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

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

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

Organizing modules

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

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

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

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

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

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

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

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

Absolute imports

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

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

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

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

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

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

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

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

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

Relative imports

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

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

```
from .database import Database
```

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

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

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

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

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

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

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

Packages as a whole

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

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

```
from .database import db
```

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

```
from ecommerce import db
```

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

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

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

Organizing our code in modules

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

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

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

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

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

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

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

```
db: Optional[Database] = None

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

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

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

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

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

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

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

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

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


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

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


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

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

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

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

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

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

```
from typing import Optional

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

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

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

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

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)
```

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

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

We can execute this function like this:

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

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

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

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

Who can access my data?

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

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

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

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

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

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

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

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

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

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

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

Third-party libraries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Case study

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

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

Logical view

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

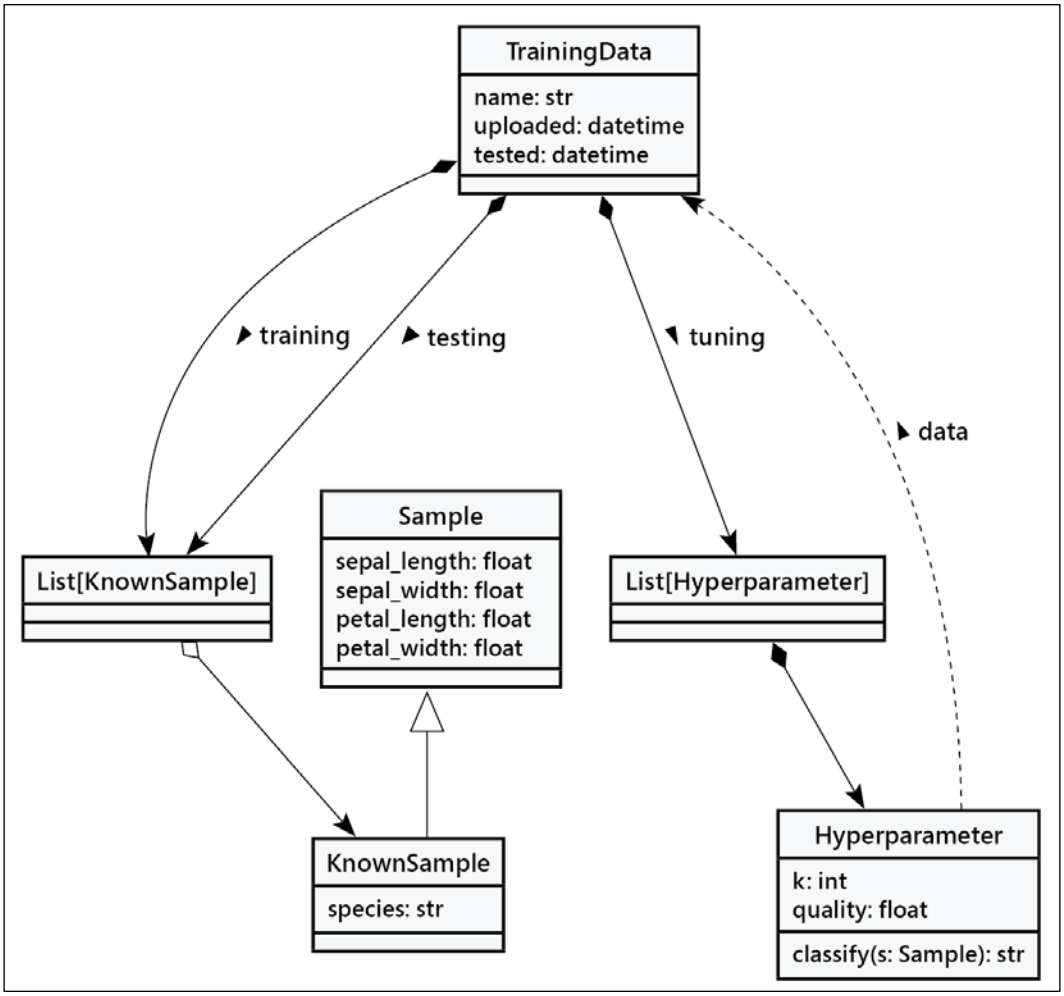

Figure 2.2: Logical view diagram

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

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

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

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

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

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

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

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

Samples and their states

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

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

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

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

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

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

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

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

This leads to a second important question:

What class has responsibility for making this state change?

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

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

Sample state transitions

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

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

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

There's no "right" answer.

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

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

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

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

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

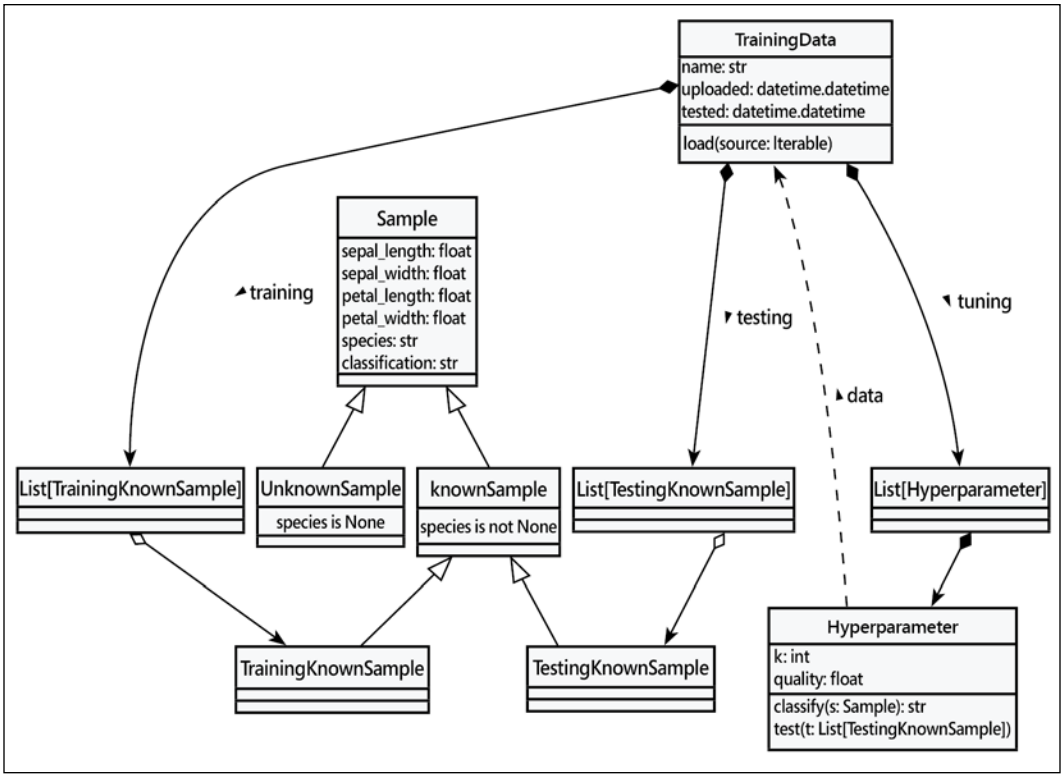

Figure 2.3: The updated UML diagram

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

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

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

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

```
class Sample:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Class responsibilities

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

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

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

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

Here's the start of the class definition:

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

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

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

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

The testing is defined by the following method:

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

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

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

The TrainingData class

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

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

This class has the two methods that initiate the processing:

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

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

Here's the start of the class definition:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Recall

Some key points in this chapter:

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

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

Exercises

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

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

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

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

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

Summary

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

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

3
When Objects Are Alike

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

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

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

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

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

Basic inheritance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Extending built-ins

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

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

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

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

class Contact:
    all_contacts = ContactList()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Overriding and super

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

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

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

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

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

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

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

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

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

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

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

Multiple inheritance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The diamond problem

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

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

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

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

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

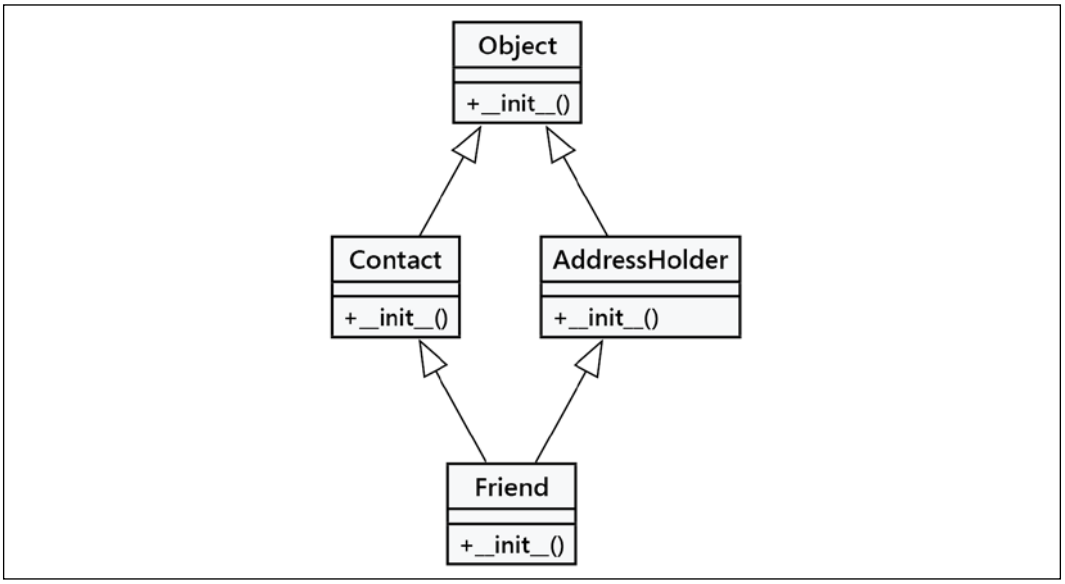

Figure 3.1: Inheritance diagram for our multiple inheritance implementation

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

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

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

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

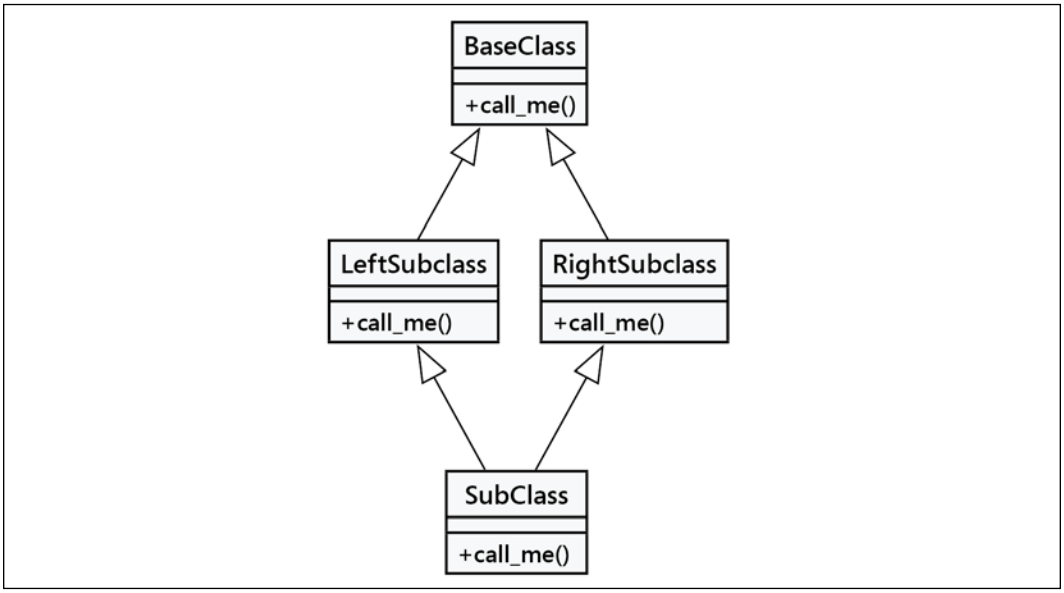

Figure 3.2: Diamond inheritance

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

```
class BaseClass:
    num_base_calls = 0

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

class LeftSubclass(BaseClass):
    num_left_calls = 0

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

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

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

class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

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

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

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

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

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

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

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

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

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

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

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

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

```
class BaseClass:
    num_base_calls = 0

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

class LeftSubclass_S(BaseClass):
    num_left_calls = 0

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

class RightSubclass_S(BaseClass):
```

```
    num_right_calls = 0

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

class Subclass_S(LeftSubclass_S, RightSubclass_S):
    num_sub_calls = 0

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

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

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

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

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

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

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

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

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

Different sets of arguments

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

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

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

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

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

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

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

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

```
class Contact:
    all_contacts = ContactList()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Polymorphism

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

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

```
audio_file.play()
```

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

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

```
from pathlib import Path

class AudioFile:
    ext: str
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Case study

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

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

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

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

Logical view

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

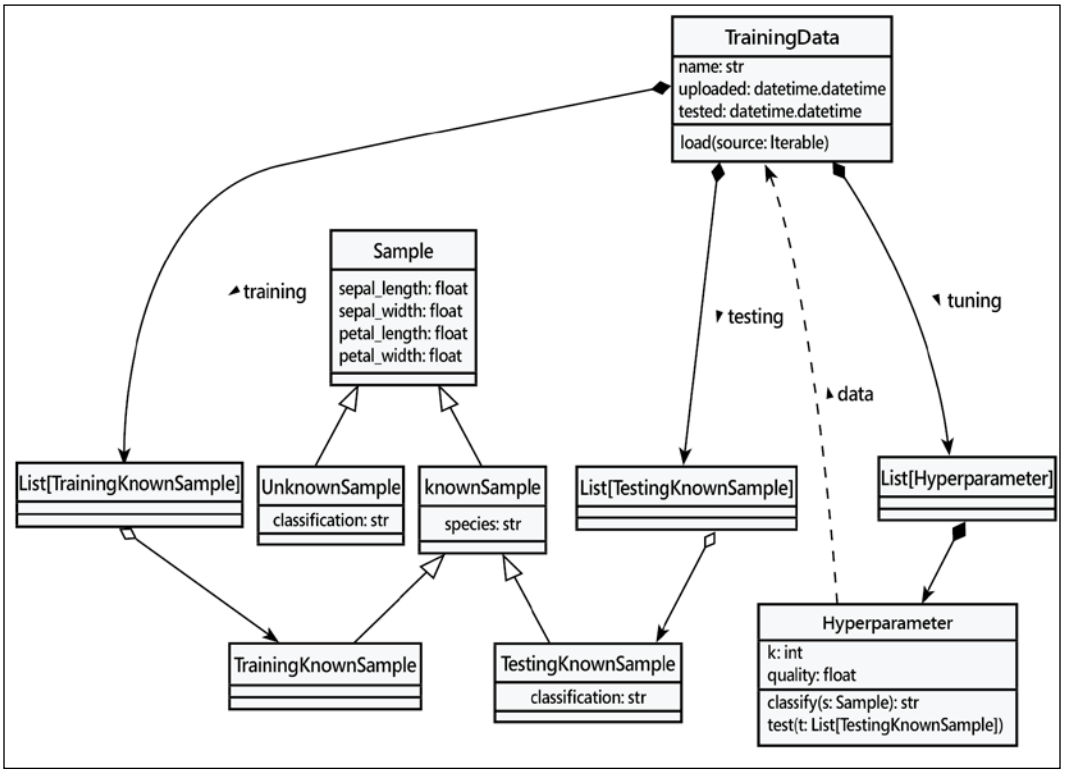

Figure 3.3: Class overview

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

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

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

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

We can visualize it like this:

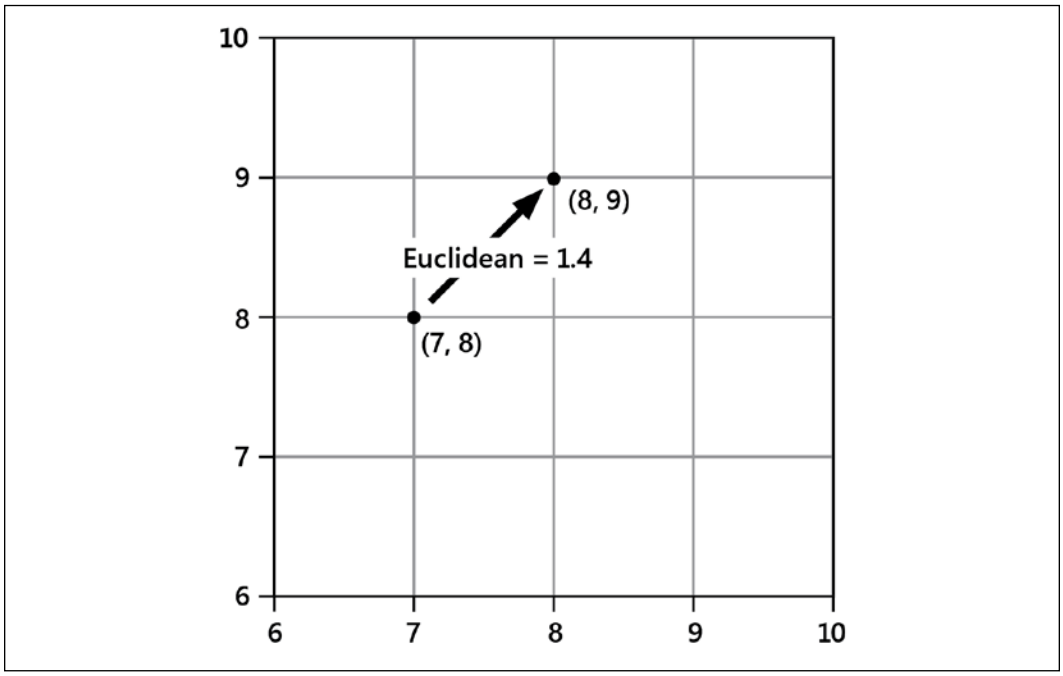

Figure 3.4: Euclidean distance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here's a view of the Manhattan distance:

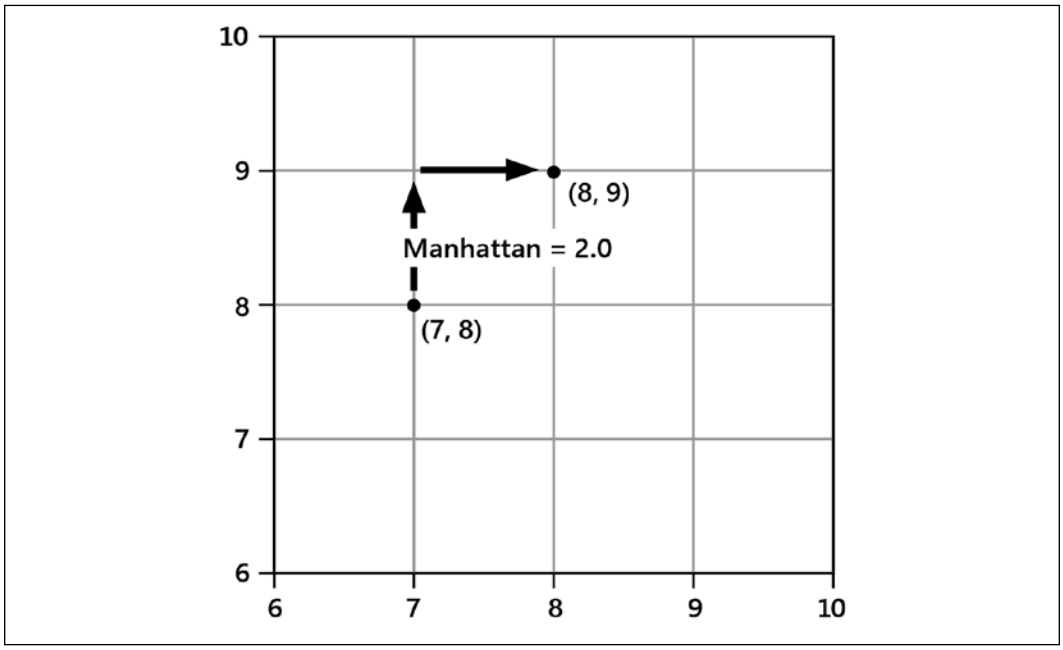

Figure 3.5: Manhattan distance

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

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

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

The Chebyshev distance is the largest of the absolute x or y distances. This tends to minimize the effects of multiple dimensions:

$$CD(k,u) = max\ (|k_x - u_x|, |k_y - u_y|)$$

Here's a view of the Chebyshev distance; it tends to emphasize neighbors that are closer to each other:

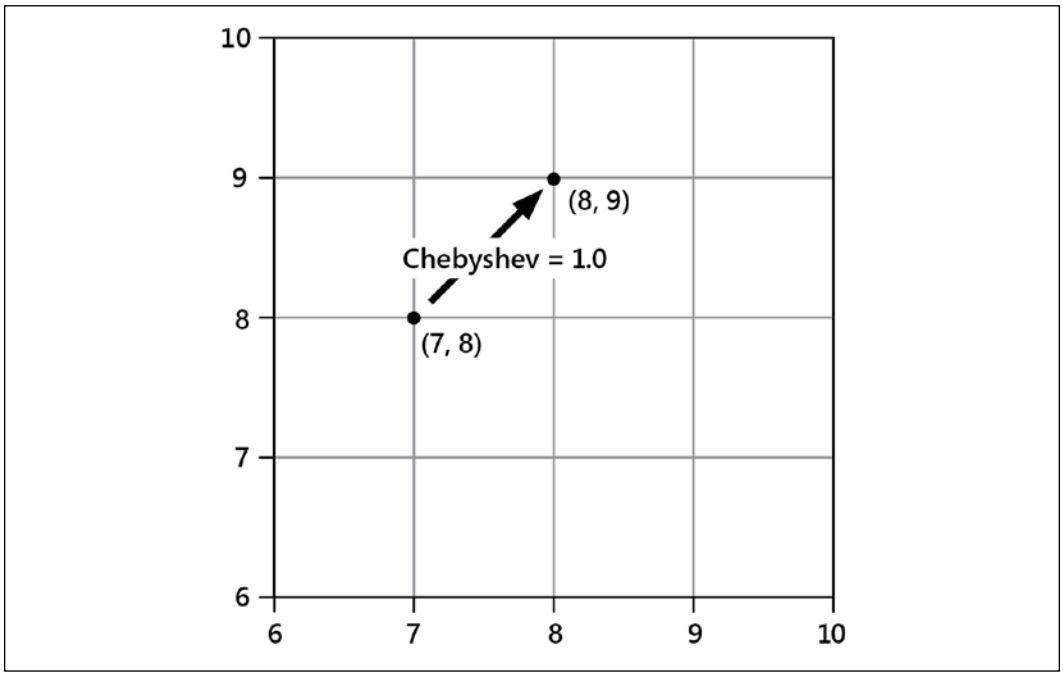

Figure 3.6: Chebyshev distance

Here's a subclass of `Distance` that performs this variant on the distance computation:

```
class CD(Distance())
    def distance(self, s1: Sample, s2: Sample) -> float:
        return sum(
            [
                abs(s1.sepal_length - s2.sepal_length),
                abs(s1.sepal_width - s2.sepal_width),
                abs(s1.petal_length - s2.petal_length),
                abs(s1.petal_width - s2.petal_width),
            ]
        )
```

See *Effects of Distance Measure Choice on KNN Classifier Performance - A Review* (`https://arxiv.org/pdf/1708.04321.pdf`). This paper contains 54 distinct metrics computations. The examples we're looking at are collectively identified as "Minkowski" measures because they're similar and measure each axis equally. Each alternative distance strategy yields different results in the model's ability to classify unknown samples given a set of training data.

This changes the idea behind the `Hyperparameter` class: we now have two distinct hyperparameters. The value of *k*, to decide how many neighbors to examine, and the distance computation, which tells us how to compute "nearest." These are both changeable parts of the algorithm, and we'll need to test various combinations to see which works best for our data.

How can we have all of these different distance computations available? The short answer is we'll need a lot of subclass definitions of a common distance class. The review paper cited above lets us pare down the domain to a few of the more useful distance computations. To be sure we've got a good design, let's look at one more distance.

Another distance

Just to make it clear how easy it is to add subclasses, we'll define a somewhat more complex distance metric. This is the Sorensen distance, also known as Bray-Curtis. If our distance class can handle these kinds of more complex formulas, we can be confident it's capable of handling others:

$$SD(k,u) = \frac{|k_x - u_x| + |k_y - u_y|}{(k_x + u_x) + (k_y + u_y)}$$

We've effectively standardized each component of the Manhattan distance by dividing by the possible range of values.

Here's a diagram to illustrate how the Sorensen distance works:

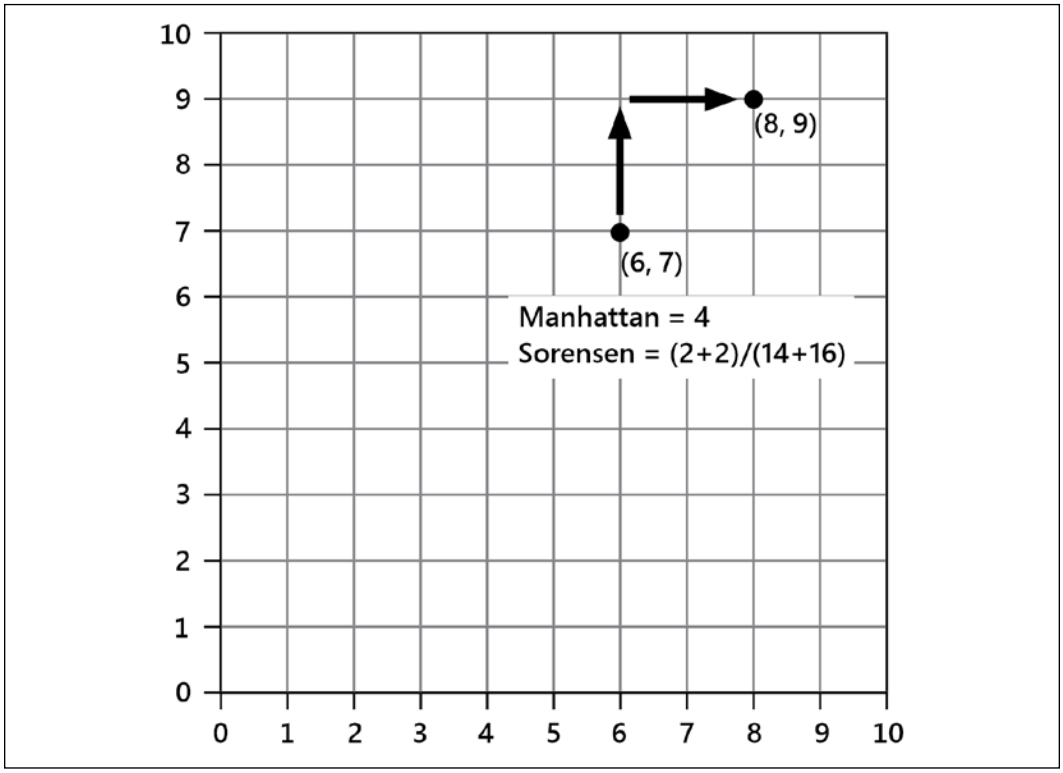

Figure 3.7: Manhattan versus Sorensen distance

The simple Manhattan distance applies no matter how far from the origin we are. The Sorensen distance reduces the importance of measures that are further from the origin so they don't dominate the *k*-NN by virtue of being large-valued outliers.

We can introduce this into our design by adding a new subclass of `Distance`. While this is similar, in some ways, to the Manhattan distance, it's often classified separately:

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

```
                abs(s1.sepal_length - s2.sepal_length),
                abs(s1.sepal_width - s2.sepal_width),
                abs(s1.petal_length - s2.petal_length),
                abs(s1.petal_width - s2.petal_width),
            ]
        ) / sum(
            [
                s1.sepal_length + s2.sepal_length,
                s1.sepal_width + s2.sepal_width,
                s1.petal_length + s2.petal_length,
                s1.petal_width + s2.petal_width,
            ]
        )
```

This design approach lets us leverage object-oriented inheritance to build a polymorphic family of distance computation functions. We can build on the first few functions to create a wide family of functions and use these as part of hyperparameter tuning to locate the best way to measure distances and perform the required classification.

We'll need to integrate a `Distance` object into the `Hyperparameter` class. This means providing an instance of one of these subclasses. Because they're all implementing the same `distance()` method, we can replace different alternative distance computations to find which performs best with our unique collection of data and attributes.

For now, we can reference a specific distance subclass in our `Hyperparameter` class definition. In *Chapter 11, Common Design Patterns*, we'll look at how we can flexibly plug in any possible distance computation from the hierarchy of `Distance` class definitions.

Recall

Some key points in this chapter:

- A central object-oriented design principle is inheritance: a subclass can inherit aspects of a superclass, saving copy-and-paste programming. A subclass can extend the superclass to add features or specialize the superclass in other ways.
- Multiple inheritance is a feature of Python. The most common form is a host class with mixin class definitions. We can combine multiple classes leveraging the method resolution order to handle common features like initialization.

- Polymorphism lets us create multiple classes that provide alternative implementations for fulfilling a contract. Because of Python's duck typing rules, any classes that have the right methods can substitute for each other.

Exercises

Look around you at some of the physical objects in your workspace and see if you can describe them in an inheritance hierarchy. Humans have been dividing the world into taxonomies like this for centuries, so it shouldn't be difficult. Are there any non-obvious inheritance relationships between classes of objects? If you were to model these objects in a computer application, what properties and methods would they share? Which ones would have to be polymorphically overridden? What properties would be completely different between them?

Now write some code. No, not for the physical hierarchy; that's boring. Physical items have more properties than methods. Just think about a pet programming project you've wanted to tackle in the past year, but never gotten around to. For whatever problem you want to solve, try to think of some basic inheritance relationships and then implement them. Make sure that you also pay attention to the sorts of relationships that you actually don't need to use inheritance for. Are there any places where you might want to use multiple inheritance? Are you sure? Can you see any place where you would want to use a mixin? Try to knock together a quick prototype. It doesn't have to be useful or even partially working. You've seen how you can test code using `python -i` already; just write some code and test it in the interactive interpreter. If it works, write some more. If it doesn't, fix it!

Now, take a look at the various distance computations in the case study. We need to be able to work with testing data as well as unknown samples provided by a user. What do these two kinds of samples have in common? Can you create a common superclass and use inheritance for these two classes with similar behavior? (We haven't looked closely at the *k*-NN classification yet, but you can provide a "mock" classifier that will provide fake answers.)

When we look at the distance computation, we can see how a `Hyperparameter` is a composition that includes a distance algorithm plug-in as one of the parameters. Is this a good candidate for a mixin? Why or why not? What limitations does a mixin have that a plug-in does not have?

Summary

We've gone from simple inheritance, one of the most useful tools in the object-oriented programmer's toolbox, all the way through to multiple inheritance – one of the most complicated. Inheritance can be used to add functionality to existing classes and built-in generics. Abstracting similar code into a parent class can help increase maintainability. Methods on parent classes can be called using `super`, and argument lists must be formatted safely for these calls to work when using multiple inheritance.

In the next chapter, we'll cover the subtle art of handling exceptional circumstances.

4

Expecting the Unexpected

Systems built with software can be fragile. While the software is highly predictable, the runtime context can provide unexpected inputs and situations. Devices fail, networks are unreliable, mere anarchy is loosed on our application. We need to have a way to work around the spectrum of failures that plague computer systems.

There are two broad approaches to dealing with the unforeseen. One approach is to return a recognizable error-signaling value from a function. A value, like `None`, could be used. Other library functions can then be used by an application to retrieve details of the erroneous condition. A variation on this theme is to pair a return from an OS request with a success or failure indicator. The other approach is to interrupt the normal, sequential execution of statements and divert to statements that handle exceptions. This second approach is what Python does: it eliminates the need to check return values for errors.

In this chapter, we will study **exceptions**, special error objects raised when a normal response is impossible. In particular, we will cover the following:

- How to cause an exception to occur
- How to recover when an exception has occurred
- How to handle different exception types in different ways
- Cleaning up when an exception has occurred
- Creating new types of exception
- Using the exception syntax for flow control

The case study for this chapter will look at data validation. We'll examine a number of ways exceptions can be used to ensure that inputs to our classifier are valid.

We'll start by looking at Python's concept of an `Exception`, and how exceptions are raised and handled.

Raising exceptions

Python's normal behavior is to execute statements in the order they are found, either in a file or at the `>>>` prompt interactively. A few statements, specifically `if`, `while`, and `for`, alter the simple top-to-bottom sequence of statement execution. Additionally, an exception can break the sequential flow of execution. Exceptions are raised, and this interrupts the sequential execution of statements.

In Python, the exception that's raised is also an object. There are many different exception classes available, and we can easily define more of our own. The one thing they all have in common is that they inherit from a built-in class called `BaseException`.

When an exception is raised, everything that was supposed to happen is pre-empted. Instead, exception handling replaces normal processing. Make sense? Don't worry, it will!

The easiest way to cause an exception to occur is to do something silly. Chances are you've done this already and seen the exception output. For example, any time Python encounters a line in your program that it can't understand, it bails with `SyntaxError`, which is a type of exception. Here's a common one:

```
>>> print "hello world"
  File "<input>", line 1
    print "hello world"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print("hello world")?
```

The `print()` function requires the arguments to be enclosed in parentheses. So, if we type the preceding command into a Python 3 interpreter, we raise a `SyntaxError` exception.

In addition to `SyntaxError`, some other common exceptions are shown in the following example:

```
>>> x = 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> lst = [1,2,3]
>>> print(lst[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> lst + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> lst.add
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'

>>> d = {'a': 'hello'}
>>> d['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'

>>> print(this_is_not_a_var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'this_is_not_a_var' is not defined
```

We can partition these exceptions into roughly four categories. Some cases are blurry, but some edges have a bright line separating them:

- Sometimes, these exceptions are indicators of something clearly wrong in our program. Exceptions like `SyntaxError` and `NameError` mean we need to find the indicated line number and fix the problem.
- Sometimes, these exceptions are indicators of something wrong in the Python runtime. There's a `RuntimeError` exception that can get raised. In many cases, this is resolved by downloading and installing a newer Python. (Or, if you're wrestling with a "Release Candidate" version, reporting the bug to the maintainers.)

- Some exceptions are design problems. We may fail to account for an edge case properly and sometimes try to compute an average of an empty list. This will result in a `ZeroDivisionError`. When we find these, again, we'll have to go to the indicated line number. But once we've found the resulting exception, we'll need to work backwards from there to find out what caused the problem that raised the exception. Somewhere there will be an object in an unexpected or not-designed-for state.
- The bulk of the exceptions arise near our program's interfaces. Any user input, or operating system request, including file operations, can encounter problems with the resources outside our program, leading to exceptions. We can subdivide these interface problems further into two sub-groups:
    - External objects in an unusual or unanticipated state. This is common with files that aren't found because the path was spelled incorrectly, or directories that already exist because our application crashed earlier and we restarted it. These will often be some kind of `OSError` with a reasonably clear root cause. It's also common with users entering things incorrectly, or even users maliciously trying to subvert the application. These should be application-specific exceptions to prevent dumb mistakes or intentional abuse.
    - And there's also the (relatively small) category of simple chaos. In the final analysis, a computer system is a lot of interconnected devices and any one of the components could behave badly. These are hard to anticipate and it's harder still to plan a recovery strategy. When working with a small IoT computer, there are few parts, but it may be installed in a challenging physical environment. When working with an enterprise server farm with thousands of components, a 0.1% failure rate means something is always broken.

You may have noticed all of Python's built-in exceptions end with the name `Error`. In Python, the words **error** and **exception** are used almost interchangeably. Errors are sometimes considered more dire than exceptions, but they are dealt with in exactly the same way. Indeed, all the error classes in the preceding example have `Exception` (which extends `BaseException`) as their superclass.

Raising an exception

We'll get to responding to such exceptions in a minute, but first, let's discover what we should do if we're writing a program that needs to inform the user or a calling function that the inputs are invalid. We can use the exact same mechanism that Python uses. Here's a simple class that adds items to a list only if they are even-numbered integers:

```
from typing import List

class EvenOnly(List[int]):
    def append(self, value: int) -> None:
        if not isinstance(value, int):
            raise TypeError("Only integers can be added")
        if value % 2 != 0:
            raise ValueError("Only even numbers can be added")
        super().append(value)
```

This class extends the built-in list, as we discussed in *Chapter 2, Objects in Python*. We've provided a type hint suggesting we're creating a list of integer objects only. To do this, we've overridden the `append` method to check two conditions that ensure the item is an even integer. We first check whether the input is an instance of the `int` type, and then use the modulo operator to ensure it is divisible by two. If either of the two conditions is not met, the `raise` keyword causes an exception to occur.

The `raise` keyword is followed by the object being raised as an exception. In the preceding example, two objects are constructed from the built-in `TypeError` and `ValueError` classes. The raised object could just as easily be an instance of a new `Exception` class we create ourselves (we'll see how shortly), an exception that was defined elsewhere, or even an `Exception` object that has been previously raised and handled.

If we test this class in the Python interpreter, we can see that it is outputting useful error information when exceptions occur, just as before:

```
>>> e = EvenOnly()
>>> e.append("a string")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 7, in add
    raise TypeError("Only integers can be added")
TypeError: Only integers can be added

>>> e.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 9, in add
    raise ValueError("Only even numbers can be added")
ValueError: Only even numbers can be added
>>> e.append(2)
```