DSC 510

WEEK 11 OBJECT ORIENTED PROGRAMMING, MARKDOWN

Object Oriented Programming

Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming you write *classes* that represent real-world things and situations, and you create *objects* based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.

When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modeled with object-oriented programming.

Making an object from a class is called *instantiation*, and you work with *instances* of a class. In this chapter you'll write classes and create instances of those classes. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances. You'll also write classes that extend the functionality of existing classes, so similar classes can share code efficiently. You'll store your classes in modules and import classes written by other programmers into your own program files.

Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

Terminology

What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life.

What Is a Class?

A class is a blueprint or prototype from which objects are created.

What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software.

What is self?

self represents the instance of the class. By using the "*self*" keyword we can access the *attributes* and *methods* of the class in Python.

What is __init__?

"__init__" is a reserved method in Python classes. It is known as a constructor in object oriented concepts. This method called when an object is created from the class and it allow the class to initialize the attributes of a class.

Classes Explained

As an example let's imagine that we create a class called car. Every car has similar characteristics;
doors, windshield, engine, transmission and operations that are typical of any car; accelerate, stop, turn, reverse etc…

When we declare a class we define attributes and methods common to all cars. These are the blueprint of a car. When we create an object, we are creating an instance of a class (instantiating a class).

Using the car example we might instantiate an instance of different cars, one might be a convertible and have 2 doors the other might be a hard top with 4 doors. When we create an object we get the basic attributes of the class as well as the methods. Regardless of whether we create a convertible object or a hardtop object they'll both have the capability to accelerate, stop, turn, reverse, etc… and they'll both have doors, windshield, engine, and transmission.

The power of classes is to define attributes and methods that we can use over and over again throughout our programs. Each instance of the class (object) will obtain the attributes and methods defined by the class without needing to rewrite that functionality each time.

Dog Class

```
1 class Dog():
2     """A simple attempt to model a dog."""

3     def __init__(self, name, age):
          """Initialize name and age attributes."""
4         self.name = name
          self.age = age

5         def sit(self):
              """Simulate a dog sitting in response to a command."""”
              print(self.name.title() + " is now sitting.")

          def roll_over(self):
              """Simulate rolling over in response to a command."""”
              print(self.name.title() + " rolled over!")
```

Dog Class Explained

Line 1 defines the class using the class keyword. Class names should start with a capital letter
Line 2 uses a Python document string (docstring) to explain the purpose of the class.
Line 3 defines the `__init__()` method. The `__init__()` method is a special method Python runs automatically whenever we create a new instance based on the Dog class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names.

We define the `__init__()` method to have three parameters: self, name, and age. The self parameter is required in the method definition, and it must come first before the other parameters. It must be included in the definition because when Python calls this `__init__()` method later (to create an instance of Dog), the method call will automatically pass the self argument. Every method call associated with a class automatically passes self, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of Dog, Python will call the `__init__()` method from the Dog class. We'll pass Dog() a name and an age as arguments; self is passed automatically, so we don't need to pass it. Whenever we want to make an instance from the Dog class, we'll provide values for only the last two parameters, name and age.

Dog Class Explained Continued…

Line 4 The two variables defined at **i**n line 4 each have the prefix self. Any variable prefixed with self is available to every method in the class, and we'll also be able to access these variables through any instance created from the class. `self.name = name` takes the value stored in the parameter name and stores it in the variable name, which is then attached to the instance being created. The same process happens with `self.age = age`. Variables that are accessible through instances like this are called *attributes*.

Line 5 The Dog class has two other methods defined: `sit()` and `roll_over()` Because these methods don't need additional information like a name or age, we just define them to have one parameter, self. The instances we create later will have access to these methods. In other words, they'll be able to sit and roll over. For now, `sit()` and `roll_over()` don't do much. They simply print a message saying the dog is sitting or rolling over. But the concept can be extended to realistic situations: if this class were part of an actual computer game, these methods would contain code to make an animated dog sit and roll over. If this class was written to control a robot, these methods would direct movements that cause a dog robot to sit and roll over.

Creating a Dog Object

```
1 my_dog = Dog('willie', 6)
2 print("My dog's name is " + my_dog.name.title() + ".")
3 print("My dog is " + str(my_dog.age) + " years old.")
```

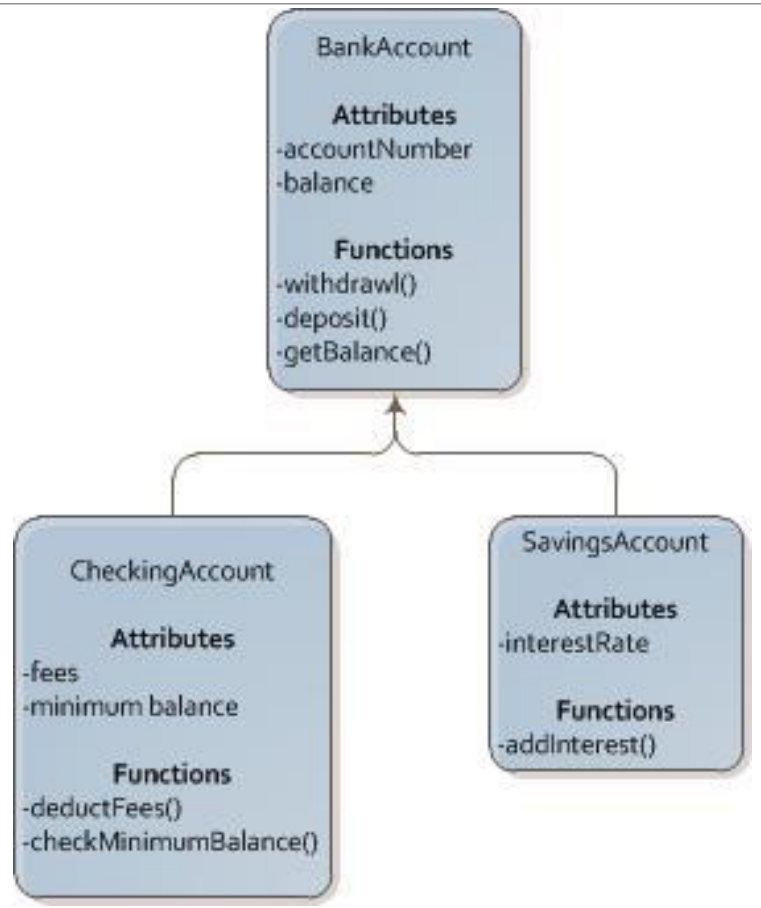
In **line 1**, we define a my_dog object. My_dog is of type dog and has the name willie and the age 6.
In **line 2**, we use dot notation to access the attributes of our dog object. (my_dog.name). In this same line we also call the title() function to capitalize the first letter of our dog's name.
In **line 3**, we again use dot notation to access our dog's age (my_dog.age). Notice that we've seen dot notation before in many of our previous examples such as when we wrote `dictionary.items()` to obtain the items in a dictionary.

If we would like to call our class methods (sit and roll_over) we again use dot notation.

```
my_dog.sit()
my_dog.roll_over()
```

Inheritance

This diagram demonstrates inheritance. In the example we have a parent class called BankAccount. BankAccount classes all have two attributes and two functions. From BankAccount we can derive a CheckingAccount and a SavingsAccount. Each CheckingAccount and SavingsAccount automatically obtain the attributes of their parent class (BankAccount) but also define attributes and methods specific to their class.



Class Inheritance

The syntax for defining a child class is as follows:

```
class ChildClassName(ParentClassName):
    classAttributes

    classMethods()
```

Inheritance Example (parent class)

```
#Definition of parent class Person

class Person:
     def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self): #Name function returns the firstName and lastName.
        return self.firstname + " " + self.lastname
```

Inheritance Example Continued (child class)

```
#Definition of child class called Employee.  Employee is a child of the Person
#class and derives behavior from the Person class.

class Employee(Person): #Notice the syntax here, for declaring a child class.
     def __init__(self, first, last, employeeNum):#when we declare an employee object
                                                   #we are instantiating a Employee
                                                   #class as well as a Person class.
         Person.__init__(self,first, last)
         self.employeeNum = employeeNum

     def GetEmployee(self): #GetEmployee returns the Name of the employee as defined
                            #by the parent class and the employeeNum as defined
                            #by the Employee class
         return self.Name() + ", " +  self.employeeNum
```

Inheritance Example Continued (implementation)

```
#Create an object of type Person
examplePerson = Person("Mark", "Zuckerberg")
#Create an object of type Employee
exampleEmployee = Employee("Bill", "Gates", "1983")

#Call the name function from the person class
print(examplePerson.Name())
#Call the getEmployee function from the employee class
print(exampleEmployee.GetEmployee())
```

Overriding Classes

You can override any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

Overriding Method Example

In this example we have a class SavingsAccount which has a method (getRate) to obtain the interest rate of 2. Objects of type SavingsAccount will have a 2% interest rate.

By creating a child class called MoneyMarket we obtain the functionality of the parent class including the getRate() method. Overriding the functionality of this method in the child class we can set the interest rate to a different value in the child.

In this case an object of type SavingsAccount would have an interest rate of 2 and a MoneyMarket would have an interest rate of 3.

```
class SavingsAccount:
    def __init__(self):
        self.interestRate = 2
    def getRate(self):
        return self.interestRate

class MoneyMarket(SavingsAccount):
    def getRate(self):
        return self.interestRate + 1

savings = SavingsAccount()
market = MoneyMarket()

print(savings.getRate())
print(market.getRate())
```

Getting started with Markdown

The nice part about generating a report in Markdown is that you can do a number of things with it. For example, you can export that to HTML, PDF, or any sort of file you need. It is a tremendously useful skill to have some exposure to.