8.2 Examples of Classes

Don't be put off by the terminology of classes we presented in the preceding section. *Designing* classes that appropriately break down complex data is a skill that takes practice to learn. In most situations *implementing* classes is easy and straightforward. One nice thing about coding with classes is that you always have a place to start – the constructor (remember that in Python this is a function called `__init__(self ...)`) should assign a value to each of the instance variables of the class. Whatever description you have of the class should tell you what those instance variables are.

Example 1: *Write a class that will hold data about pets. The class should include variables for the pet's name, species (dog, cat, hamster, etc.), color and age.* To implement this we need to make a few decisions. One concerns changeability of the variables: once we set the species for a pet, there is probably no need to ever change it. The same is true of the pet's color. It may or may not be true for the name; for simplicity in this example we will assume that names are not modifiable. Age, on the other hand, is certainly something that changes. For all of the variables we will create *getter* methods – methods that return the value of the variables. For the changeable variables we also need to create *setter* methods – methods that allow us to set or change the value of the variable. The next decision regards the arguments to the constructor. All of these variables will be set in the constructor; some will get default values, some will get values specified in the arguments to the constructor. We don't want to make calls to the constructor too long, so it would be better to not have too many arguments. The pet's name and species seem like things we would want to name when we create a pet; age and color might be specified later, so we'll make name and species be arguments to the constructor. One could certainly justify other ways of doing this. Finally, we need to decide how to represent `name`, `species`, `color` and `age`. The pet's age is certainly a number; the other three can easily be respresented as strings.

All of these decisions lead to the following code for the constructor:

```
def __init__(self, name, species):
    self.name = name
    self.species = species
    self.color = "black"
    self.age = 0
```

As we said, this passes `name` and `species` as arguments and uses default values for the other two variables. Don't be confused by code such as

```
self.name=name
```

The left side of the = sign has the instance varible `self.name`. The right side of the = sign has the value we are assigning this variable, which is the value of the first argument to the constructor. If we construct a specific pet, such as

```
R = Pet( "Rosie", "cat")
```

the instance variable elf.name} will be et to "Rosie".

After the constructor we define any getter and setter for each of the instance variables. This is very easy code. For variable name we don't need need a setter and the getter is a one-line function:

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

for variable color we also have a setter:

```
def getColor(self):
    return self.color

def setColor(self, c):
    self.color = c
```

For variable age we have a getter and setter and also an *increment* method that needs no arguments; it just adds 1 to the pet's age:

```
def getAge(self):
    return self.age

def setAge(self, x):
    self.age = x

def incrementAge(self):
    self.age = self.age + 1
```

Finally, we add a Print() method to format the instance data of an object:

```
def Print(self):
    print("%s is a %s %s."%(self.name, self.color, self.species))
```

Putting all of this together, here is our class definition:

```
class Pet:
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.color = "black"
        self.age = 0

    def getName(self):
        return self.name

    def getSpecies(self):
        return self.species

    def getColor(self):
        return self.color

    def setColor(self, c):
        self.color = c

    def getAge(self):
        return self.age

    def setAge(self, x):
        self.age = x

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

    def Print(self):
        print("%s is a %s %s."
                %(self.name, self.color, self.species))
```

Program 8.2.1: A simple Pet class

Notice how simple and repetitive most of this code is. lt is not difficult to write code for a class once you understand what the class does; this is one of the great attractions of this style of coding.

Here is a small main() program that uses this class defintion:

```
def main():
    R = Pet("Rosie", "cat")
    R.setColor("black")

    N = Pet("Norton", "cat")
    N.setColor("orange")
    N.incrementAge()
    N.incrementAge()

    R.Print()
    N.Print()
main()
```

An application program for the Pet class

Example 2: *Write a class that represents people with names and ages and also models the ability of one person to marry another.* In the preceding section we gave a Person class:

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

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

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

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

The Person class from Program 8.1.1

We need to modify this class to include the idea of marriage. Actions in classes tend to take the form of methods – when we call a method of an object, we tell the object to perform that method on its data. For this class we need a method marry() that tells one object to marry another. In terms of data what does this do? After this method has called and returned, the object still needs to know who it married. This means we need an instance variable spouse. This will need to be initialized in the constructor; the easiest default value is None, which is the value Python uses for objects that are not yet constructed. For this example we won't make use of age, so we will eliminate to keep our program shorter. There would be no problem with keeping it. This makes our constructor

```
def __init__(self, myName):
    self.name = myName
    self.spouse = None
```

We will only let Persons marry other Persons, which means that we have access to the instance variables of the marriage partner. To enable self to marry Person x, we need to set self.spouse=x, but that is not sufficient. We also need to make self be x's spouse: x.spouse=self:

```
def Marry(self, x):
    # A person can only marry another Person
    self.spouse = x
    x.spouse = self
```

It is easy to tell if a Person is married. The following method returns True if the current person is married, and False if that person is single:

```
def IsMarried(self):
    return self.spouse != None
```

If x is married we can always find the name of x's spouse as x.spouse.name, but this is a bit unwieldy. Here is a method that returns the name of self's spouse. Of course there might not be a spouse, so we include an option for this case as well:

```
def SpouseName(self):
    if self.IsMarried():
        return self.spouse.name
    else:
        return "not married"
```

Note how this uses the IsMarried() method: we want to know if self is married, so we call the self.IsMarried() method.

Finally, here is a Print method that makes use of several of the methods we have implemented:

```
def Print(self):
    if self.IsMarried():
        print( "%s is married to %s" %
                 (self.name, self.SpouseName()) )
    else:
        print( self.name )
```

Here is the complete code for our new class Person:

```
class Person:
    def __init__(self, myName):
        self.name = myName
        self.spouse = None

    def Marry(self, x):
        # A person can only marry another Person
        self.spouse = x
        x.spouse = self

    def IsMarried(self):
        return self.spouse != None

    def SpouseName(self):
        if self.IsMarried():
            return self.spouse.name
        else:
            return "not married"

    def Print(self):
        if self.IsMarried():
            print( "%s is married to %s" %
                    (self.name, self.SpouseName()))
        else:
            print( self.name )
```

Program 8.2.2: Person class with marriage

And here is a main() function that makes use of this class

```
def main():
    x = Person( "bob")
    y = Person( "suzie")
    z = Person( "joe")

    y.Marry(z)

    x.Print()
    y.Print()
    z.Print()

main()
```

An application program for the Person class

Example 3: *Write a class that represents names of people and can recognize family names and given names.* Our last example defines a Name class where the instance variables hold the given name (self.given) and the family name (self.family) of each object. The constructor takes a string, such as "Fred Flintstone" and separates it into a string holding the given name or names, and a string holding the family name. Note that there may be more than one given name, as in "Charles Philip Arthur George Windsor". We could represent this as a list of strings, but instead we will put all of the given names into one string and the family name in another. In most situations we want one of three things:

- The family name only
- The first name (the first of the given names) only
- The full name

All of those will be easily available in our representation.

The constructor is mostly a straightforward application of the String class strip() and split() methods. The idea is to split the argument into a list of strings, then pull of the last of these as the family name and concatenate the rest together, separated by spaces, as the given name. If **str** is the input string (such as "Fred Flintstone"), the following code does this:

```
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
```

We need to modify this slightly to account for anomalies: **str** might contain only one name, as in "Madonna", or it might even be the empty string. Single names are easy: we just put whatever we have into the family name, and make the given name be the empty string. An empty **str** is more of a problem. We can split an empty string, but the result is an empty list and our code above fails for an empty list. We fix this by first stripping **str** to eliminate any unnecessary spaces at the front and back, and then testing whether the result is an empty string. If so we assign the empty string to both the given name and family name and exit. If not, we use the code above. Here is the final code for the constructor:

```
    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
```

The remaining methods handle the tree cases we mention above: family name only, given name only, first name only, or full name. Here is our class:

```
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
```

The Name class

Here is a short application program,which just works with the names as names:

```
def main():
    L = []
    L.append( Name("Fred Flintstone") )
    L.append( Name("Barney Rubble") )
    L.append( Name("Wilma Flintstone") )
    L.append( Name("Dino") )
    L.append( Name( "") )
    L.append( Name("Bam Bam Rubble") )
    L.append( Name( "Mr. Slate" ) )

    for x in L:
        print(x.LastName())

main()
```

Program 8.2.3: Application program using the Name class

This will print:

```
Flintstone
Rubble
Flintstone
Dino

Rubble
Slate
```

Here is a more interesting application. For this example we put the class by itself in a file called "MyNameClass.py". In a more authentic situation we would probably call the file "Name.py", but I want to make a distinction between the name of the file and the name of the class. We write a separate application program, making use of the Name class. In this case our application is another Person program; this time the Person class uses Name to store the person's name. Since the Name class is not in the same file as the application, the latter program needs to import it. We use the syntax

```
from MyNameClass import *
```

and then refer directly to the Name class. Alternatively, we could use the following syntax for importing:

```
import MyNameClass
```

and then refer to the names as MyNameClass.Name. For example, the second line of the constructor would be

```
self.name = MyNameClass.Name(myName)
```

Here is the application program:

```
from MyNameClass import *

class Person:
    def __init__(self, myName):
        self.name = 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 main():
    L = []
    x = Person("Donald Ervin Knuth")
    x.SetAge( 77 )
    L.append(x)

    y = Person("Grace Brewster Murray Hopper")
    y.SetAge(109)
    L.append(y)

    z = Person("Alan Mathison Turing")
    z.SetAge(103)

    L.append(z)
    for person in L:
        person.Print()

main()
```

Program 8.2.4: Another application program using the Name class