There are several third-party tools for managing virtual environments effectively. Some of these include `pyenv`, `virtualenvwrapper`, and `conda`. My personal preference at the time of writing is `pyenv`, but there is no clear winner here. Do a quick web search and see what works for you.

Case study

To tie it all together, let's build a simple command-line notebook application. This is a fairly simple task, so we won't be experimenting with multiple packages. We will, however, see common usage of classes, functions, methods, and docstrings.

Let's start with a quick analysis: notes are short memos stored in a notebook. Each note should record the day it was written and can have tags added for easy querying. It should be possible to modify notes. We also need to be able to search for notes. All of these things should be done from the command line.

An obvious object is the `Note` object; a less obvious one is a `Notebook` container object. Tags and dates also seem to be objects, but we can use dates from Python's standard library and a comma-separated string for tags. To avoid complexity, in the prototype, we need not define separate classes for these objects.

`Note` objects have attributes for `memo` itself, `tags`, and `creation_date`. Each note will also need a unique integer `id` so that users can select them in a menu interface. Notes could have a method to modify note content and another for tags, or we could just let the notebook access those attributes directly. To make searching easier, we should put a `match` method on the `Note` object. This method will accept a string and can tell us whether a note matches the string without accessing the attributes directly. This way, if we want to modify the search parameters (to search tags instead of note contents, for example, or to make the search case-insensitive), we only have to do it in one place.

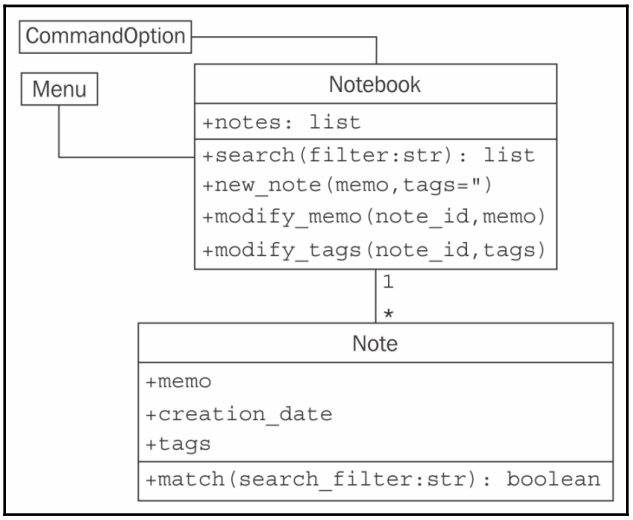
The `Notebook` object obviously has the list of notes as an attribute. It will also need a search method that returns a list of filtered notes.

But how do we interact with these objects? We've specified a command-line app, which can mean either that we run the program with different options to add or edit commands, or we have some kind of menu that allows us to pick different things to do to the notebook. We should try to design it such that either interface is supported and future interfaces, such as a GUI toolkit or web-based interface, could be added in the future.

As a design decision, we'll implement the menu interface now, but will keep the command-line options version in mind to ensure we design our `Notebook` class with extensibility in mind.

If we have two command-line interfaces, each interacting with the `Notebook` object, then `Notebook` will need some methods for those interfaces to interact with. We need to be able to `add` a new note, and `modify` an existing note by `id`, in addition to the `search` method we've already discussed. The interfaces will also need to be able to list all notes, but they can do that by accessing the `notes` list attribute directly.

We may be missing a few details, but we have a really good overview of the code we need to write. We can summarize all this analysis in a simple class diagram:



Before writing any code, let's define the folder structure for this project. The menu interface should clearly be in its own module, since it will be an executable script, and we may have other executable scripts accessing the notebook in the future. The `Notebook` and `Note` objects can live together in one module. These modules can both exist in the same top-level directory without having to put them in a package. An empty `command_option.py` module can help remind us in the future that we were planning to add new user interfaces:

```
parent_directory/
    notebook.py
    menu.py
    command_option.py
```

Now let's see some code. We start by defining the `Note` class, as it seems simplest. The following example presents `Note` in its entirety. Docstrings within the example explain how it all fits together, demonstrated as follows:

```
import datetime

# Store the next available id for all new notes
last_id = 0


class Note:
    """Represent a note in the notebook. Match against a
    string in searches and store tags for each note."""

    def __init__(self, memo, tags=""):
        """initialize a note with memo and optional
        space-separated tags. Automatically set the note's
        creation date and a unique id."""
        self.memo = memo
        self.tags = tags
        self.creation_date = datetime.date.today()
        global last_id
        last_id += 1
        self.id = last_id

    def match(self, filter):
        """Determine if this note matches the filter
        text. Return True if it matches, False otherwise.

        Search is case sensitive and matches both text and
        tags."""
        return filter in self.memo or filter in self.tags
```

Before continuing, we should quickly fire up the interactive interpreter and test our code so far. Test frequently and often, because things never work the way you expect them to. Indeed, when I tested my first version of this example, I found out I had forgotten the `self` argument in the `match` function! We'll discuss automated testing in `Chapter 12`, *Testing Object-Oriented Programs*. For now, it suffices to check a few things using the interpreter:

```
>>> from notebook import Note
>>> n1 = Note("hello first")
>>> n2 = Note("hello again")
>>> n1.id
1
>>> n2.id
2
>>> n1.match('hello')
```

```
True
>>> n2.match('second')
False
```

It looks like everything is behaving as expected. Let's create our notebook next:

```
class Notebook:
    """Represent a collection of notes that can be tagged,
    modified, and searched."""

    def __init__(self):
        """Initialize a notebook with an empty list."""
        self.notes = []

    def new_note(self, memo, tags=""):
        """Create a new note and add it to the list."""
        self.notes.append(Note(memo, tags))

    def modify_memo(self, note_id, memo):
        """Find the note with the given id and change its
        memo to the given value."""
        for note in self.notes:
            if note.id == note_id:
                note.memo = memo
                break

    def modify_tags(self, note_id, tags):
        """Find the note with the given id and change its
        tags to the given value."""
        for note in self.notes:
            if note.id == note_id:
                note.tags = tags
                break

    def search(self, filter):
        """Find all notes that match the given filter
        string."""
        return [note for note in self.notes if note.match(filter)]
```

We'll clean this up in a minute. First, let's test it to make sure it works:

```
>>> from notebook import Note, Notebook
>>> n = Notebook()
>>> n.new_note("hello world")
>>> n.new_note("hello again")
>>> n.notes
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
0xb73103ac>]
```

```
>>> n.notes[0].id
1
>>> n.notes[1].id
2
>>> n.notes[0].memo
'hello world'
>>> n.search("hello")
[<notebook.Note object at 0xb730a78c>, <notebook.Note object at
0xb73103ac>]
>>> n.search("world")
[<notebook.Note object at 0xb730a78c>]
>>> n.modify_memo(1, "hi world")
>>> n.notes[0].memo
'hi world'
```

It does work. The code is a little messy though; our `modify_tags` and `modify_memo` methods are almost identical. That's not good coding practice. Let's see how we can improve it.

Both methods are trying to identify the note with a given ID before doing something to that note. So, let's add a method to locate the note with a specific ID. We'll prefix the method name with an underscore to suggest that the method is for internal use only, but, of course, our menu interface can access the method if it wants to:

```
def _find_note(self, note_id):
    """Locate the note with the given id."""
    for note in self.notes:
        if note.id == note_id:
            return note
    return None

def modify_memo(self, note_id, memo):
    """Find the note with the given id and change its
    memo to the given value."""
    self._find_note(note_id).memo = memo

def modify_tags(self, note_id, tags):
    """Find the note with the given id and change its
    tags to the given value."""
    self._find_note(note_id).tags = tags
```

This should work for now. Let's have a look at the menu interface. The interface needs to present a menu and allow the user to input choices. Here's our first attempt:

```
import sys
from notebook import Notebook


class Menu:
    """Display a menu and respond to choices when run."""

    def __init__(self):
        self.notebook = Notebook()
        self.choices = {
            "1": self.show_notes,
            "2": self.search_notes,
            "3": self.add_note,
            "4": self.modify_note,
            "5": self.quit,
        }

    def display_menu(self):
        print(
            """
Notebook Menu

1. Show all Notes
2. Search Notes
3. Add Note
4. Modify Note
5. Quit
"""
        )

    def run(self):
        """Display the menu and respond to choices."""
        while True:
            self.display_menu()
            choice = input("Enter an option: ")
            action = self.choices.get(choice)
            if action:
                action()
            else:
                print("{0} is not a valid choice".format(choice))

    def show_notes(self, notes=None):
        if not notes:
            notes = self.notebook.notes
        for note in notes:
```

```
                print("{0}: {1}\n{2}".format(note.id, note.tags, note.memo))

        def search_notes(self):
            filter = input("Search for: ")
            notes = self.notebook.search(filter)
            self.show_notes(notes)

        def add_note(self):
            memo = input("Enter a memo: ")
            self.notebook.new_note(memo)
            print("Your note has been added.")

        def modify_note(self):
            id = input("Enter a note id: ")
            memo = input("Enter a memo: ")
            tags = input("Enter tags: ")
            if memo:
                self.notebook.modify_memo(id, memo)
            if tags:
                self.notebook.modify_tags(id, tags)

        def quit(self):
            print("Thank you for using your notebook today.")
            sys.exit(0)


    if __name__ == "__main__":
        Menu().run()
```

This code first imports the notebook objects using an absolute import. Relative imports wouldn't work because we haven't placed our code inside a package. The `Menu` class's `run` method repeatedly displays a menu and responds to choices by calling functions on the notebook. This is done using an idiom that is rather peculiar to Python; it is a lightweight version of the command pattern that we will discuss in `Chapter 10`, *Python Design Patterns I*. The choices entered by the user are strings. In the menu's `__init__` method, we create a dictionary that maps strings to functions on the menu object itself. Then, when the user makes a choice, we retrieve the object from the dictionary. The `action` variable actually refers to a specific method, and is called by appending empty brackets (since none of the methods require parameters) to the variable. Of course, the user might have entered an inappropriate choice, so we check if the action really exists before calling it.

Each of the various methods request user input and call appropriate methods on the `Notebook` object associated with it. For the `search` implementation, we notice that after we've filtered the notes, we need to show them to the user, so we make the `show_notes` function serve double duty; it accepts an optional `notes` parameter. If it's supplied, it displays only the filtered notes, but if it's not, it displays all notes. Since the `notes` parameter is optional, `show_notes` can still be called with no parameters as an empty menu item.

If we test this code, we'll find that it fails if we try to modify a note. There are two bugs, namely:

- The notebook crashes when we enter a note ID that does not exist. We should never trust our users to enter correct data!
- Even if we enter a correct ID, it will crash because the note IDs are integers, but our menu is passing a string.

The latter bug can be solved by modifying the `Notebook` class's `_find_note` method to compare the values using strings instead of the integers stored in the note, as follows:

```
def _find_note(self, note_id):
    """Locate the note with the given id."""
    for note in self.notes:
        if str(note.id) == str(note_id):
            return note
    return None
```

We simply convert both the input (`note_id`) and the note's ID to strings before comparing them. We could also convert the input to an integer, but then we'd have trouble if the user entered the letter `a` instead of the number `1`.

The problem with users entering note IDs that don't exist can be fixed by changing the two `modify` methods on the notebook to check whether `_find_note` returned a note or not, like this:

```
def modify_memo(self, note_id, memo):
    """Find the note with the given id and change its
    memo to the given value."""
    note = self._find_note(note_id)
    if note:
        note.memo = memo
        return True
    return False
```

This method has been updated to return `True` or `False`, depending on whether a note has been found. The menu could use this return value to display an error if the user entered an invalid note.

This code is a bit unwieldy. It would look a bit better if it raised an exception instead. We'll cover those in `Chapter 4`, *Expecting the Unexpected*.

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.

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. (Hint: it would be similar to the design of the notebook application, but with extra date management methods.) It can keep track of things you want to do each day, and allow you to mark them as completed.

Now try designing a bigger project. As before, it doesn't have to actually do anything, but 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 how to share 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.

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 managing specific details in the 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

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 very little in terms of data and behaviors (the behaviors it does provide are all double-underscore methods intended for internal use only), but it does allow Python to treat all objects in the same way.

If we don't explicitly inherit from a different class, our classes will automatically inherit from `object`. However, we can openly 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*, since Python 3 automatically inherits from `object` if we don't explicitly provide a different **superclass**. A superclass, or parent class, is a class that is being inherited from. A subclass is a class that is inheriting from a superclass. In this case, the superclass is `object`, and `MySubClass` is the subclass. A subclass is also said to be derived from its parent class or that the subclass extends the parent.

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 simple contact manager that tracks the name and email address of several people. The `Contact` class is responsible for maintaining a list of all contacts in a class variable, and for initializing the name and address for an individual contact:

```
class Contact:
    all_contacts = []

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)
```

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 this syntax, for if you ever *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`.

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:

```
class Supplier(Contact):
    def order(self, order):
        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__`, but that only suppliers have a functional 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
>>> c.all_contacts
[<__main__.Contact object at 0xb7375ecc>,
 <__main__.Supplier object at 0xb7375f8c>]
>>> 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 '
```

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

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. We can do this using inheritance:

```
class ContactList(list):
    def search(self, name):
        """Return all contacts that contain the search value
        in their name."""
        matching_contacts = []
        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, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)
```

Instead of instantiating a normal 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@example.net")
>>> c3 = Contact("Jenna C", "jennac@example.net")
>>> [c.name for c in Contact.all_contacts.search('John')]
['John A', 'John B']
```

Are you wondering how we changed the built-in syntax `[]` into something we can inherit from? Creating an empty list with `[]` is actually a shortcut for creating an empty list using `list()`; the two syntaxes behave identically:

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

In reality, the `[]` syntax is actually so-called **syntactic sugar** that calls the `list()` constructor under the hood. The `list` data type is a class that we can extend. In fact, the list itself extends the `object` class:

```
>>> isinstance([], object)
True
```

As a second example, we can extend the `dict` class, which is, similar to the list, the class that is constructed when using the `{}` syntax shorthand:

```
class LongNameDict(dict):
    def longest_key(self):
        longest = None
        for key in self:
            if not longest or len(key) > len(longest):
                longest = key
        return longest
```

This is easy to test in the interactive interpreter:

```
>>> longkeys = LongNameDict()
>>> longkeys['hello'] = 1
>>> longkeys['longest yet'] = 5
>>> longkeys['hello2'] = 'world'
>>> longkeys.longest_key()
'longest yet'
```

Most built-in types can be similarly extended. Commonly extended built-ins are `object`, `list`, `set`, `dict`, `file`, and `str`. Numerical types such as `int` and `float` are also occasionally inherited from.

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 just 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 `__init__`. 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, email, phone):          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.

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 an instance of the parent class, allowing us to call the parent method directly:

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        super().__init__(name, email)
        self.phone = phone
```

This example first gets the instance of the parent object using `super`, and calls `__init__` on that object, passing in the expected arguments. It then does its own initialization, namely, setting the `phone` attribute.

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 is able to access functionality from both of them. In practice, this is less useful than it sounds and many expert programmers recommend against using it.

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 is called a **mixin**. A mixin is a superclass that 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 MailSender:
    def send_mail(self, message):
        print("Sending mail to " + self.email)
        # Add e-mail logic here
```

For brevity, we won'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.

This class doesn't do anything special (in fact, it can barely function as a standalone class), but it 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. We can test this new hybrid to see the mixin at work:

```
>>> e = EmailableContact("John Smith", "jsmith@example.net")
>>> Contact.all_contacts
[<__main__.EmailableContact object at 0xb7205fac>]
>>> e.send_mail("Hello, test e-mail here")
Sending mail to jsmith@example.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 the dire warnings about multiple inheritance are. 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 the subclass. The disadvantage here is that the email functionality then has to be duplicated for any other classes that need an email.
- 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 would be my choice).
- We could have explored 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.
- We could monkey patch (we'll briefly cover monkey patching in `Chapter 7`, *Python Object-Oriented Shortcuts*) 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.

Multiple inheritance works all right when mixing methods from different classes, but it gets very messy when we have to call methods on the superclass. There are multiple superclasses. How do we know which one to call? How do we know what order to call them in?

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 tuple, dictionary, or dataclass (we'll discuss dataclasses in `Chapter 6`, *Python Data Structures*) and pass them into `__init__` as a single argument. This is probably the best course of action if there are no methods that need to be added to the address.

Another option would be to create a new `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.

However, inheritance is also a viable solution, and that's what we want to explore. Let's add a new class that holds 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. Then again, such convoluted naming is a decent indication we should be sticking with composition, rather than inheritance. But for pedagogical purposes, we'll stick with inheritance. Here's our `AddressHolder` class:

```
class AddressHolder:
    def __init__(self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

We just take all the data and toss it into instance variables upon initialization.

The diamond problem

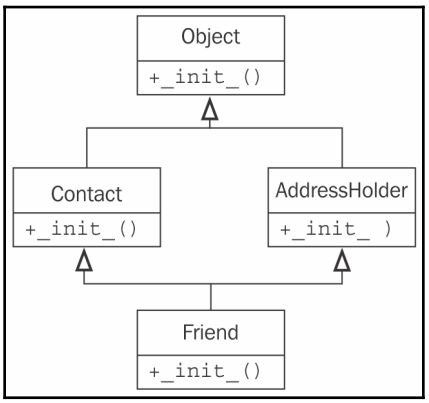
We can use multiple inheritance to add this new class as a parent of our existing `Friend` class. The tricky part is that we now have two parent `__init__` methods, both of which need to be initialized. And they need to be initialized with different arguments. How do we do this? Well, we could start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

```
class Friend(Contact, AddressHolder):
    def __init__(
        self, name, email, phone, street, city, state, code):
        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 go 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. Imagine trying to insert data into a database that has not been connected to, for example.

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

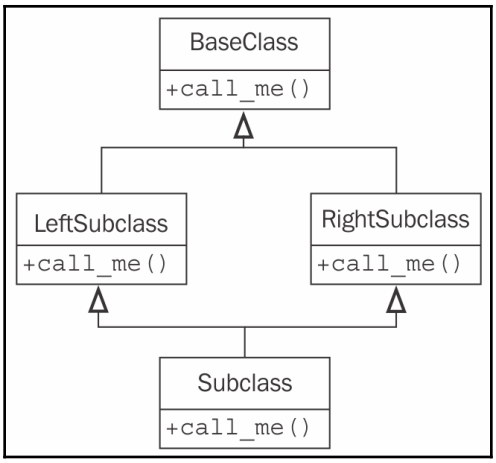


The `__init__` method from the `Friend` class first calls `__init__` on `Contact`, which implicitly initializes the `object` superclass (remember, all classes derive from `object`). `Friend` then calls `__init__` on `AddressHolder`, which implicitly initializes the `object` superclass *again*. This means 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`?

The order in which methods can be called can be adapted on the fly by modifying the `__mro__` (**Method Resolution Order**) attribute on the class. This is beyond the scope of this book. If you think you need to understand it, we recommend *Expert Python Programming*, *Tarek Ziadé*, *Packt Publishing*, or read the original documentation (beware, it's deep!) on the topic at `http://www.python.org/download/releases/2.3/mro/`.

Let's look at a second contrived example, which illustrates this problem more clearly. Here, we have a base class that has a method named `call_me`. Two subclasses override that method, and then another subclass extends both of these using multiple inheritance. This is called diamond inheritance because of the diamond shape of the class diagram:



Let's convert this diagram to code; this example shows when the methods are called:

```
class BaseClass:
    num_base_calls = 0

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


class LeftSubclass(BaseClass):
    num_left_calls = 0

    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Left Subclass")
        self.num_left_calls += 1


class RightSubclass(BaseClass):
    num_right_calls = 0

    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Right Subclass")
        self.num_right_calls += 1


class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0
```

```
    def call_me(self):
        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 updates a static variable on the class to show how many times it has been called. If we instantiate one `Subclass` object and call the method on it once, we get the output:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
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 clearly 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.

The thing to keep in mind with multiple inheritance is that we only want to call the `next` method in the class hierarchy, not the `parent` method. In fact, that next method may not be on a parent or ancestor of the current class. The `super` keyword comes to our rescue once again. Indeed, `super` was originally developed to make complicated forms of multiple inheritance possible. Here is the same code written 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(BaseClass):
    num_left_calls = 0
```

```
    def call_me(self):
        super().call_me()
        print("Calling method on Left Subclass")
        self.num_left_calls += 1


class RightSubclass(BaseClass):
    num_right_calls = 0

    def call_me(self):
        super().call_me()
        print("Calling method on Right Subclass")
        self.num_right_calls += 1


class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

    def call_me(self):
 super().call_me()
        print("Calling method on Subclass")
        self.num_sub_calls += 1
```

The change is pretty minor; we only replaced the naive direct calls with calls to `super()`, although the bottom subclass 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:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
>>> print(s.num_sub_calls, s.num_left_calls, s.num_right_calls,
s.num_base_calls)
1 1 1 1
```

Looks good; our base method is only being called once. But what is `super()` actually doing here? 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.

First, `call_me` of `Subclass` calls `super().call_me()`, which happens to refer to `LeftSubclass.call_me()`. The `LeftSubclass.call_me()` method then calls `super().call_me()`, but in this case, `super()` is referring to `RightSubclass.call_me()`.

Pay particular attention to this: the `super` call is *not* calling the method on the superclass of `LeftSubclass` (which is `BaseClass`). Rather, it is calling `RightSubclass`, even though it is not a direct parent of `LeftSubclass`! This is the *next* method, not the parent method. `RightSubclass` 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` multiple inheritance example. In the `__init__` method for `Friend`, we were originally calling `__init__` for 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 don't necessarily know which class `super` is going to try to initialize first. Even if we did, we need a way to pass the `extra` arguments so that subsequent calls to `super`, on other subclasses, receive the right arguments.

Specifically, if the first call to `super` passes the `name` and `email` arguments to `Contact.__init__`, and `Contact.__init__` then calls `super`, it needs to be able to pass the address-related arguments to the `next` method, which is `AddressHolder.__init__`.

This problem manifests itself anytime we want to call superclass methods with the same name, but with different sets of arguments. Most often, the only time you would want to call a superclass with a completely different set of arguments is in `__init__`, as we're doing here. Even with regular methods, though, we may want to add optional parameters that only make sense to one subclass or set of subclasses.

Sadly, the only way to solve this problem is to plan for it from the beginning. We have to design our base class parameter lists to accept keyword arguments for any parameters that are not required by every subclass implementation. Finally, we must ensure the method freely accepts unexpected arguments and passes them on to its `super` call, in case they are necessary to later methods in the inheritance order.

Python's function parameter syntax provides all the tools we need to do this, but it makes the overall code look cumbersome. Have a look at the proper version of the `Friend` multiple inheritance code, as follows:

```
class Contact:
    all_contacts = []

    def __init__(self, name="", email="", **kwargs):
        super().__init__(**kwargs)
        self.name = name
        self.email = email
        self.all_contacts.append(self)


class AddressHolder:
    def __init__(self, street="", city="", state="", code="", **kwargs):
        super().__init__(**kwargs)
        self.street = street
        self.city = city
        self.state = state
        self.code = code


class Friend(Contact, AddressHolder):
    def __init__(self, phone="", **kwargs):
        super().__init__(**kwargs)
        self.phone = phone
```

We've changed all arguments to keyword arguments by giving them an empty string as a default value. We've also ensured that a `**kwargs` parameter is included to capture any additional parameters that our particular method doesn't know what to do with. It passes these parameters up to the next class with the `super` call.

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 different method (for example, `super().__init__`) with a `**kwargs` syntax, it unpacks the dictionary and passes the results to the method as normal keyword arguments. We'll cover this in detail in `Chapter 7`, *Python Object-Oriented Shortcuts*.

The previous example does what it is supposed to do. But it's starting to look messy, and it is 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 what is happening.

Furthermore, even this implementation is insufficient if we want to *reuse* variables in parent classes. When we pass the `**kwargs` variable to `super`, the dictionary does not include any of the variables that were included as explicit keyword arguments. For example, in `Friend.__init__`, the call to `super` does not have `phone` in the `kwargs` dictionary. If any of the other classes need the `phone` parameter, we need to ensure it is in the dictionary that is passed. Worse, if we forget to do this, it will be extremely frustrating to debug because the superclass will not complain, but will simply assign the default value (in this case, an empty string) to the variable.

There are a few ways to ensure that the variable is passed upward. Assume the `Contact` class does, for some reason, need to be initialized with a `phone` parameter, and the `Friend` class will also need access to it. We can do any of the following:

- Don't include `phone` as an explicit keyword argument. Instead, leave it in the `kwargs` dictionary. `Friend` can look it up using the `kwargs['phone']` syntax. When it passes `**kwargs` to the `super` call, `phone` will still be in the dictionary.
- Make `phone` an explicit keyword argument, but update the `kwargs` dictionary before passing it to `super`, using the standard dictionary `kwargs['phone'] = phone` syntax.
- Make `phone` an explicit keyword argument, but update the `kwargs` dictionary using the `kwargs.update` method. This is useful if you have several arguments to update. You can create the dictionary passed into `update` using either the `dict(phone=phone)` constructor, or the dictionary `{'phone': phone}` syntax.
- Make `phone` an explicit keyword argument, but pass it to the super call explicitly with the `super().__init__(phone=phone, **kwargs)` syntax.

We have covered many of the caveats involved with multiple inheritance in Python. When we need to account for all possible situations, we have to plan for them and our code will get messy. Basic multiple inheritance can be handy but, in many cases, we may want to choose a more transparent way of combining two disparate classes, usually using composition or one of the design patterns we'll be covering in `Chapter 10`, *Design Patterns I*, and `Chapter 11`, *Design Patterns II*.

I have wasted entire days of my life trawling through complex multiple inheritance hierarchies trying to figure out what arguments I need to pass into one of the deeply nested subclasses. The author of the code tended not to document his classes and often passed the kwargs—Just in case they might be needed someday. This was a particularly bad example of using multiple inheritance when it was not needed. Multiple inheritance is a big fancy term that new coders like to show off, but I recommend avoiding it, even when you think it's a good choice. Your future self and other coders will be glad they understand your code when they have to read it later.

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. 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 look:

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename
```

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

    def play(self):
        print("playing {} as mp3".format(self.filename))


class WavFile(AudioFile):
    ext = "wav"

    def play(self):
        print("playing {} as wav".format(self.filename))


class OggFile(AudioFile):
    ext = "ogg"

    def play(self):
        print("playing {} as ogg".format(self.filename))
```

All audio files check to ensure that a valid extension was given upon initialization. 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. If the filename doesn't end with the correct name, it raises an exception (exceptions will be covered in detail in the next chapter). The fact that the `AudioFile` parent class doesn't actually store a reference to the `ext` variable doesn't stop it from being able to access it on the subclass.

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:

```
>>> ogg = OggFile("myfile.ogg")
>>> ogg.play()
playing myfile.ogg as ogg
>>> mp3 = MP3File("myfile.mp3")
>>> mp3.play()
playing myfile.mp3 as mp3
>>> not_an_mp3 = MP3File("myfile.ogg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "polymorphic_audio.py", line 4, in __init__
    raise Exception("Invalid file format")
Exception: Invalid file format
```

See how `AudioFile.__init__` is able to 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, filename):
        if not filename.endswith(".flac"):
            raise Exception("Invalid file format")

        self.filename = filename

    def play(self):
        print("playing {} as flac".format(self.filename))
```

Our media player can play this object just as easily as one that extends `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.

Of course, just because an object satisfies a particular interface (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. For example, our chess AI object from `Chapter 1`, *Object-Oriented Design*, may have a `play()` method that moves a chess piece. Even though it satisfies the interface, this class would likely break in spectacular ways if we tried to plug it into a media player!

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 interface that is actually accessed.

Abstract base classes

While duck typing is useful, it is not always easy to tell in advance if a class is going to fulfill the protocol you require. Therefore, Python introduced the idea of **abstract base classes** (**ABC**s). Abstract base classes define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class. The class can extend the abstract base class itself in order to be used as an instance of that class, but it must supply all the appropriate methods.

In practice, it's rarely necessary to create new abstract base classes, but we may find occasions to implement instances of existing ABCs. We'll cover implementing ABCs first, and then briefly see how to create your own, should you ever need to.

Using an abstract base class

Most of the abstract base classes that exist in the Python standard library live in the `collections` module. One of the simplest ones is the `Container` class. Let's inspect it in the Python interpreter to see what methods this class requires:

```
>>> from collections import Container
>>> Container.__abstractmethods__
frozenset(['__contains__'])
```

So, the `Container` class has exactly one abstract method that needs to be implemented, `__contains__`. You can issue `help(Container.__contains__)` to see what the function signature should look like:

```
Help on method __contains__ in module _abcoll:
 __contains__(self, x) unbound _abcoll.Container method
```

We can see that `__contains__` needs to take a single argument. Unfortunately, the help file doesn't tell us much about what that argument should be, but it's pretty obvious from the name of the ABC and the single method it implements that this argument is the value the user is checking to see whether the container holds.

This method is implemented by `list`, `str`, and `dict` to indicate whether or not a given value is *in* that data structure. However, we can also define a silly container that tells us whether a given value is in the set of odd integers:

```
class OddContainer:
    def __contains__(self, x):
        if not isinstance(x, int) or not x % 2:
            return False
        return True
```

Here's the interesting part: we can instantiate an `OddContainer` object and determine that, even though we did not extend `Container`, the class is a `Container` object:

```
>>> from collections import Container
>>> odd_container = OddContainer()
>>> isinstance(odd_container, Container)
True
>>> issubclass(OddContainer, Container)
True
```

And that is why duck typing is way more awesome than classical polymorphism. We can create is a relationships without the overhead of writing the code to set up inheritance (or worse, multiple inheritance).

One cool thing about the `Container` ABC is that any class that implements it gets to use the `in` keyword for free. In fact, `in` is just syntax sugar that delegates to the `__contains__` method. Any class that has a `__contains__` method is a `Container` and can therefore be queried by the `in` keyword, for example:

```
>>> 1 in odd_container
True
>>> 2 in odd_container
False
>>> 3 in odd_container
True
>>> "a string" in odd_container
False
```

Creating an abstract base class

As we saw earlier, it's not necessary to have an abstract base class to enable duck typing. However, imagine we were creating a media player with third-party plugins. It is advisable to create an abstract base class in this case to document what API the third-party plugins should provide (documentation is one of the stronger use cases for ABCs). The `abc` module provides the tools you need to do this, but I'll warn you in advance, this utilizes some of Python's most arcane concepts, as demonstrated in the following block of code::

```
import abc

class MediaLoader(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def play(self):
        pass

    @abc.abstractproperty
    def ext(self):
        pass

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MediaLoader:
            attrs = set(dir(C))
            if set(cls.__abstractmethods__) <= attrs:
                return True

        return NotImplemented
```

This is a complicated example that includes several Python features that won't be explained until later in this book. It is included here for completeness, but you do not need to understand all of it to get the gist of how to create your own ABC.

The first weird thing is the `metaclass` keyword argument that is passed into the class where you would normally see the list of parent classes. This is a seldom-used construct from the mystic art of metaclass programming. We won't be covering metaclasses in this book, so all you need to know is that by assigning the `ABCMeta` metaclass, you are giving your class superhero (or at least superclass) abilities.

Next, we see the `@abc.abstractmethod` and `@abc.abstractproperty` constructs. These are Python decorators. We'll discuss those in `Chapter 10`, *Python Design Patterns I*. For now, just know that by marking a method or property as being abstract, you are stating that any subclass of this class must implement that method or supply that property in order to be considered a proper member of the class.

See what happens if you implement subclasses that do, or don't, supply those properties:

```
>>> class Wav(MediaLoader):
...     pass
...
>>> x = Wav()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Wav with abstract methods ext,
play
>>> class Ogg(MediaLoader):
...     ext = '.ogg'
...     def play(self):
...         pass
...
>>> o = Ogg()
```

Since the `Wav` class fails to implement the abstract attributes, it is not possible to instantiate that class. The class is still a legal abstract class, but you'd have to subclass it to actually do anything. The `Ogg` class supplies both attributes, so it instantiates cleanly.

Going back to the `MediaLoader` ABC, let's dissect that `__subclasshook__` method. It is basically saying that any class that supplies concrete implementations of all the abstract attributes of this ABC should be considered a subclass of `MediaLoader`, even if it doesn't actually inherit from the `MediaLoader` class.

More common object-oriented languages have a clear separation between the interface and the implementation of a class. For example, some languages provide an explicit `interface` keyword that allows us to define the methods that a class must have without any implementation. In such an environment, an abstract class is one that provides both an interface and a concrete implementation of some, but not all, methods. Any class can explicitly state that it implements a given interface.

Python's ABCs help to supply the functionality of interfaces without compromising on the benefits of duck typing.

Demystifying the magic

You can copy and paste the subclass code without understanding it if you want to make abstract classes that fulfill this particular contract. We'll cover most of the unusual syntaxes in the book, but let's go over it line by line to get an overview:

```
    @classmethod
```

This decorator marks the method as a class method. It essentially says that the method can be called on a class instead of an instantiated object:

```
    def __subclasshook__(cls, C):
```

This defines the `__subclasshook__` class method. This special method is called by the Python interpreter to answer the question: Is the class `C` a subclass of this class?

```
        if cls is MediaLoader:
```

We check to see whether the method was called specifically on this class, rather than, say, a subclass of this class. This prevents, for example, the `Wav` class from being thought of as a parent class of the `Ogg` class:

```
            attrs = set(dir(C))
```

All this line does is get the set of methods and properties that the class has, including any parent classes in its class hierarchy:

```
            if set(cls.__abstractmethods__) <= attrs:
```

This line uses set notation to see whether the set of abstract methods in this class has been supplied in the candidate class. We'll cover sets in detail in the `Chapter 6`, *Python Data Structures.* Note that it doesn't check to see whether the methods have been implemented; just if they are there. Thus, it's possible for a class to be a subclass and yet still be an abstract class itself.

```
                return True
```

If all the abstract methods have been supplied, then the candidate class is a subclass of this class and we return `True`. The method can legally return one of the three values: `True`, `False`, or `NotImplemented`. `True` and `False` indicate that the class is, or isn't, definitively a subclass of this class:

```
    return NotImplemented
```

If any of the conditionals have not been met (that is, the class is not `MediaLoader` or not all abstract methods have been supplied), then return `NotImplemented`. This tells the Python machinery to use the default mechanism (does the candidate class explicitly extend this class?) for subclass detection.

In short, we can now define the `Ogg` class as a subclass of the `MediaLoader` class without actually extending the `MediaLoader` class:

```
>>> class Ogg(): ... ext = '.ogg' ... def play(self): ... print("this will
play an ogg file") ... >>> issubclass(Ogg, MediaLoader) True >>>
isinstance(Ogg(), MediaLoader) True
```

Case study

Let's try to tie everything we've learned together with a larger example. We'll be developing an automated grading system for programming assignments, similar to that employed at Dataquest or Coursera. The system will need to provide a simple class-based interface for course writers to create their assignments and should give a useful error message if it does not fulfill that interface. The writers need to be able to supply their lesson content and to write custom answer checking code to make sure their students got the answer right. It will also be nice for them to have access to the students' names to make the content seem a little friendlier.

The grader itself will need to keep track of which assignment the student is currently working on. A student might make several attempts at an assignment before they get it right. We want to keep track of the number of attempts so the course authors can improve the content of the more difficult lessons.

Let's start by defining the interface that the course authors will need to use. Ideally, it will require the course authors to write a minimal amount of extra code besides their lesson content and answer checking code. Here is the simplest class I could come up with:

```
class IntroToPython:
    def lesson(self):
        return f"""
            Hello {self.student}. define two variables,
            an integer named a with value 1
            and a string named b with value 'hello'

        """

    def check(self, code):
        return code == "a = 1\nb = 'hello'"
```

Admittedly, that particular course author may be a little naive in how they do their answer checking. If you haven't seen the `f"""` syntax before, we'll cover it in detail in the `Chapter 8`, *Strings and Serialization*.

We can start with an abstract base class that defines this interface, as follows:

```
class Assignment(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def lesson(self, student):
        pass

    @abc.abstractmethod
    def check(self, code):
        pass

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Assignment:
            attrs = set(dir(C))
            if set(cls.__abstractmethods__) <= attrs:
                return True

        return NotImplemented
```

This ABC defines the two required abstract methods and provides the magic `__subclasshook__` method to allow a class to be perceived as a subclass without having to explicitly extend it (I usually just copy and paste this code. It isn't worth memorizing.)

We can confirm that the `IntroToPython` class fulfills this interface using `issubclass(IntroToPython, Assignment)`, which should return `True`. Of course, we can explicitly extend the `Assignment` class if we prefer, as seen in this second assignment:

```
class Statistics(Assignment):
    def lesson(self):
        return (
            "Good work so far, "
            + self.student
            + ". Now calculate the average of the numbers "
            + " 1, 5, 18, -3 and assign to a variable named 'avg'"
        )

    def check(self, code):
        import statistics

        code = "import statistics\n" + code

        local_vars = {}
```

```
        global_vars = {}
        exec(code, global_vars, local_vars)

        return local_vars.get("avg") == statistics.mean([1, 5, 18, -3])
```

This course author, unfortunately, is also rather naive. The `exec` call will execute the student's code right inside the grading system, giving them access to the entire system. Obviously, the first thing they will do is hack the system to make their grades 100%. They probably think that's easier than doing the assignments correctly!

Next, we'll create a class that manages how many attempts the student has made at a given assignment:

```
class AssignmentGrader:
    def __init__(self, student, AssignmentClass):
        self.assignment = AssignmentClass()
        self.assignment.student = student
        self.attempts = 0
        self.correct_attempts = 0

    def check(self, code):
        self.attempts += 1
        result = self.assignment.check(code)
        if result:
            self.correct_attempts += 1

        return result

    def lesson(self):
        return self.assignment.lesson()
```

This class uses composition instead of inheritance. At first glance, it would make sense for these methods to exist on the `Assignment` superclass. That would eliminate the annoying `lesson` method, which just proxies through to the same method on the assignment object. It would certainly be possible to put all this logic directly on the `Assignment` abstract base class, or even to have the ABC inherit from this `AssignmentGrader` class. In fact, I would normally recommend that, but in this case, it would force all course authors to explicitly extend the class, which violates our request that content authoring be as simple as possible.

Finally, we can start to put together the `Grader` class, which is responsible for managing which assignments are available and which one each student is currently working on. The most interesting part is the register method:

```
import uuid

class Grader:
    def __init__(self):
        self.student_graders = {}
        self.assignment_classes = {}

    def register(self, assignment_class):
        if not issubclass(assignment_class, Assignment):
            raise RuntimeError(
                "Your class does not have the right methods"
            )

        id = uuid.uuid4()
        self.assignment_classes[id] = assignment_class
        return id
```

This code block includes the initializer, which includes two dictionaries we'll discuss in a minute. The `register` method is a bit complex, so we'll dissect it thoroughly.

The first odd thing is the parameter this method accepts: `assignment_class`. This parameter is intended to be an actual class, not an instance of the class. Remember, classes are objects, too, and can be passed around like other classes. Given the `IntroToPython` class we defined earlier, we might register it without instantiating it, as follows:

```
from grader import Grader
from lessons import IntroToPython, Statistics

grader = Grader()
itp_id = grader.register(IntroToPython)
```

The method first checks whether that class is a subclass of the `Assignment` class. Of course, we implemented a custom `__subclasshook__` method, so this includes classes that do not explicitly subclass `Assignment`. The naming is, perhaps, a bit deceitful! If it doesn't have the two required methods, it raises an exception. Exceptions are a topic we'll cover in detail in the next chapter; for now, just assume that it makes the program get angry and quit.

Then, we generate a random identifier to represent that specific assignment. We store the `assignment_class` in a dictionary indexed by that ID, and return the ID so that the calling code can look that assignment up in the future. Presumably, another object would then place that ID in a course syllabus of some sort so students do the assignments in order, but we won't be doing that for this part of the project.

The `uuid` function returns a specially formatted string called a universally unique identifier, also known as a globally unique identifier. It essentially represents an extremely large random number that is almost, but not quite, impossible to conflict with another similarly generated identifier. It is a great, quick, and clean way to create an arbitrary ID to keep track of items.

Next up, we have the `start_assignment` function, which allows a student to start working on an assignment given the ID of that assignment. All it does is construct an instance of the `AssignmentGrader` class we defined earlier and plop it in a dictionary stored on the `Grader` class, as follows:

```
def start_assignment(self, student, id):
    self.student_graders[student] = AssignmentGrader(
        student, self.assignment_classes[id]
    )
```

After that, we write a couple of proxy methods that get the lesson or check the code for whatever assignment the student is currently working on:

```
def get_lesson(self, student):
    assignment = self.student_graders[student]
    return assignment.lesson()

def check_assignment(self, student, code):
    assignment = self.student_graders[student]
    return assignment.check(code)
```

Finally, we create a method that gives a summary of a student's current assignment progress. It looks up the assignment object and creates a formatted string with all the information we have about that student:

```
def assignment_summary(self, student):
    grader = self.student_graders[student]
    return f"""
    {student}'s attempts at {grader.assignment.__class__.__name__}:

    attempts: {grader.attempts}
    correct: {grader.correct_attempts}
```

```
            passed: {grader.correct_attempts > 0}
            """
```

And that's it. You'll notice that this case study does not use a ton of inheritance, which may seem a bit odd given the topic of the chapter, but duck typing is very prevalent. It is quite common for Python programs to be designed with inheritance that gets simplified into more versatile constructs as it is iterated on. As another example, I originally defined the `AssignmentGrader` as an inheritance relationship, but realized halfway through that it would be better to use composition, for the reasons outlined previously.

Here's a bit of test code that shows all these objects connected together:

```
grader = Grader()
itp_id = grader.register(IntroToPython)
stat_id = grader.register(Statistics)

grader.start_assignment("Tammy", itp_id)
print("Tammy's Lesson:", grader.get_lesson("Tammy"))
print(
    "Tammy's check:",
    grader.check_assignment("Tammy", "a = 1 ; b = 'hello'"),
)
print(
    "Tammy's other check:",
    grader.check_assignment("Tammy", "a = 1\nb = 'hello'"),
)

print(grader.assignment_summary("Tammy"))

grader.start_assignment("Tammy", stat_id)
print("Tammy's Lesson:", grader.get_lesson("Tammy"))
print("Tammy's check:", grader.check_assignment("Tammy", "avg=5.25"))
print(
    "Tammy's other check:",
    grader.check_assignment(
        "Tammy", "avg = statistics.mean([1, 5, 18, -3])"
    ),
)

print(grader.assignment_summary("Tammy"))
```

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 student grader system in the case study. There is a lot missing from it, and not just decent course content! How do students get into the system? Is there a curriculum that defines which order they should study lessons in? What happens if you change the `AssignmentGrader` to use inheritance, rather than composition, on the `Assignment` objects?

Finally, try to come up with some good use cases for mixins, then experiment with them until you realize that there is probably a better design using composition!

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-ins using inheritance. 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. Abstract base classes allow you to document what methods and properties a class must have to fulfill a particular interface, and even allow you to change the very definition of *subclass*.

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

4
Expecting the Unexpected

Programs are very fragile. It would be ideal if code always returned a valid result, but sometimes a valid result can't be calculated. For example, it's not possible to divide by zero, or to access the eighth item in a five-item list.

In the old days, the only way around this was to rigorously check the inputs for every function to make sure they made sense. Typically, functions had special return values to indicate an error condition; for example, they could return a negative number to indicate that a positive value couldn't be calculated. Different numbers might mean different errors occurred. Any code that called this function would have to explicitly check for an error condition and act accordingly. A lot of developers didn't bother to do this, and programs simply crashed. However, in the object-oriented world, this is not the case.

In this chapter, we will study **exceptions**, special error objects that only need to be handled when it makes sense to handle them. 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

Raising exceptions

In principle, an exception is just 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`. These exception objects become special when they are handled inside the program's flow of control. When an exception occurs, everything that was supposed to happen doesn't happen, unless it was supposed to happen when an exception occurred. 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 "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

This `print` statement was a valid command way back in the Python 2 and earlier days, but in Python 3, because `print` is a function, we have to enclose the arguments in parentheses. So, if we type the preceding command into a Python 3 interpreter, we get `SyntaxError`.

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: int division or modulo 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
```

Sometimes, these exceptions are indicators of something wrong in our program (in which case, we would go to the indicated line number and fix it), but they also occur in legitimate situations. A `ZeroDivisionError` error doesn't always mean we received an invalid input. It could also mean we have received a different input. The user may have entered a zero by mistake, or on purpose, or it may represent a legitimate value, such as an empty bank account or the age of a newborn child.

You may have noticed all the preceding 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:

```
class EvenOnly(list):
    def append(self, integer):
        if not isinstance(integer, int):
            raise TypeError("Only integers can be added")
        if integer % 2:
            raise ValueError("Only even numbers can be added")
        super().append(integer)
```

This class extends the `list` built-in, as we discussed in `Chapter 2`, *Objects in Python*, and overrides 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 modulus 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)
```

While this class is effective for demonstrating exceptions in action, it isn't very good at its job. It is still possible to get other values into the list using index notation or slice notation. This can all be avoided by overriding other appropriate methods, some of which are magic double-underscore methods.

The effects of an exception

When an exception is raised, it appears to stop program execution immediately. Any lines that were supposed to run after the exception is raised are not executed, and unless the exception is dealt with, the program will exit with an error message. Take a look at this basic function:

```
def no_return():
    print("I am about to raise an exception")
    raise Exception("This is always raised")
    print("This line will never execute")
    return "I won't be returned"
```

If we execute this function, we see that the first `print` call is executed and then the exception is raised. The second `print` function call is never executed, nor is the `return` statement:

```
>>> no_return()
I am about to raise an exception
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "exception_quits.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

Furthermore, if we have a function that calls another function that raises an exception, nothing is executed in the first function after the point where the second function was called. Raising an exception stops all execution right up through the function call stack until it is either handled or forces the interpreter to exit. To demonstrate, let's add a second function that calls the earlier one:

```
def call_exceptor():
    print("call_exceptor starts here...")
    no_return()
    print("an exception was raised...")
    print("...so these lines don't run")
```

When we call this function, we see that the first `print` statement executes, as well as the first line in the `no_return` function. But once the exception is raised, nothing else executes:

```
>>> call_exceptor()
call_exceptor starts here...
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "method_calls_excepting.py", line 9, in call_exceptor
    no_return()
  File "method_calls_excepting.py", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

We'll soon see that when the interpreter is not actually taking a shortcut and exiting immediately, we can react to and deal with the exception inside either method. Indeed, exceptions can be handled at any level after they are initially raised.

Look at the exception's output (called a traceback) from bottom to top, and notice how both methods are listed. Inside `no_return`, the exception is initially raised. Then, just above that, we see that inside `call_exceptor`, that pesky `no_return` function was called and the exception *bubbled up* to the calling method. From there, it went up one more level to the main interpreter, which, not knowing what else to do with it, gave up and printed a traceback.

Handling exceptions

Now let's look at the tail side of the exception coin. If we encounter an exception situation, how should our code react to or recover from it? We handle exceptions by wrapping any code that might throw one (whether it is exception code itself, or a call to any function or method that may have an exception raised inside it) inside a `try...except` clause. The most basic syntax looks like this:

```
try:
    no_return()
except:
    print("I caught an exception")
print("executed after the exception")
```

If we run this simple script using our existing `no_return` function—which, as we know very well, always throws an exception—we get this output:

```
I am about to raise an exception
I caught an exception
executed after the exception
```

The `no_return` function happily informs us that it is about to raise an exception, but we fooled it and caught the exception. Once caught, we were able to clean up after ourselves (in this case, by outputting that we were handling the situation), and continue on our way, with no interference from that offensive function. The remainder of the code in the `no_return` function still went unexecuted, but the code that called the function was able to recover and continue.

Note the indentation around `try` and `except`. The `try` clause wraps any code that might throw an exception. The `except` clause is then back on the same indentation level as the `try` line. Any code to handle the exception is indented after the `except` clause. Then normal code resumes at the original indentation level.

The problem with the preceding code is that it will catch any type of exception. What if we were writing some code that could raise both `TypeError` and `ZeroDivisionError`? We might want to catch `ZeroDivisionError`, but let `TypeError` propagate to the console. Can you guess the syntax?

Here's a rather silly function that does just that:

```
def funny_division(divider):
    try:
        return 100 / divider
    except ZeroDivisionError:
        return "Zero is not a good idea!"


print(funny_division(0))
print(funny_division(50.0))
print(funny_division("hello"))
```

The function is tested with the `print` statements that show it behaving as expected:

```
Zero is not a good idea!
2.0
Traceback (most recent call last):
  File "catch_specific_exception.py", line 9, in <module>
    print(funny_division("hello"))
  File "catch_specific_exception.py", line 3, in funny_division
    return 100 / divider
TypeError: unsupported operand type(s) for /: 'int' and 'str'.
```

The first line of output shows that if we enter `0`, we get properly mocked. If we call with a valid number (note that it's not an integer, but it's still a valid divisor), it operates correctly. Yet if we enter a string (you were wondering how to get a `TypeError`, weren't you?), it fails with an exception. If we had used an empty `except` clause that didn't specify a `ZeroDivisionError`, it would have accused us of dividing by zero when we sent it a string, which is not a proper behavior at all.

The *bare except* syntax is generally frowned upon, even if you really do want to catch all instances of an exception. Use the `except Exception:` syntax to explicitly catch all exception types. This tell the reader that you meant to catch exception objects and all subclasses of `Exception`. The bare except syntax is actually the same as using `except BaseException:`, which actually catches system-level exceptions that are very rare to intentionally want to catch, as we'll see in the next section. If you really do want to catch them, explicitly use `except BaseException:` so that anyone who reads your code knows that you didn't just forget to specify what kind of exception you wanted.

We can even catch two or more different exceptions and handle them with the same code. Here's an example that raises three different types of exception. It handles `TypeError` and `ZeroDivisionError` with the same exception handler, but it may also raise a `ValueError` error if you supply the number `13`:

```
def funny_division2(divider):
    try:
        if divider == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / divider
    except (ZeroDivisionError, TypeError):
        return "Enter a number other than zero"


for val in (0, "hello", 50.0, 13):

    print("Testing {}:".format(val), end=" ")
    print(funny_division2(val))
```

The `for` loop at the bottom loops over several test inputs and prints the results. If you're wondering about that `end` argument in the `print` statement, it just turns the default trailing newline into a space so that it's joined with the output from the next line. Here's a run of the program:

```
Testing 0: Enter a number other than zero
Testing hello: Enter a number other than zero
Testing 50.0: 2.0
Testing 13: Traceback (most recent call last):
  File "catch_multiple_exceptions.py", line 11, in <module>
    print(funny_division2(val))
  File "catch_multiple_exceptions.py", line 4, in funny_division2
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

The number `0` and the string are both caught by the `except` clause, and a suitable error message is printed. The exception from the number `13` is not caught because it is a `ValueError`, which was not included in the types of exceptions being handled. This is all well and good, but what if we want to catch different exceptions and do different things with them? Or maybe we want to do something with an exception and then allow it to continue to bubble up to the parent function, as if it had never been caught?

We don't need any new syntax to deal with these cases. It's possible to stack the `except` clauses, and only the first match will be executed. For the second question, the `raise` keyword, with no arguments, will re-raise the last exception if we're already inside an exception handler. Observe the following code:

```
def funny_division3(divider):
    try:
        if divider == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / divider
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise
```

The last line re-raises the `ValueError` error, so after outputting `No, No, not 13!`, it will raise the exception again; we'll still get the original stack trace on the console.

If we stack exception clauses like we did in the preceding example, only the first matching clause will be run, even if more than one of them fits. How can more than one clause match? Remember that exceptions are objects, and can therefore be subclassed. As we'll see in the next section, most exceptions extend the `Exception` class (which is itself derived from `BaseException`). If we catch `Exception` before we catch `TypeError`, then only the `Exception` handler will be executed, because `TypeError` is an `Exception` by inheritance.

This can come in handy in cases where we want to handle some exceptions specifically, and then handle all remaining exceptions as a more general case. We can simply catch `Exception` after catching all the specific exceptions and handle the general case there.

Often, when we catch an exception, we need a reference to the `Exception` object itself. This most often happens when we define our own exceptions with custom arguments, but can also be relevant with standard exceptions. Most exception classes accept a set of arguments in their constructor, and we might want to access those attributes in the exception handler. If we define our own `Exception` class, we can even call custom methods on it when we catch it. The syntax for capturing an exception as a variable uses the `as` keyword:

```
try:
    raise ValueError("This is an argument")
except ValueError as e:
    print("The exception arguments were", e.args)
```

If we run this simple snippet, it prints out the string argument that we passed into `ValueError` upon initialization.

We've seen several variations on the syntax for handling exceptions, but we still don't know how to execute code regardless of whether or not an exception has occurred. We also can't specify code that should be executed **only** if an exception does **not** occur. Two more keywords, `finally` and `else`, can provide the missing pieces. Neither one takes any extra arguments. The following example randomly picks an exception to throw and raises it. Then some not-so-complicated exception handling code runs that illustrates the newly introduced syntax:

```
import random
some_exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = random.choice(some_exceptions)
    print("raising {}".format(choice))
    if choice:
        raise choice("An error")
except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" %
        ( e.__class__.__name__))
else:
    print("This code called if there is no exception")
finally:
    print("This cleanup code is always called")
```

If we run this example—which illustrates almost every conceivable exception handling scenario—a few times, we'll get different output each time, depending on which exception `random` chooses. Here are some example runs:

```
$ python finally_and_else.py
raising None
This code called if there is no exception
This cleanup code is always called

$ python finally_and_else.py
raising <class 'TypeError'>
Caught a TypeError
This cleanup code is always called
$ python finally_and_else.py
raising <class 'IndexError'>
```

```
Caught some other error: IndexError
This cleanup code is always called

$ python finally_and_else.py
raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called
```

Note how the `print` statement in the `finally` clause is executed no matter what happens. This is extremely useful when we need to perform certain tasks after our code has finished running (even if an exception has occurred). Some common examples include the following:

- Cleaning up an open database connection
- Closing an open file
- Sending a closing handshake over the network

The `finally` clause is also very important when we execute a `return` statement from inside a `try` clause. The `finally` handler will still be executed before the value is returned without executing any code following the `try...finally` clause.

Also, pay attention to the output when no exception is raised: both the `else` and the `finally` clauses are executed. The `else` clause may seem redundant, as the code that should be executed only when no exception is raised could just be placed after the entire `try...except` block. The difference is that the `else` block will not be executed if an exception is caught and handled. We'll see more on this when we discuss using exceptions as flow control later.

Any of the `except`, `else`, and `finally` clauses can be omitted after a `try` block (although `else` by itself is invalid). If you include more than one, the `except` clauses must come first, then the `else` clause, with the `finally` clause at the end. The order of the `except` clauses normally goes from most specific to most generic.

The exception hierarchy

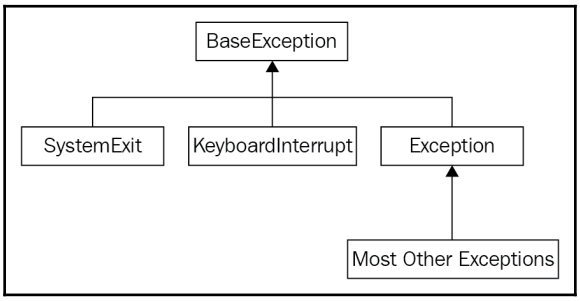
We've already seen several of the most common built-in exceptions, and you'll probably encounter the rest over the course of your regular Python development. As we noticed earlier, most exceptions are subclasses of the `Exception` class. But this is not true of all exceptions. `Exception` itself actually inherits from a class called `BaseException`. In fact, all exceptions must extend the `BaseException` class or one of its subclasses.

There are two key built-in the exception classes, `SystemExit` and `KeyboardInterrupt`, that derive directly from `BaseException` instead of `Exception`. The `SystemExit` exception is raised whenever the program exits naturally, typically because we called the `sys.exit` function somewhere in our code (for example, when the user selected an exit menu item, clicked the *Close* button on a window, or entered a command to shut down a server). The exception is designed to allow us to clean up code before the program ultimately exits. However, we generally don't need to handle it explicitly because cleanup code can happen inside a `finally` clause.

If we do handle it, we would normally re-raise the exception, since catching it would stop the program from exiting. There are, of course, situations where we might want to stop the program exiting; for example, if there are unsaved changes and we want to prompt the user if they really want to exit. Usually, if we handle `SystemExit` at all, it's because we want to do something special with it, or are anticipating it directly. We especially don't want it to be accidentally caught in generic clauses that catch all normal exceptions. This is why it derives directly from `BaseException`.

The `KeyboardInterrupt` exception is common in command-line programs. It is thrown when the user explicitly interrupts program execution with an OS-dependent key combination (normally, *Ctrl* + *C*). This is a standard way for the user to deliberately interrupt a running program, and like `SystemExit`, it should almost always respond by terminating the program. Also, like `SystemExit`, it should handle any cleanup tasks inside the `finally` blocks.

Here is a class diagram that fully illustrates the hierarchy:



When we use the `except:` clause without specifying any type of exception, it will catch all subclasses of `BaseException`; which is to say, it will catch all exceptions, including the two special ones. Since we almost always want these to get special treatment, it is unwise to use the `except:` statement without arguments. If you want to catch all exceptions other than `SystemExit` and `KeyboardInterrupt`, explicitly catch `Exception`. Most Python developers assume that `except:` without a type is an error and will flag it in code review. If you really do want to catch everything, just explicitly use `except BaseException:`.

Defining our own exceptions

Occasionally, when we want to raise an exception, we find that none of the built-in exceptions are suitable. Luckily, it's trivial to define new exceptions of our own. The name of the class is usually designed to communicate what went wrong, and we can provide arbitrary arguments in the initializer to include additional information.

All we have to do is inherit from the `Exception` class. We don't even have to add any content to the class! We can, of course, extend `BaseException` directly, but I have never encountered a use case where this would make sense.

Here's a simple exception we might use in a banking application:

```
class InvalidWithdrawal(Exception):
    pass

raise InvalidWithdrawal("You don't have $50 in your account")
```

The last line illustrates how to raise the newly defined exception. We are able to pass an arbitrary number of arguments into the exception. Often a string message is used, but any object that might be useful in a later exception handler can be stored. The `Exception.__init__` method is designed to accept any arguments and store them as a tuple in an attribute named `args`. This makes exceptions easier to define without needing to override `__init__`.

Of course, if we do want to customize the initializer, we are free to do so. Here's an exception whose initializer accepts the current balance and the amount the user wanted to withdraw. In addition, it adds a method to calculate how overdrawn the request was:

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__(f"account doesn't have ${amount}")
        self.amount = amount
        self.balance = balance
```

```
    def overage(self):
        return self.amount - self.balance

raise InvalidWithdrawal(25, 50)
```

The `raise` statement at the end illustrates how to construct this exception. As you can see, we can do anything with an exception that we would do with other objects.

Here's how we would handle an `InvalidWithdrawal` exception if one was raised:

```
try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("I'm sorry, but your withdrawal is "
            "more than your balance by "
            f"${e.overage()}")
```

Here we see a valid use of the `as` keyword. By convention, most Python coders name the exception `e` or the `ex` variable, although, as usual, you are free to call it `exception`, or `aunt_sally` if you prefer.

There are many reasons for defining our own exceptions. It is often useful to add information to the exception or log it in some way. But the utility of custom exceptions truly comes to light when creating a framework, library, or API that is intended for access by other programmers. In that case, be careful to ensure your code is raising exceptions that make sense to the client programmer. They should be easy to handle and clearly describe what went on. The client programmer should easily see how to fix the error (if it reflects a bug in their code) or handle the exception (if it's a situation they need to be made aware of).

Exceptions aren't exceptional. Novice programmers tend to think of exceptions as only useful for exceptional circumstances. However, the definition of exceptional circumstances can be vague and subject to interpretation. Consider the following two functions:

```
def divide_with_exception(number, divisor):
    try:
        print(f"{number} / {divisor} = {number / divisor}")
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print(f"{number} / {divisor} = {number / divisor}")
```

These two functions behave identically. If `divisor` is zero, an error message is printed; otherwise, a message printing the result of division is displayed. We could avoid `ZeroDivisionError` ever being thrown by testing for it with an `if` statement. Similarly, we can avoid `IndexError` by explicitly checking whether or not the parameter is within the confines of the list, and `KeyError` by checking whether the key is in a dictionary.

But we shouldn't do this. For one thing, we might write an `if` statement that checks whether or not the index is lower than the parameters of the list, but forget to check negative values.

Remember, Python lists support negative indexing; `-1` refers to the last element in the list.

Eventually, we would discover this and have to find all the places where we were checking code. But if we had simply caught `IndexError` and handled it, our code would just work.

Python programmers tend to follow a model of *ask forgiveness rather than permission*, which is to say, they execute code and then deal with anything that goes wrong. The alternative, to *look before you leap*, is generally less popular. There are a few reasons for this, but the main one is that it shouldn't be necessary to burn CPU cycles looking for an unusual situation that is not going to arise in the normal path through the code. Therefore, it is wise to use exceptions for exceptional circumstances, even if those circumstances are only a little bit exceptional. Taking this argument further, we can actually see that the exception syntax is also effective for flow control. Like an `if` statement, exceptions can be used for decision making, branching, and message passing.

Imagine an inventory application for a company that sells widgets and gadgets. When a customer makes a purchase, the item can either be available, in which case the item is removed from inventory and the number of items left is returned, or it might be out of stock. Now, being out of stock is a perfectly normal thing to happen in an inventory application. It is certainly not an exceptional circumstance. But what do we return if it's out of stock? A string saying out of stock? A negative number? In both cases, the calling method would have to check whether the return value is a positive integer or something else, to determine if it is out of stock. That seems a bit messy, especially if we forget to do it somewhere in our code.

Instead, we can raise `OutOfStock` and use the `try` statement to direct program flow control. Make sense? In addition, we want to make sure we don't sell the same item to two different customers, or sell an item that isn't in stock yet. One way to facilitate this is to lock each type of item to ensure only one person can update it at a time. The user must lock the item, manipulate the item (purchase, add stock, count items left...), and then unlock the item. Here's an incomplete `Inventory` example with docstrings that describes what some of the methods should do:

```
class Inventory:
    def lock(self, item_type):
        """Select the type of item that is going to
        be manipulated. This method will lock the
        item so nobody else can manipulate the
        inventory until it's returned. This prevents
        selling the same item to two different
        customers."""
        pass

    def unlock(self, item_type):
        """Release the given type so that other
        customers can access it."""
        pass

    def purchase(self, item_type):
        """If the item is not locked, raise an
        exception. If the item_type does not exist,
        raise an exception. If the item is currently
        out of stock, raise an exception. If the item
        is available, subtract one item and return
        the number of items left."""
        pass
```

We could hand this object prototype to a developer and have them implement the methods to do exactly as they say while we work on the code that needs to make a purchase. We'll use Python's robust exception handling to consider different branches, depending on how the purchase was made:

```
item_type = "widget"
inv = Inventory()
inv.lock(item_type)
try:
    num_left = inv.purchase(item_type)
except InvalidItemType:
    print("Sorry, we don't sell {}".format(item_type))
except OutOfStock:
    print("Sorry, that item is out of stock.")
else:
```

```
        print("Purchase complete. There are {num_left} {item_type}s left")
    finally:
        inv.unlock(item_type)
```

Pay attention to how all the possible exception handling clauses are used to ensure the correct actions happen at the correct time. Even though `OutOfStock` is not a terribly exceptional circumstance, we are able to use an exception to handle it suitably. This same code could be written with an `if...elif...else` structure, but it wouldn't be as easy to read or maintain.

We can also use exceptions to pass messages between different methods. For example, if we wanted to inform the customer as to what date the item is expected to be in stock again, we could ensure our `OutOfStock` object requires a `back_in_stock` parameter when it is constructed. Then, when we handle the exception, we can check that value and provide additional information to the customer. The information attached to the object can be easily passed between two different parts of the program. The exception could even provide a method that instructs the inventory object to reorder or backorder an item.

Using exceptions for flow control can make for some handy program designs. The important thing to take from this discussion is that exceptions are not a bad thing that we should try to avoid. Having an exception occur does not mean that you should have prevented this exceptional circumstance from happening. Rather, it is just a powerful way to communicate information between two sections of code that may not be directly calling each other.

Case study

We've been looking at the use and handling of exceptions at a fairly low level of detail—syntax and definitions. This case study will help tie it all in with our previous chapters so we can see how exceptions are used in the larger context of objects, inheritance, and modules.

Today, we'll be designing a simple central authentication and authorization system. The entire system will be placed in one module, and other code will be able to query that module object for authentication and authorization purposes. We should admit, from the start, that we aren't security experts, and that the system we are designing may be full of security holes. Our purpose is to study exceptions, not to secure a system. It will be sufficient, however, for a basic login and permission system that other code can interact with. Later, if that other code needs to be made more secure, we can have a security or cryptography expert review or rewrite our module, preferably without changing the API.

Authentication is the process of ensuring a user is really the person they say they are. We'll follow the lead of common web systems today, which use a username and private password combination. Other methods of authentication include voice recognition, fingerprint or retinal scanners, and identification cards.

Authorization, on the other hand, is all about determining whether a given (authenticated) user is permitted to perform a specific action. We'll create a basic permission list system that stores a list of the specific people allowed to perform each action.

In addition, we'll add some administrative features to allow new users to be added to the system. For brevity, we'll leave out editing of passwords or changing of permissions once they've been added, but these (highly necessary) features can certainly be added in the future.

There's a simple analysis; now let's proceed with design. We're obviously going to need a `User` class that stores the username and an encrypted password. This class will also allow a user to log in by checking whether a supplied password is valid. We probably won't need a `Permission` class, as those can just be strings mapped to a list of users using a dictionary. We should have a central `Authenticator` class that handles user management and logging in or out. The last piece of the puzzle is an `Authorizor` class that deals with permissions and checking whether a user can perform an activity. We'll provide a single instance of each of these classes in the `auth` module so that other modules can use this central mechanism for all their authentication and authorization needs. Of course, if they want to instantiate private instances of these classes, for non-central authorization activities, they are free to do so.

We'll also be defining several exceptions as we go along. We'll start with a special `AuthException` base class that accepts a `username` and optional `user` object as parameters; most of our self-defined exceptions will inherit from this one.

Let's build the `User` class first; it seems simple enough. A new user can be initialized with a username and password. The password will be stored encrypted to reduce the chances of its being stolen. We'll also need a `check_password` method to test whether a supplied password is the correct one. Here is the class in full:

```
import hashlib


class User:
    def __init__(self, username, password):
        """Create a new user object. The password
        will be encrypted before storing."""
        self.username = username
        self.password = self._encrypt_pw(password)
```

```
        self.is_logged_in = False

    def _encrypt_pw(self, password):
        """Encrypt the password with the username and return
        the sha digest."""
        hash_string = self.username + password
        hash_string = hash_string.encode("utf8")
        return hashlib.sha256(hash_string).hexdigest()

    def check_password(self, password):
        """Return True if the password is valid for this
        user, false otherwise."""
        encrypted = self._encrypt_pw(password)
        return encrypted == self.password
```

Since the code for encrypting a password is required in both `__init__` and `check_password`, we pull it out to its own method. This way, it only needs to be changed in one place if someone realizes it is insecure and needs improvement. This class could easily be extended to include mandatory or optional personal details, such as names, contact information, and birth dates.

Before we write code to add users (which will happen in the as-yet undefined `Authenticator` class), we should examine some use cases. If all goes well, we can add a user with a username and password; the `User` object is created and inserted into a dictionary. But in what ways can all not go well? Well, clearly we don't want to add a user with a username that already exists in the dictionary. If we did so, we'd overwrite an existing user's data and the new user might have access to that user's privileges. So, we'll need a `UsernameAlreadyExists` exception. Also, for security's sake, we should probably raise an exception if the password is too short. Both of these exceptions will extend `AuthException`, which we mentioned earlier. So, before writing the `Authenticator` class, let's define these three exception classes:

```
class AuthException(Exception):
    def __init__(self, username, user=None):
        super().__init__(username, user)
        self.username = username
        self.user = user

class UsernameAlreadyExists(AuthException):
    pass

class PasswordTooShort(AuthException):
    pass
```

The `AuthException` requires a username and has an optional user parameter. This second parameter should be an instance of the `User` class associated with that username. The two specific exceptions we're defining simply need to inform the calling class of an exceptional circumstance, so we don't need to add any extra methods to them.

Now let's start on the `Authenticator` class. It can simply be a mapping of usernames to user objects, so we'll start with a dictionary in the initialization function. The method for adding a user needs to check the two conditions (password length and previously existing users) before creating a new `User` instance and adding it to the dictionary:

```
class Authenticator:
    def __init__(self):
        """Construct an authenticator to manage
        users logging in and out."""
        self.users = {}

    def add_user(self, username, password):
        if username in self.users:
            raise UsernameAlreadyExists(username)
        if len(password) < 6:
            raise PasswordTooShort(username)
        self.users[username] = User(username, password)
```

We could, of course, extend the password validation to raise exceptions for passwords that are too easy to crack in other ways, if we desired. Now let's prepare the `login` method. If we weren't thinking about exceptions just now, we might just want the method to return `True` or `False`, depending on whether the login was successful or not. But we are thinking about exceptions, and this could be a good place to use them for a not-so-exceptional circumstance. We could raise different exceptions, for example, if the username does not exist or the password does not match. This will allow anyone trying to log a user in to elegantly handle the situation using a `try/except/else` clause. So, first we add these new exceptions:

```
class InvalidUsername(AuthException):
    pass

class InvalidPassword(AuthException):
    pass
```

Then we can define a simple `login` method to our `Authenticator` class that raises these exceptions if necessary. If not, it flags the `user` as logged in and returns the following:

```
    def login(self, username, password):
        try:
            user = self.users[username]
        except KeyError:
            raise InvalidUsername(username)

        if not user.check_password(password):
            raise InvalidPassword(username, user)

        user.is_logged_in = True
        return True
```

Notice how `KeyError` is handled. This could have been handled using `if username not in self.users:` instead, but we chose to handle the exception directly. We end up eating up this first exception and raising a brand new one of our own that better suits the user-facing API.

We can also add a method to check whether a particular username is logged in. Deciding whether to use an exception here is trickier. Should we raise an exception if the username does not exist? Should we raise an exception if the user is not logged in?

To answer these questions, we need to think about how the method would be accessed. Most often, this method will be used to answer the yes/no question, *should I allow them access to <something>?* The answer will either be, *yes, the username is valid and they are logged in*, or *no, the username is not valid or they are not logged in*. Therefore, a Boolean return value is sufficient. There is no need to use exceptions here, just for the sake of using an exception:

```
    def is_logged_in(self, username):
        if username in self.users:
            return self.users[username].is_logged_in
        return False
```

Finally, we can add a default authenticator instance to our module so that the client code can access it easily using `auth.authenticator`:

```
authenticator = Authenticator()
```

This line goes at the module level, outside any class definition, so the `authenticator` variable can be accessed as `auth.authenticator`. Now we can start on the `Authorizor` class, which maps permissions to users. The `Authorizor` class should not permit user access to a permission if they are not logged in, so they'll need a reference to a specific authenticator. We'll also need to set up the permission dictionary upon initialization:

```
class Authorizor:
    def __init__(self, authenticator):
        self.authenticator = authenticator
        self.permissions = {}
```

Now we can write methods to add new permissions and to set up which users are associated with each permission:

```
    def add_permission(self, perm_name):
        '''Create a new permission that users
        can be added to'''
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            self.permissions[perm_name] = set()
        else:
            raise PermissionError("Permission Exists")

    def permit_user(self, perm_name, username):
        '''Grant the given permission to the user'''
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in self.authenticator.users:
                raise InvalidUsername(username)
            perm_set.add(username)
```

The first method allows us to create a new permission, unless it already exists, in which case an exception is raised. The second allows us to add a username to a permission, unless either the permission or the username doesn't yet exist.

We use `set` instead of `list` for usernames, so that even if you grant a user permission more than once, the nature of sets means the user is only in the set once. We'll discuss sets further in a later chapter.

A `PermissionError` error is raised in both methods. This new error doesn't require a username, so we'll make it extend `Exception` directly, instead of our custom `AuthException`:

```
class PermissionError(Exception):
    pass
```

Finally, we can add a method to check whether a user has a specific `permission` or not. In order for them to be granted access, they have to be both logged into the authenticator and in the set of people who have been granted access to that privilege. If either of these conditions is unsatisfied, an exception is raised:

```
    def check_permission(self, perm_name, username):
        if not self.authenticator.is_logged_in(username):
            raise NotLoggedInError(username)
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in perm_set:
                raise NotPermittedError(username)
            else:
                return True
```

There are two new exceptions in here; they both take usernames, so we'll define them as subclasses of `AuthException`:

```
class NotLoggedInError(AuthException):
    pass

class NotPermittedError(AuthException):
    pass
```

Finally, we can add a default `authorizor` to go with our default authenticator:

```
authorizor = Authorizor(authenticator)
```

That completes a basic authentication/authorization system. We can test the system at the Python prompt, checking to see whether a user, `joe`, is permitted to do tasks in the paint department:

```
>>> import auth
>>> auth.authenticator.add_user("joe", "joepassword")
>>> auth.authorizor.add_permission("paint")
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "auth.py", line 109, in check_permission
    raise NotLoggedInError(username)
auth.NotLoggedInError: joe
>>> auth.authenticator.is_logged_in("joe")
False
>>> auth.authenticator.login("joe", "joepassword")
True
>>> auth.authorizor.check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 116, in check_permission
    raise NotPermittedError(username)
auth.NotPermittedError: joe
>>> auth.authorizor.check_permission("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 111, in check_permission
    perm_set = self.permissions[perm_name]
KeyError: 'mix'

During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 113, in check_permission
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist
>>> auth.authorizor.permit_user("mix", "joe")
Traceback (most recent call last):
  File "auth.py", line 99, in permit_user
    perm_set = self.permissions[perm_name]
KeyError: 'mix'
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 101, in permit_user
    raise PermissionError("Permission does not exist")
auth.PermissionError: Permission does not exist
>>> auth.authorizor.permit_user("paint", "joe")
>>> auth.authorizor.check_permission("paint", "joe")
True
```

While verbose, the preceding output shows all of our code and most of our exceptions in action, but to really understand the API we've defined, we should write some exception handling code that actually uses it. Here's a basic menu interface that allows certain users to change or test a program:

```
import auth

# Set up a test user and permission
auth.authenticator.add_user("joe", "joepassword")
auth.authorizor.add_permission("test program")
auth.authorizor.add_permission("change program")
auth.authorizor.permit_user("test program", "joe")


class Editor:
    def __init__(self):
        self.username = None
        self.menu_map = {
            "login": self.login,
            "test": self.test,
            "change": self.change,
            "quit": self.quit,
        }

    def login(self):
        logged_in = False
        while not logged_in:
            username = input("username: ")
            password = input("password: ")
            try:
                logged_in = auth.authenticator.login(username, password)
            except auth.InvalidUsername:
                print("Sorry, that username does not exist")
            except auth.InvalidPassword:
                print("Sorry, incorrect password")
            else:
                self.username = username

    def is_permitted(self, permission):
        try:
            auth.authorizor.check_permission(permission, self.username)
        except auth.NotLoggedInError as e:
            print("{} is not logged in".format(e.username))
            return False
        except auth.NotPermittedError as e:
            print("{} cannot {}".format(e.username, permission))
            return False
        else:
```

```
            return True

    def test(self):
        if self.is_permitted("test program"):
            print("Testing program now...")

    def change(self):
        if self.is_permitted("change program"):
            print("Changing program now...")

    def quit(self):
        raise SystemExit()

    def menu(self):
        try:
            answer = ""
            while True:
                print(
                    """
Please enter a command:
\tlogin\tLogin
\ttest\tTest the program
\tchange\tChange the program
\tquit\tQuit
"""
                )
                answer = input("enter a command: ").lower()
                try:
                    func = self.menu_map[answer]
                except KeyError:
                    print("{} is not a valid option".format(answer))
                else:
                    func()
        finally:
            print("Thank you for testing the auth module")


Editor().menu()
```

This rather long example is conceptually very simple. The `is_permitted` method is probably the most interesting; this is a mostly internal method that is called by both `test` and `change` to ensure the user is permitted access before continuing. Of course, those two methods are stubs, but we aren't writing an editor here; we're illustrating the use of exceptions and exception handlers by testing an authentication and authorization framework.

Exercises

If you've never dealt with exceptions before, the first thing you need to do is look at any old Python code you've written and notice if there are places you should have been handling exceptions. How would you handle them? Do you need to handle them at all? Sometimes, letting the exception propagate to the console is the best way to communicate to the user, especially if the user is also the script's coder. Sometimes, you can recover from the error and allow the program to continue. Sometimes, you can only reformat the error into something the user can understand and display it to them.

Some common places to look are file I/O (is it possible your code will try to read a file that doesn't exist?), mathematical expressions (is it possible that a value you are dividing by is zero?), list indices (is the list empty?), and dictionaries (does the key exist?). Ask yourself whether you should ignore the problem, handle it by checking values first, or handle it with an exception. Pay special attention to areas where you might have used `finally` and `else` to ensure the correct code is executed under all conditions.

Now write some new code. Think of a program that requires authentication and authorization, and try writing some code that uses the `auth` module we built in the case study. Feel free to modify the module if it's not flexible enough. Try to handle all the exceptions in a sensible way. If you're having trouble coming up with something that requires authentication, try adding authorization to the Notepad example from `Chapter 2`, *Objects in Python*, or add authorization to the `auth` module itself—it's not a terribly useful module if just anybody can start adding permissions! Maybe require an administrator username and password before allowing privileges to be added or changed.

Finally, try to think of places in your code where you can raise exceptions. It can be in code you've written or are working on; or you can write a new project as an exercise. You'll probably have the best luck for designing a small framework or API that is meant to be used by other people; exceptions are a terrific communication tool between your code and someone else's. Remember to design and document any self-raised exceptions as part of the API, or they won't know whether or how to handle them!

Summary

In this chapter, we went into the gritty details of raising, handling, defining, and manipulating exceptions. Exceptions are a powerful way to communicate unusual circumstances or error conditions without requiring a calling function to explicitly check return values. There are many built-in exceptions and raising them is trivially easy. There are several different syntaxes for handling different exception events.

In the next chapter, everything we've studied so far will come together as we discuss how object-oriented programming principles and structures should best be applied in Python applications.

5
When to Use Object-Oriented Programming

In previous chapters, we've covered many of the defining features of object-oriented programming. We now know the principles and paradigms of object-oriented design, and we've covered the syntax of object-oriented programming in Python.

Yet, we don't know exactly how and, especially, when to utilize these principles and syntax in practice. In this chapter, we'll discuss some useful applications of the knowledge we've gained, looking at some new topics along the way:

- How to recognize objects
- Data and behaviors, once again
- Wrapping data behaviors using properties
- Restricting data using behaviors
- The Don't Repeat Yourself principle
- Recognizing repeated code

Treat objects as objects

This may seem obvious; you should generally give separate objects in your problem domain a special class in your code. We've seen examples of this in the case studies in previous chapters: first, we identify objects in the problem, and then model their data and behaviors.

Identifying objects is a very important task in object-oriented analysis and programming. But it isn't always as easy as counting the nouns in short paragraphs that, frankly, I have constructed explicitly for that purpose. Remember, objects are things that have both data and behavior. If we are working only with data, we are often better off storing it in a list, set, dictionary, or other Python data structure (which we'll be covering thoroughly in `Chapter 6`, *Python Data Structures*). On the other hand, if we are working only with behavior, but no stored data, a simple function is more suitable.

An object, however, has both data and behavior. Proficient Python programmers use built-in data structures unless (or until) there is an obvious need to define a class. There is no reason to add an extra level of abstraction if it doesn't help organize our code. On the other hand, the *obvious* need is not always self-evident.

We can often start our Python programs by storing data in a few variables. As the program expands, we will later find that we are passing the same set of related variables to a set of functions. This is the time to think about grouping both variables and functions into a class. If we are designing a program to model polygons in two-dimensional space, we might start with each polygon represented as a list of points. The points would be modeled as two tuples (*x*, *y*) describing where that point is located. This is all data, stored in a set of nested data structures (specifically, a list of tuples):

```
square = [(1,1), (1,2), (2,2), (2,1)]
```

Now, if we want to calculate the distance around the perimeter of the polygon, we need to sum the distances between each point. To do this, we need a function to calculate the distance between two points. Here are two such functions:

```
import math

def distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def perimeter(polygon):
    perimeter = 0
    points = polygon + [polygon[0]]
    for i in range(len(polygon)):
        perimeter += distance(points[i], points[i+1])
    return perimeter
```

Now, as object-oriented programmers, we clearly recognize that a `polygon` class could encapsulate the list of points (data) and the `perimeter` function (behavior). Further, a `point` class, such as we defined in `Chapter 2`, *Objects in Python*, might encapsulate the `x` and `y` coordinates and the `distance` method. The question is: is it valuable to do this?

For the previous code, maybe yes, maybe no. With our recent experience in object-oriented principles, we can write an object-oriented version in record time. Let's compare them as follows:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, p2):
        return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)

class Polygon:
    def __init__(self):
        self.vertices = []

    def add_point(self, point):
        self.vertices.append((point))

    def perimeter(self):
        perimeter = 0
        points = self.vertices + [self.vertices[0]]
        for i in range(len(self.vertices)):
            perimeter += points[i].distance(points[i+1])
        return perimeter
```

As we can see from the highlighted sections, there is twice as much code here as there was in our earlier version, although we could argue that the `add_point` method is not strictly necessary.

Now, to understand the differences a little better, let's compare the two APIs in use. Here's how to calculate the perimeter of a square using the object-oriented code:

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> square.perimeter()
4.0
```

That's fairly succinct and easy to read, you might think, but let's compare it to the function-based code:

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
>>> perimeter(square)
4.0
```

Hmm, maybe the object-oriented API isn't so compact! That said, I'd argue that it was easier to *read* than the functional example. How do we know what the list of tuples is supposed to represent in the second version? How do we remember what kind of object we're supposed to pass into the `perimeter` function? (a list of two tuples? That's not intuitive!) We would need a lot of documentation to explain how these functions should be used.

In contrast, the object-oriented code is relatively self-documenting. We just have to look at the list of methods and their parameters to know what the object does and how to use it. By the time we wrote all the documentation for the functional version, it would probably be longer than the object-oriented code.

Finally, code length is not a good indicator of code complexity. Some programmers get hung up on complicated *one liners* that do an incredible amount of work in one line of code. This can be a fun exercise, but the result is often unreadable, even to the original author the following day. Minimizing the amount of code can often make a program easier to read, but do not blindly assume this is the case.

Luckily, this trade-off isn't necessary. We can make the object-oriented `Polygon` API as easy to use as the functional implementation. All we have to do is alter our `Polygon` class so that it can be constructed with multiple points. Let's give it an initializer that accepts a list of `Point` objects. In fact, let's allow it to accept tuples too, and we can construct the `Point` objects ourselves, if needed:

```
def __init__(self, points=None):
    points = points if points else []
    self.vertices = []
    for point in points:
        if isinstance(point, tuple):
            point = Point(*point)
        self.vertices.append(point)
```

This initializer goes through the list and ensures that any tuples are converted to points. If the object is not a tuple, we leave it as is, assuming that it is either a `Point` object already, or an unknown duck-typed object that can act like a `Point` object.

If you are experimenting with the above code, you could subclass `Polygon` and override the `__init__` function instead of replacing the initializer or copying the `add_point` and `perimeter` methods.

Still, there's no clear winner between the object-oriented and more data-oriented versions of this code. They both do the same thing. If we have new functions that accept a polygon argument, such as `area(polygon)` or `point_in_polygon(polygon, x, y)`, the benefits of the object-oriented code become increasingly obvious. Likewise, if we add other attributes to the polygon, such as `color` or `texture`, it makes more and more sense to encapsulate that data into a single class.

The distinction is a design decision, but in general, the more important a set of data is, the more likely it is to have multiple functions specific to that data, and the more useful it is to use a class with attributes and methods instead.

When making this decision, it also pays to consider how the class will be used. If we're only trying to calculate the perimeter of one polygon in the context of a much greater problem, using a function will probably be quickest to code and easier to use *one time only*. On the other hand, if our program needs to manipulate numerous polygons in a wide variety of ways (calculating the perimeter, area, and intersection with other polygons, moving or scaling them, and so on), we have almost certainly identified an object; one that needs to be extremely versatile.

Additionally, pay attention to the interaction between objects. Look for inheritance relationships; inheritance is impossible to model elegantly without classes, so make sure to use them. Look for the other types of relationships we discussed in `Chapter 1`, *Object-Oriented Design*, association and composition. Composition can, technically, be modeled using only data structures; for example, we can have a list of dictionaries holding tuple values, but it is sometimes less complicated to create a few classes of objects, especially if there is behavior associated with the data.

Don't rush to use an object just because you can use an object, but don't neglect to create a class when you need to use a class.

Adding behaviors to class data with properties

Throughout this book, we've focused on the separation of behavior and data. This is very important in object-oriented programming, but we're about to see that, in Python, the distinction is uncannily blurry. Python is very good at blurring distinctions; it doesn't exactly help us to *think outside the box*. Rather, it teaches us to stop thinking about the box.

Before we get into the details, let's discuss some bad object-oriented theory. Many object-oriented languages teach us to never access attributes directly (Java is the most notorious). They insist that we write attribute access like this:

```
class Color:
    def __init__(self, rgb_value, name):
        self._rgb_value = rgb_value
        self._name = name

    def set_name(self, name):
        self._name = name
    def get_name(self):
        return self._name
```

The variables are prefixed with an underscore to suggest that they are private (other languages would actually force them to be private). Then, the `get` and `set` methods provide access to each variable. This class would be used in practice as follows:

```
>>> c = Color("#ff0000", "bright red")
>>> c.get_name()
'bright red'
>>> c.set_name("red")
>>> c.get_name()
'red'
```

This is not nearly as readable as the direct access version that Python favors:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self.name = name

c = Color("#ff0000", "bright red")
print(c.name)
c.name = "red"
print(c.name)
```

So, why would anyone insist upon the method-based syntax? Their reasoning is that, someday, we may want to add extra code when a value is set or retrieved. For example, we could decide to cache a value to avoid complex computations, or we might want to validate that a given value is a suitable input.

In code, for example, we could decide to change the `set_name()` method as follows:

```
def set_name(self, name):
    if not name:
        raise Exception("Invalid Name")
    self._name = name
```

Now, in Java and similar languages, if we had written our original code for direct attribute access, and then later changed it to a method like the preceding one, we'd have a problem: anyone who had written code that accessed the attribute directly would now have to access a method. If they didn't then change the access style from attribute access to a function call, their code will be broken.

The mantra in these languages is that we should never make public members private. This doesn't make much sense in Python since there isn't any real concept of private members!

Python gives us the `property` keyword to make methods that *look* like attributes. We can therefore write our code to use direct member access, and if we ever unexpectedly need to alter the implementation to do some calculation when getting or setting that attribute's value, we can do so without changing the interface. Let's see how it looks:

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name

    name = property(_get_name, _set_name)
```

Compared to the earlier class, we first change the `name` attribute into a (semi-)private `_name` attribute. Then, we add two more (semi-)private methods to get and set that variable, performing our validation when we set it.

Finally, we have the `property` declaration at the bottom. This is the Python magic. It creates a new attribute on the `Color` class called `name`, to replace the direct `name` attribute. It sets this attribute to be a **property**. Under the hood, `property` calls the two methods we just created whenever the value is accessed or changed. This new version of the `Color` class can be used exactly the same way as the earlier version, yet it now performs validation when we set the `name` attribute:

```
>>> c = Color("#0000ff", "bright red")
>>> print(c.name)
bright red
>>> c.name = "red"
>>> print(c.name)
red
>>> c.name = ""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "setting_name_property.py", line 8, in _set_name
    raise Exception("Invalid Name")
Exception: Invalid Name
```

So, if we'd previously written code to access the `name` attribute, and then changed it to use our `property`-based object, the previous code would still work, unless it was sending an empty `property` value, which is the behavior we wanted to forbid in the first place. Success!

Bear in mind that, even with the `name` property, the previous code is not 100% safe. People can still access the `_name` attribute directly and set it to an empty string if they want to. But if they access a variable we've explicitly marked with an underscore to suggest it is private, they're the ones that have to deal with the consequences, not us.

Properties in detail

Think of the `property` function as returning an object that proxies any requests to set or access the attribute value through the methods we have specified. The `property` built-in is like a constructor for such an object, and that object is set as the public-facing member for the given attribute.

This `property` constructor can actually accept two additional arguments, a `delete` function and a docstring for the property. The `delete` function is rarely supplied in practice, but it can be useful for logging the fact that a value has been deleted, or possibly to veto deleting if we have reason to do so. The docstring is just a string describing what the property does, no different from the docstrings we discussed in `Chapter 2`, *Objects in Python*. If we do not supply this parameter, the docstring will instead be copied from the docstring for the first argument: the `getter` method. Here is a silly example that states whenever any of the methods are called:

```
class Silly:
    def _get_silly(self):
        print("You are getting silly")
        return self._silly

    def _set_silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    def _del_silly(self):
        print("Whoah, you killed silly!")
        del self._silly

    silly = property(_get_silly, _set_silly, _del_silly, "This is a silly
property")
```

If we actually use this class, it does indeed print out the correct strings when we ask it to:

```
>>> s = Silly()
>>> s.silly = "funny"
You are making silly funny
>>> s.silly
You are getting silly
'funny'
>>> del s.silly
Whoah, you killed silly!
```

Further, if we look at the help file for the `Silly` class (by issuing `help(Silly)` at the interpreter prompt), it shows us the custom docstring for our `silly` attribute:

```
Help on class Silly in module __main__:

class Silly(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
```

```
|  __weakref__
|      list of weak references to the object (if defined)
|
|  silly
|      This is a silly property
```

Once again, everything is working as we planned. In practice, properties are normally only defined with the first two parameters: the `getter` and `setter` functions. If we want to supply a docstring for a property, we can define it on the `getter` function; the property proxy will copy it into its own docstring. The `delete` function is often left empty because object attributes are so rarely deleted. If a coder does try to delete a property that doesn't have a `delete` function specified, it will raise an exception. Therefore, if there is a legitimate reason to delete our property, we should supply that function.

Decorators – another way to create properties

If you've never used Python decorators before, you might want to skip this section and come back to it after we've discussed the decorator pattern in `Chapter 10`, *Python Design Patterns I*. However, you don't need to understand what's going on to use the decorator syntax in order to make property methods more readable.

The `property` function can be used with the decorator syntax to turn a `get` function into a `property` function, as follows:

```
class Foo:
    @property
    def foo(self):
        return "bar"
```

This applies the `property` function as a decorator, and is equivalent to the previous `foo = property(foo)` syntax. The main difference, from a readability perspective, is that we get to mark the `foo` function as a property at the top of the method, instead of after it is defined, where it can be easily overlooked. It also means we don't have to create private methods with underscore prefixes just to define a property.

Going one step further, we can specify a `setter` function for the new property as follows:

```
class Foo:
    @property
    def foo(self):
        return self._foo

    @foo.setter
    def foo(self, value):
```

```
        self._foo = value
```

This syntax looks pretty odd, although the intent is obvious. First, we decorate the `foo` method as a getter. Then, we decorate a second method with exactly the same name by applying the `setter` attribute of the originally decorated `foo` method! The `property` function returns an object; this object always comes with its own `setter` attribute, which can then be applied as a decorator to other functions. Using the same name for the get and set methods is not required, but it does help to group together the multiple methods that access one property.

We can also specify a `delete` function with `@foo.deleter`. We cannot specify a docstring using `property` decorators, so we need to rely on the property copying the docstring from the initial getter method. Here's our previous `Silly` class rewritten to use `property` as a decorator:

```
class Silly:
    @property
    def silly(self):
        "This is a silly property"
        print("You are getting silly")
        return self._silly

    @silly.setter
    def silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    @silly.deleter
    def silly(self):
        print("Whoah, you killed silly!")
        del self._silly
```

This class operates *exactly* the same as our earlier version, including the help text. You can use whichever syntax you feel is more readable and elegant.

Deciding when to use properties

With the built-in property clouding the division between behavior and data, it can be confusing to know when to choose an attribute, or a method, or a property. The use case example we saw earlier is one of the most common uses of properties; we have some data on a class that we later want to add behavior to. There are also other factors to take into account when deciding to use a property.