Classes
Object-Oriented Programming

See Chapter 8 of the Notes

Object-oriented programming was developed in the 1980's as a methodology that would help reduce the number of programming errors.

Here are some of its features. All but the first of these apply to programs we will write this semester.:

- Object-orientation helps programmers organize very large programs.
- Object-orientation provides a discipline for working with data, which helps prevent programs from accidentally changing data.
- Object-orientation allows code written for one program to be imported into another program.
- Object-orientation allows programs to hide implementation details

The most important concept in object-oriented programming is the idea of a *class*. A class is a template for the kinds of data needed to represent some particular type of object.

For example, we might have class Person. Persons have height, hair color, age, names, and phone numbers. In a particular program we would restrict the data for class Person to the types of things the program is concerned with, perhaps just name and phone number.

An *object* is one instance of a class. It is a collection of particular values for the data of the class. For example, if class Person has data values for name and phone number, we might have a particular object that has name "bob" and phone number "775-8386". Another object of this class might have name "Marvin" and phone number "775-8400".

Another important idea of object-oriented programming is that, in addition to data, objects have functions for manipulating that data. These functions are called the *methods* of the class. Just as the class describes the data stored in its objects, the class describes the methods that are available for the objects to work with their data.

If the data of class Person is the name and phone number of the person, the class might have a method that changes the person's phone number. It might have another method for printing out all of its data. Another method might return the value of is phone number (so the program could ask an object for its phone number).

This is important because it helps us localize the parts of a program that can manipulate data elements. It is easiest to write the functions that manipulate structures at the time you create the structures. In addition, if a particular structure can only be modified by its methods and the data in that structure changes unexpectedly, it is easier to find where the change occurred because it had to be done by calling those methods.

Programs store data in variables. The data elements of the class are called *instance variables* because each instance (object) of the class has its own copy of each of the instance variables.

For example, class Person probably has an instance variable *name*. Each object of class Person (i.e., each person) has its own name.

We won't use them a lot, but classes also have a different kind of variable called a *class variable.* Some people call these *static* variables. Unlike instance variables, there is only one copy of a class variable and it is shared by all of the objects of the class.

For example, class Person might have a class variable called *PopulationSize* that keeps track of the number of Person objects. Instance variable *name* changes as you go from person to person, but all persons see the same PopulationSize.

The instance variables, methods, and class variables of a class are called the *properties* of the class.

Object-oriented programming uses a "dot-notation" to refer to the properties of an object.

Suppose class Person has instance variables *name* and *phone_number*, both of which are strings. Suppose it also has a method myPhoneNumber() that returns the value of the phone_number variable. If variable x holds an object of class Person, then x.name and x.phone_number are its two instance variables and x.myPhoneNumber() returns x's phone number.

We could write code such as

```
print( x.name )
```

or

```
x.name = "bob"
```

Here is the most confusing part of Python's terminology for classes.

The code for a class describes the methods and instance variables of the class. This applies to all objects of the class. There needs to be some way to refer to the properties of a particular object within the class. This is the role of the word *self.*

The word "self" always refers to the current object.

For example, suppose class Person now has instance variables *name* and *age*. We might want to have a GetOlder( )method for the class that increases the age by 1. This would apply to any object of the class, whether it is a baby that is one year old or a grandfather that is 81 years old. Here is such a method:

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

This essentially says "Whatever object you are talking about, increase its age by 1."

There might be an object x whose name is "bob" and whose age is 64 and another object y whose name is "Mary" and whose age is 18. If we call x.GetOlder( ), then "self" in the line

```
self.age = self.age + 1
```

refers to object x. Saying x.GetOlder( ) ages only object x, not all of the objects that have been created.

Notice that self is the parameter of the getOlder, method, but no argument is passed to the method for self.

self should be the first parameter of every method. When calling the method we don't give an argument for self.

self's value is the object through which the method is called.
We would call this through an object x of the Person class, with

x.GetOlder( )

Finally, we create objects of a class by using the name of the class as a function. For example, to make objects of class Person we might say

```
x = Person( )
```

Here is a complete program that defines and uses a class Person:

```
class Person:
        def SetName( self, myName ):
                self.name = myName

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

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

def main():
        x = Person()
        x.SetName( "bob")
        x.SetAge(63)
        x.GetOlder()
        print( x.age )   # this prints 64
main()
```

Here is a program:

```
class Person:
        def setAge(a):
                age = a
        def Print():
                print(age)
def main():
        x = Person()
        x.setAge(18)
        x.Print()
main()
```

What will this do?

A) Not run because it doesn't know what a Person is

B) Run but crash because it doesn't say what variable *age* is in setAge()

C) Crash because it doesn't say what *age* is in Print()

D) Run and print 18

What about this one:

```
class Person:
        def setAge(self, a):
                self.age = self.a
        def Print(self):
                print(self.age)
def main():
        x = Person()
        x.setAge(18)
        x.Print()
main()
```

A) Crash because it doesn't know what self.a is in setAge(self, a)
B) Crash because it doesn't know what self.age is in Print(self)
C) It runs and prints 18

One final one:

```
class Person:
        def setAge(self, a):
                self.age = a
        def Print(self):
                print(self.age)
def main():
        x = Person()
        x.setAge(self, 18)
        x.Print()
main()
```

A) Crash because it doesn't know what self is in
   x.setAge(self, 18)
B) Crash because it should have said
   self.setAge(18)
C) Runs and prints 18

Here is what that program should be:

```
class Person:
        def setAge(self, a):
                self.age = a
        def Print(self):
                print(self.age)
def main():
        x = Person()
        x.setAge(18)
        x.Print()
main()
```