PGR107
Python Programming

Lecture 9 – Objects & Classes

Chapter 9
Objects and Classes

Chapter Goals

- To understand the concepts of classes, objects, and encapsulation
- To implement instance variables, methods, and constructors
- To be able to design, implement, and test your own classes
- To understand the behavior of object references

Object Oriented Programming

- Python is an object oriented programming language.

- Almost everything in Python is an object, with its properties and methods.

- A Class is like an object constructor, or a "blueprint" for creating objects.

Object Oriented Programming

- Python supports many different kinds of data

  777 3.2345 “Python” [1, 2, 3, 4, 5]

  {“John” : 34567865, “Sara” : 23456435 }

- Each is an object and every object has:
  - a type
  - an internal data representation
  - a set of methods for interaction with the object

- An object is an instance of a type
  - 777 is an instance of an int
  - “Python” is an instance of a string

Object Oriented Programming

- **Create a class:** To create a class, use the keyword **class**.

```
class MyClass:
  x = 5
```

- **Create Object**: Now we can use the class named **MyClass** to create objects.

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

- The example in the previous slide is class and object in its simplest form, and is not really useful in real life applications.

- To understand the meaning of classes we have to understand the built-in **__init__() function**.

- All classes have a function called **__init__()**, which is always executed when the class is being initiated.

- Use the **__init__() function** to assign values to object properties, or other operations that are necessary to do when the object is being created

The __init__() Function

- The **__init__ function** is called automatically every time the class is being used to create a new object.

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

```
p1 = Person("John", 36)
```

```
print(p1.name)
print(p1.age)
```

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.

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

  def myfunc(self):
    print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)
p1.myfunc()
```

The **self parameter** is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class.

Modify Object Properties

- You can modify properties on objects as follows:

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

p1 = Person("John", 36)
```

```
print(p1.name)
print(p1.age)
```

```
p1.age = 40
```

Defining Your Own Print Method

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
c = Coordinate (3, 4)

print(c.x)
print(c.y)
```

```
print(c)
```

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"
```

```
c = Coordinate (3, 4)

print(c)
```

Other Special Methods

- like **print**, can override other operators (+, -, ==, <, >, len (), and many others) to work with your class.
- define them with double underscores before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print self
```

- … and many others

__add__() Method

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"

    def __add__(self, other):
        first = self.x + other.x
        second = self.y + other.y
        return Coordinate(first, second)
```

```
a = Coordinate (3, 4)
b = Coordinate (4, 5)

c = a + b

print(c)
```

Public vs. Private Members

- All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self.salary=sal
```

```
e1 = employee("James", 30000)
```

```
print(e1.name)
print(e1.salary)
```

Public vs. Private Members

- Python's convention to make an instance variable **protected** is to add a prefix _ (single underscore) to it.
- In fact, this doesn't prevent instance variables from accessing or modifying the instance.

```
class employee:
    def __init__(self, name, sal):
        self._name=name  # protected attribute
        self._salary=sal # protected attribute
```

```
e1 = employee("James", 30000)
```

```
print(e1._name)
print(e1._salary)
```

Public vs. Private Members

- A double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an **AttributeError**:

```
class employee:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        self.__salary=sal # private attribute
```

```
e1 = employee("James", 30000)
```

```
print(e1.__name)
print(e1.__salary)
```

```
AttributeError: 'employee' object has no attribute '__name'
```

Public vs. Private Members

- **How to access private attributes:** You can access private members of a class using public methods of the same class.

```
class employee:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        self.__salary=sal # private attribute

    def get_info(self):
        print(self.__name)
        print(self.__salary)
```

```
e1 = employee("James", 30000)
```

```
e1.get_info()
```

End of Chapter 9