Chapter 14

Object-Oriented Programming

About a year or so after I started programming, I decided to make a game to play *Wheel of Fortune*. I wrote the program in the BASIC programming language and it got to be pretty large, a couple thousand lines. It mostly worked, but whenever I tried to fix something, my fix would break something in a completely different part of the program. I would then fix that and break something else. Eventually I got the program working, but after a while I was afraid to even touch it.

The problem with the program was that each part of the program had access to the variables from the other parts. A change of a variable in one part would mess up things in the others. One solution to this type of problem is *object-oriented programming*. One of its chief benefits is *encapsulation*, where you divide your program into pieces and each piece internally operates independently of the others. The pieces interact with each other, but they don't need to know exactly how each one accomplishes its tasks. This requires some planning and set-up time before you start your program, and so it is not always appropriate for short programs, like many of the ones that we have written so far.

We will just cover the basics of object-oriented programming here. Object-oriented programming is used extensively in software design and I would recommend picking up another book on programming or software design to learn more about designing programs in an object-oriented way.

14.1 Python is objected-oriented

Python is an object-oriented programming language, and we have in fact been using many object-oriented concepts already. The key notion is that of an *object*. An object consists of two things: data and functions (called *methods*) that work with that data. As an example, strings in Python are objects. The data of the string object is the actual characters that make up that string. The methods are things like `lower`, `replace`, and `split`. In Python, everything is an object. That includes not only strings and lists, but also integers, floats, and even functions themselves.

14.2 Creating your own classes

A *class* is a template for objects. It contains the code for all the object's methods.

A simple example Here is a simple example to demonstrate what a class looks like. It does not do anything interesting.

```
class Example:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

e = Example(8, 6)
print(e.add())
```

- To create a class, we use the `class` statement. Class names usually start with a capital.
- Most classes will have a method called `__init__`. The underscores indicate that it is a special kind of method. It is called a *constructor*, and it is automatically called when someone creates a new object from your class. The constructor is usually used to set up the class's variables. In the above program, the constructor takes two values, `a` and `b`, and assigns the class variables `a` and `b` to those values.
- The first argument to every method in your class is a special variable called `self`. Every time your class refers to one of its variables or methods, it must precede them by `self`. The purpose of `self` is to distinguish your class's variables and methods from other variables and functions in the program.
- To create a new object from the class, you call the class name along with any values that you want to send to the constructor. You will usually want to assign it to a variable name. This is what the line `e=Example(8,6)` does.
- To use the object's methods, use the dot operator, as in `e.addmod()`.

A more practical example Here is a class called `Analyzer` that performs some simple analysis on a string. There are methods to return how many words are in the string, how many are of a given length, and how many start with a given string.

```
from string import punctuation

class Analyzer:
    def __init__(self, s):
        for c in punctuation:
            s = s.replace(c,'')
```

```
        s = s.lower()
        self.words = s.split()

    def number_of_words(self):
        return len(self.words)

    def starts_with(self, s):
        return len([w for w in self.words if w[:len(s)]==s])

    def number_with_length(self, n):
        return len([w for w in self.words if len(w)==n])


s = 'This is a test of the class.'
analyzer = Analyzer(s)
print(analyzer.words)
print('Number of words:', analyzer.number_of_words())
print('Number of words starting with "t":', analyzer.starts_with('t'))
print('Number of 2-letter words:', analyzer.number_with_length(2))
```

```
['this', 'is', 'a', 'test', 'of', 'the', 'class']
Number of words: 7
Number of words starting with "t": 3
Number of 2-letter words: 2
```

A few notes about this program:

- One reason why we would wrap this code up in a class is we can then use it a variety of different programs. It is also good just for organizing things. If all our program is doing is just analyzing some strings, then there's not too much of a point of writing a class, but if this were to be a part of a larger program, then using a class provides a nice way to separate the `Analyzer` code from the rest of the code. It also means that if we were to change the internals of the `Analyzer` class, the rest of the program would not be affected as long as the interface, the way the rest of the program interacts with the class, does not change. Also, the `Analyzer` class can be imported as-is in other programs.

- The following line accesses a class variable:

  ```
  print(analyzer.words)
  ```

  You can also change class variables. This is not always a good thing. In some cases this is convenient, but you have to be careful with it. Indiscriminate use of class variables goes against the idea of encapsulation and can lead to programming errors that are hard to fix. Some other object-oriented programming languages have a notion of public and private variables, public variables being those that anyone can access and change, and private variables being only accessible to methods within the class. In Python all variables are public, and it is up to the programmer to be responsible with them. There is a convention where you name those variables that you want to be private with a starting underscore, like `_var1`. This serves to let others know that this variable is internal to the class and shouldn't be touched.

14.3 Inheritance

In object-oriented programming there is a concept called *inheritance* where you can create a class that builds off of another class. When you do this, the new class gets all of the variables and methods of the class it is inheriting from (called the *base class*). It can then define additional variables and methods that are not present in the base class, and it can also *override* some of the methods of the base class. That is, it can rewrite them to suit its own purposes. Here is a simple example:

```
class Parent:
    def __init__(self, a):
        self.a = a
    def method1(self):
        print(self.a*2)
    def method2(self):
        print(self.a+'!!!')

class Child(Parent):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def method1(self):
        print(self.a*7)
    def method3(self):
        print(self.a + self.b)

p = Parent('hi')
c = Child('hi', 'bye')

print('Parent method 1: ', p.method1())
print('Parent method 2: ', p.method2())
print()
print('Child method 1: ', c.method1())
print('Child method 2: ', c.method2())
print('Child method 3: ', c.method3())
```

```
Parent method 1: hihi
Parent method 2: hi!!!

Child method 1: hihihihihihihi
Child method 2: hi!!!
Child method 3: hibye
```

We see in the example above that the child has overridden the parent's `method1`, causing it to now repeat the string seven times. The child has inherited the parent's `method2`, so it can use it without having to define it. The child also adds some features to the parent class, namely a new variable `b` and a new method, `method3`.

A note about syntax: when inheriting from a class, you indicate the parent class in parentheses in the `class` statement.

If the child class adds some new variables, it can call the parent class's constructor as demonstrated below. Another use is if the child class just wants to add on to one of the parent's methods. In the example below, the child's `print_var` method calls the parent's `print_var` method and adds an additional line.

```
class Parent:
    def __init__(self, a):
        self.a = a

    def print_var(self):
        print("The value of this class's variables are:")
        print(self.a)

class Child(Parent):
    def __init__(self, a, b):
        Parent.__init__(self, a)
        self.b = b

    def print_var(self):
        Parent.print_var(self)
        print(self.b)
```

Note You can also inherit from Python built-in types, like strings (**`str`**) and lists (**`list`**), as well as any classes defined in the various modules that come with Python.

Note Your code can inherit from more than one class at a time, though this can be a little tricky.

14.4 A playing-card example

In this section we will show how to design a program with classes. We will create a simple hi-lo card game where the user is given a card and they have to say if the next card will be higher or lower than it. This game could easily be done without classes, but we will create classes to represent a card and a deck of cards, and these classes can be reused in other card games.

We start with a class for a playing card. The data associated with a card consists of its value (2 through 14) and its suit. The `Card` class below has only one method, `__str__`. This is a special method that, among other things, tells the **`print`** function how to print a `Card` object.

```
class Card:
    def __init__(self, value, suit):
        self.value = value
        self.suit = suit

    def __str__(self):
        names = ['Jack', 'Queen', 'King', 'Ace']
        if self.value <= 10:
```

```
            return '{} of {}'.format(self.value, self.suit)
        else:
            return '{} of {}'.format(names[self.value-11], self.suit)
```

Next we have a class to represent a group of cards. Its data consists of a list of `Card` objects. It has a number of methods: `nextCard` which removes the first card from the list and returns it; `hasCard` which returns **`True`** or **`False`** depending on if there are any cards left in the list; `size`, which returns how many cards are in the list; and `shuffle`, which shuffles the list.

```
import random

class Card_group:
    def __init__(self, cards=[]):
        self.cards = cards

    def nextCard(self):
        return self.cards.pop(0)

    def hasCard(self):
        return len(self.cards)>0

    def size(self):
        return len(self.cards)

    def shuffle(self):
        random.shuffle(self.cards)
```

We have one more class `Standard_deck`, which inherits from `Card_group`. The idea here is that `Card_group` represents an arbitrary group of cards, and `Standard_deck` represents a specific group of cards, namely the standard deck of 52 cards used in most card games.

```
class Standard_deck(Card_group):
    def __init__(self):
        self.cards = []
        for s in ['Hearts', 'Diamonds', 'Clubs', 'Spades']:
            for v in range(2,15):
                self.cards.append(Card(v, s))
```

Suppose we had just created a single class that represented a standard deck along with all the common operations like shuffling. If we wanted to create a new class for a Pinochle game or some other game that doesn't use the standard deck, then we would have to copy and paste the standard deck code and modify lots of things. By doing things more generally, like we've done here, each time we want a new type of deck, we can build off of of (inherit from) what is in `Card_group`. For instance, a Pinochle deck class would look like this:

```
class Pinochle_deck(Card_group):
    def __init__(self):
        self.cards = []
        for s in ['Hearts', 'Diamonds', 'Clubs', 'Spades']*2:
```

```
            for v in range(9,15):
                self.cards.append(Card(v, s))
```

A Pinochle deck has only nines, tens, jacks, queens, kings, and aces. There are two copies of each card in each suit.

Here is the hi-low program that uses the classes we have developed here. One way to think of what we have done with the classes is that we have built up a miniature card programming language, where we can think about how a card game works and not have to worry about exactly how cards are shuffled or dealt or whatever, since that is wrapped up into the classes. For the hi-low game, we get a new deck of cards, shuffle it, and then deal out the cards one at a time. When we run out of cards, we get a new deck and shuffle it. A nice feature of this game is that it deals out all 52 cards of a deck, so a player can use their memory to help them play the game.

```
deck = Standard_deck()
deck.shuffle()

new_card = deck.nextCard()
print('\n', new_card)
choice = input("Higher (h) or lower (l): ")
streak = 0

while (choice=='h' or choice=='l'):
    if not deck.hasCard():
        deck = Standard_deck()
        deck.shuffle()

    old_card = new_card
    new_card = deck.nextCard()

    if (choice.lower()=='h' and new_card.value>old_card.value or\
        choice.lower()=='l' and new_card.value<old_card.value):
        streak = streak + 1
        print("Right!  That's", streak, "in a row!")

    elif (choice.lower()=='h' and new_card.value<old_card.value or\
          choice.lower()=='l' and new_card.value>old_card.value):
        streak = 0
        print('Wrong.')
    else:
        print('Push.')

    print('\n', new_card)
    choice = input("Higher (h) or lower (l): ")
```

```
 King of Clubs
Higher (h) or lower (l): l
Right!  That's 1 in a row!

 2 of Spades
```

```
Higher (h) or lower (l): h
Right!  That's 2 in a row!
```

14.5 A Tic-tac-toe example

In this section we create an object-oriented Tic-tac-toe game. We use a class to wrap up the logic of the game. The class contains two variables, an integer representing the current player, and a 3×3 list representing the board. The board variable consists of zeros, ones, and twos. Zeros represent an open spot, while ones and twos represent spots marked by players 1 and 2, respectively. There are four methods:

- `get_open_spots` — returns a list of the places on the board that have not yet been marked by players
- `is_valid_move` — takes a row and a column representing a potential move, and returns **`True`** if move is allowed and **`False`** otherwise
- `make_move` — takes a row and a column representing a potential move, calls `is_valid_move` to see if the move is okay, and if it is, sets the board array accordingly and changes the player
- `check_for_winner` — scans through the board list and returns 1 if player 1 has won, 2 if player 2 has won, 0 if there are no moves remaining and no winner, and -1 if the game should continue

Here is the code for the class:

```
class tic_tac_toe:
    def __init__(self):
        self.B = [[0,0,0],
                  [0,0,0],
                  [0,0,0]]
        self.player = 1

    def get_open_spots(self):
        return [[r,c] for r in range(3) for c in range(3)
                if self.B[r][c]==0]

    def is_valid_move(self,r,c):
        if 0<=r<=2 and 0<=c<=2 and self.board[r][c]==0:
            return True
        return False

    def make_move(self,r,c):
        if self.is_valid_move(r,c):
            self.B[r][c] = self.player
            self.player = (self.player+2)%2 + 1

    def check_for_winner(self):
```

```
        for c in range(3):
            if self.B[0][c]==self.B[1][c]==self.B[2][c]!=0:
                return self.B[0][c]
        for r in range(3):
            if self.B[r][0]==self.B[r][1]==self.B[r][2]!=0:
                return self.B[r][0]
        if self.B[0][0]==self.B[1][1]==self.B[2][2]!=0:
            return self.B[0][0]
        if self.B[2][0]==self.B[1][1]==self.B[0][2]!=0:
            return self.B[2][0]
        if self.get_open_spots()==[]:
            return 0
        return -1
```

This class consists of the logic of the game. There is nothing in the class that is specific to the user interface. Below we have a text-based interface using print and input statements. If we decide to use a graphical interface, we can use the `Tic_tac_toe` class without having to change anything about it. Note that the `get_open_spots` method is not used by this program. It is useful, however, if you want to implement a computer player. A simple computer player would call that method and use `random.choice` method to choose a random element from the returned list of spots.

```
def print_board():
    chars = ['-', 'X', 'O']
    for r in range(3):
        for c in range(3):
            print(chars[game.B[r][c]], end=' ')
        print()

game = tic_tac_toe()
while game.check_for_winner()==-1:
    print_board()
    r,c = eval(input('Enter spot, player ' + str(game.player) + ': '))
    game.make_move(r,c)

print_B()
x = game.check_for_winner()
if x==0:
    print("It's a draw.")
else:
    print('Player', x, 'wins!')
```

Here is what the first couple of turns look like:

```
- - -
- - -
- - -
Enter spot, player 1: 1,1
- - -
- X -
- - -
```

```
Enter spot, player 2: 0,2
- - O
- X -
- - -
Enter spot, player 1:
```

14.6 Further topics

- **Special methods** — We have seen two special methods already, the constructor `__init__` and the method `__str__` which determines what your objects look like when printed. There are many others. For instance, there is `__add__` that allows your object to use the + operator. There are special methods for all the Python operators. There is also a method called `__len__` which allows your object to work with the built in `len` function. There is even a special method, `__getitem__` that lets your program work with list and string brackets `[]`.

- **Copying objects** — If you want to make a copy of an object `x`, it is not enough to do the following:

  ```
  x_copy = x
  ```

  The reason is discussed in Section 19.1. Instead, do the following:

  ```
  from copy import copy
  x_copy = copy(x)
  ```

- **Keeping your code in multiple files** — If you want to reuse a class in several programs, you do not have to copy and paste the code into each. You can save it in a file and use an import statement to import it into your programs. The file will need to be somewhere your program can find it, like in the same directory.

  ```
  from analyzer import Analyzer
  ```

14.7 Exercises

1. Write a class called `Investment` with fields called `principal` and `interest`. The constructor should set the values of those fields. There should be a method called `value_after` that returns the value of the investment after n years. The formula for this is $p(1+i)^n$, where p is the principal, and i is the interest rate. It should also use the special method `__str__` so that printing the object will result in something like below:

   ```
   Principal - $1000.00, Interest rate - 5.12%
   ```

2. Write a class called `Product`. The class should have fields called `name`, `amount`, and `price`, holding the product's name, the number of items of that product in stock, and the regular price of the product. There should be a method `get_price` that receives the number of items to be bought and returns a the cost of buying that many items, where the regular price

is charged for orders of less than 10 items, a 10% discount is applied for orders of between 10 and 99 items, and a 20% discount is applied for orders of 100 or more items. There should also be a method called `make_purchase` that receives the number of items to be bought and decreases `amount` by that much.

3. Write a class called `Password_manager`. The class should have a list called `old_passwords` that holds all of the user's past passwords. The last item of the list is the user's current password. There should be a method called `get_password` that returns the current password and a method called `set_password` that sets the user's password. The `set_password` method should only change the password if the attempted password is different from all the user's past passwords. Finally, create a method called `is_correct` that receives a string and returns a boolean **`True`** or **`False`** depending on whether the string is equal to the current password or not.

4. Write a class called `Time` whose only field is a time in seconds. It should have a method called `convert_to_minutes` that returns a string of minutes and seconds formatted as in the following example: if seconds is 230, the method should return `'5:50'`. It should also have a method called `convert_to_hours` that returns a string of hours, minutes, and seconds formatted analogously to the previous method.

5. Write a class called `Wordplay`. It should have a field that holds a list of words. The user of the class should pass the list of words they want to use to the class. There should be the following methods:

   - `words_with_length(length)` — returns a list of all the words of length `length`
   - `starts_with(s)` — returns a list of all the words that start with `s`
   - `ends_with(s)` — returns a list of all the words that end with `s`
   - `palindromes()` — returns a list of all the palindromes in the list
   - `only(L)` — returns a list of the words that contain only those letters in `L`
   - `avoids(L)` — returns a list of the words that contain none of the letters in `L`

6. Write a class called `Converter`. The user will pass a length and a unit when declaring an object from the class—for example, `c = Converter(9,'inches')`. The possible units are inches, feet, yards, miles, kilometers, meters, centimeters, and millimeters. For each of these units there should be a method that returns the length converted into those units. For example, using the `Converter` object created above, the user could call `c.feet()` and should get `0.75` as the result.

7. Use the `Standard_deck` class of this section to create a simplified version of the game *War*. In this game, there are two players. Each starts with half of a deck. The players each deal the top card from their decks and whoever has the higher card wins the other player's cards and adds them to the bottom of his deck. If there is a tie, the two cards are eliminated from play (this differs from the actual game, but is simpler to program). The game ends when one player runs out of cards.

8. Write a class that inherits from the `Card_group` class of this chapter. The class should represent a deck of cards that contains only hearts and spaces, with only the cards 2 through 10 in each suit. Add a method to the class called `next2` that returns the top two cards from the deck.

9. Write a class called `Rock_paper_scissors` that implements the logic of the game Rock-paper-scissors. For this game the user plays against the computer for a certain number of rounds. Your class should have fields for the how many rounds there will be, the current round number, and the number of wins each player has. There should be methods for getting the computer's choice, finding the winner of a round, and checking to see if someone has one the (entire) game. You may want more methods.

10. (a) Write a class called `Connect4` that implements the logic of a Connect4 game. Use the `Tic_tac_toe` class from this chapter as a starting point.

    (b) Use the `Connect4` class to create a simple text-based version of the game.

11. Write a class called `Poker_hand` that has a field that is a list of `Card` objects. There should be the following self-explanatory methods:

    ```
    has_royal_flush, has_straight_flush, has_four_of_a_kind,
    has_full_house, has_flush, has_straight,
    has_three_of_a_kind, has_two_pair, has_pair
    ```

    There should also be a method called `best` that returns a string indicating what the best hand is that can be made from those cards.