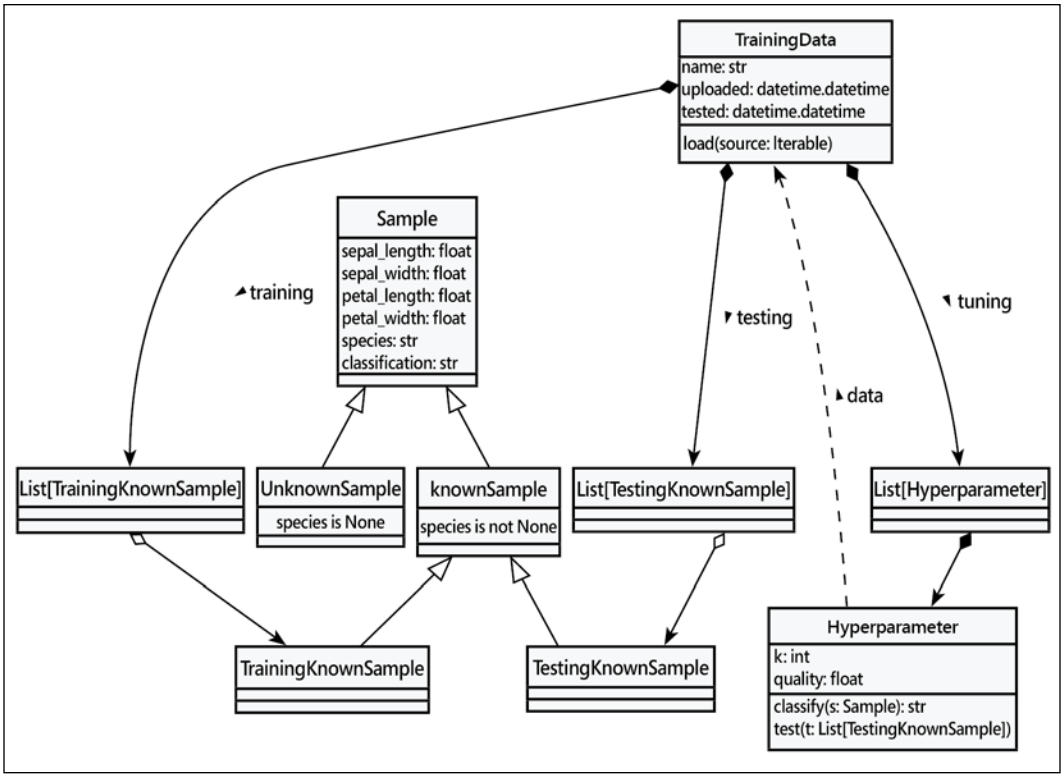

Figure 2.3: The updated UML diagram

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"
```