8.3 Special Methods

There are a number of method names that have special significance in Python. One of these we have already seen: the constructor method is always named __init__ ( ). This method is called whenever a new object of the class is created; its purpose is to give initial values to the instance variables of the object. In this section we will see a number of similar methods that have pre-defined meanings. All of these have names that start and end with two underscores.

First, the method __str__ (self) is called whenever the system needs to have a string representation of the object. This method should return the string representation. If x is an object of a class containing this method, the following statements will all result in calls to __str__ ( ):

```
print(x)
print("%s" % x)
y = str(x)
```

For example, the Person class from section 7.1 might have such a method:

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.age = 0

    def __str__(self):
        return "%s is %d years old."% (self.name,self.age)
```

Program 8.3.1: A Person constructor and __str__ method

This would eliminate the need for a separate Print( ) method for this class; we could use the standard Python **print** statement to print objects of the class. Of course, nothing requires us to return a string containing all of the instance variables of the class. For some applications we might want the string representation of a person to consist of just the persons name:

```
def __str__(self):
        return self.name
```

We can also define methods that implement arithmetic operators in any class. The methods:

```
__add__(self, x)
__sub__(self, x)
__mul__(self, x)
__div__(self, x)
```

are called when the operators +, −, ∗,and / are used. Each of these methods should return a new object that is the result of the operation. For example, if a and b are objects of a class that defines these operators, we might use the statement

```
c = a+b
```

Variable c then gets the value that is returned from the call to the method __add__(a, b). Argument self refers to the object that is the left operand and argument x is the right operand.

In the next example we define a class Cents that represents money. Objects of this class have one instance variable, which holds the value of the object in pennies (so a value of 420 represents $4.20). We define a __str__( ) method to allow objects of the class to be printed, and an __add__( ) method to allow monetary values to be added.

```
class Cents:
    def __init__(self, x):
        self.value = x

    def __str__(self):
        dollars = self.value/100
        cents = self.value % 100
        if cents < 10:
            return "$%d.0%d" % (dollars, cents)
        else:
            return "$%d.%d" % (dollars, cents)

    def __add__(self, x):
        v = self.value + x.value
        return Cents( v )

def main():
    x = Cents(405)
    y = Cents(995)
    print("%s + %s = %s" % (x, y, x+y))

main()
```

Program 8.3.2: Adding elements of a class

We can also implement methods that allow us to use comparison operators between objects of a class. The method names

```
__lt__(self, x)
__le__(self, x)
```

```
__eq__(self, x)
__ne__(self, x)
__ge__(self, x)
__gt__(self, x)
```

refer to the operations <, <=, ==, !=, >=, >. In particular, if the `__lt__()` method is defined, then lists of objects of this can be sorted with the list `sort()` method.

Our last example adds comparison operators to the `Name` class we created in Section 8.2. We use the usual phone-book ordering for names: `a` < `b` if `a`'s last name comes before `b`'s in alphabetical ordering, or if the two last names are the same and `a`'s first name comes before `b`'s first name. In the name class the instance variable that holds the last name is `self.family`, and the variable that holds the first name is `self.given`. Our comparison operator is thus

```
def __lt__(self, x):
    if self.family < x.family:
        return True
    elif self.family > x.family:
        return False
    elif self.given < x.given:
        return True
    else:
        return False
```

This turns the class definition into

```
class Name:
    def __init__(self, str):
        str = str.strip()
        if str == "":
            self.family = ""
            self.given = ""
        else:
            names = str.split()
            n = len(names)
            self.family = names[n-1]
            given = ""
            for name in names[0:n-1]:
                given = given + name + " "
            self.given = given

    def GivenName(self):
        return self.given

    def LastName(self):
        return self.family

    def FirstName(self):
        if self.given == "":
            return ""
        else:
            names = self.given.split()
            return names[0]

    def FullName(self):
        if self.given == "":
            return self.family
        else:
            return self.given+self.family

    def __lt__(self, x):
        if self.family < x.family:
            return True
        elif self.family > x.family:
            return False
        elif self.given < x.given:
            return True
        else:
            return False
```

The Name class with a comparison operation

We now return to the Person class that uses Name. If we add a simple comparison function to this, just comparing the names, we can then sort lists of Persons:

```
import MyNameClass

class Person:
    def __init__(self, myName):
        self.name = MyNameClass.Name(myName)
        self.age = 0

    def SetAge( self, a ):
            self.age = a

    def GetOlder(self):
            self.age = self.age + 1

    def Print(self):
        print( "%s %s" %(self.name.FirstName(),
                         self.name.LastName()))
    def __lt__(self, x):
        return self.name < x.name

def main():
    L = []
    L.append( Person("Harry Potter") )
    L.append( Person("Hermione Granger") )
    L.append( Person("Ron Weasley") )
    L.append( Person("Albus Dumbledore") )
    L.append( Person("Severus Snape") )
    L.append( Person("Draco Malfoy") )

    L.sort()
    for person in L:
        person.Print()


main()
```

Program 8.3.3: Sorting elements of the Person class

This outputs

Albus Dumbledore
Hermione Granger
Draco Malfoy
Harry Potter
Severus Snape
Ron Weasley