Jawaharlal Nehru Technological University Hyderabad
S C D E
Kukatpally, Hyderabad - 500 085, Telangana, India

Object Oriented Programming using Python part 2

Session 13 , 30 May 23

Dr N V Ganapathi Raju
Professor and HOD of IT
Gokaraju Rangaraju Institute of Eng and Tech

Public , Private and Protected members

- Object-oriented languages, like C++ and Java, use various keywords to control and restrict the resource usage of a class, using keywords like **public, private and protected** .
- Python has a different way of providing the functionality of these access modifiers.

Public Keyword

- public members of a class are available to everyone.
- So they can be accessed from outside the class and also by other classes too.

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

e1=employee("Sachin",10000)
print(e1.name)
print(e1.salary)

print()

e1.salary=20000
print(e1.salary)
print(e1.name)
```

Sachin
10000

20000
Sachin

Note:

- **All members of a class are by default public in Python.**
- **These members can be accessed outside of the class, and their values can be modified too.**

Protected Keyword

- protected members of a class can be accessed by other members within the class and are also available to their subclasses.
- No other entity can access these members.
- In order to do so, they can inherit the parent class.
- Python has a unique convention to make a member protected: Add a prefix _ (single underscore).
- This prevents its usage by outside entities unless it is a subclass.

Protected Keyword

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

e1=employee("Sachin", 10000)
print(e1._name)
print(e1._salary)

print()

e1._salary=20000
e1._name="Sourab"
print(e1._name)
print(e1._salary)
```

```
Sachin
10000

Sourab
20000
```

Private Keyword

- The private members of a class are only accessible within the class.
- In Python, a private member can be defined by using a prefix __ (double underscore).
- Every member with a double underscore will be changed to

  _object._class__variable.

Private Keyword

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

#e1=employee("Sachin",10000)
#print(e1.__salary)

e1=employee("Sachin",10000)
print(e1._employee__salary)
print(e1._employee__name)

print()

e1._employee__salary=20000
e1._employee__name="Sourab"
print(e1._employee__salary)
print(e1._employee__name)
```

```
10000
Sachin

20000
Sourab
```

Accessing attributes using built-in functions

Instead of using the normal statements to access attributes, you can use the following functions −

- The getattr(obj, name[, default]) − to access the attribute of object.
- The hasattr(obj,name) − to check if an attribute exists or not.
- The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.
- The delattr(obj, name) − to delete an attribute.

Accessing attributes

```
class Employee:
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Sachin", 10000)
emp2 = Employee("Sourab", 20000)

emp1.displayEmployee()   # Name :  Sachin , Salary:  10000
emp2.displayEmployee()   # Name :  Sourab , Salary:  20000
```

```
print(hasattr(emp1, 'salary'))  # True
print(hasattr(emp2, 'salary'))  # True

print(getattr(emp1, 'salary'))  # returns 10000
print(getattr(emp2, 'salary'))  # returns 20000

setattr(emp1, 'salary', 5000) # sets salary as 5000 for emp1
setattr(emp2, 'salary', 6000) # sets salary as 6000 for emp2

print(getattr(emp1, 'salary')) # returns 5000
print(getattr(emp2, 'salary')) # returns 6000

print ("Total Employee %d" % Employee.empCount) #Total
Employee 2

delattr(emp1, 'salary')  # deletes attribute salary
#delattr(emp2, 'salary')

#print(getattr(emp1, 'salary')) # raises error as AttributeError’
```

Built-In Class Attributes

```
class Employee:

   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Sachin", 10000)
emp2 = Employee("Sourab", 20000)
```

```
print ("Employee.__doc__:", Employee.__doc__)

print ("Employee.__name__:", Employee.__name__)

print ("Employee.__module__:", Employee.__module__)

print ("Employee.__bases__:", Employee.__bases__)

print ("Employee.__dict__:", Employee.__dict__ )
```

```
Employee.__doc__: None
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', 'empCount': 2, '__init__':
<function Employee.__init__ at 0x000000B0166B65E8>, 'displayCount': <function
Employee.displayCount at 0x000000B0166B6AF8>, 'displayEmployee': <function
Employee.displayEmployee at 0x000000B0166B6A68>, '__dict__': <attribute
'__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of
'Employee' objects>, '__doc__': None}
```

Destroying Objects (Garbage Collection)

- Python **deletes** unneeded objects (built-in types or class instances) automatically to free the memory space.
- The process by which Python periodically reclaims blocks of memory that no longer are in use is termed as **Garbage Collection.**
- A class can implement the special method ***__del__()***, called a **destructor**, that is invoked when the instance is about to be destroyed.
- This method might be used to clean up any **non-memory resources used by an instance**.

Destructor

```
class Addition:

    def __init__(self, a, b):
        self.a = a
        self.b = b
        print ("In Constrictor")

    def add(self):
        print(self.a + self.b)

    def __del__(self):
      class_name = self.__class__.__name__
      print (class_name, "destroyed")

add_obj = Addition(3,4)

add_obj.add()

del add_obj
```

```
In Constrictor
7
Addition destroyed
```

Python Bank Account class

```
class Account:
    def __init__(self, holder, number, balance,credit_line=1500):
        self.Holder = holder
        self.Number = number
        self.Balance = balance
        self.CreditLine = credit_line

    def balance(self):
        return self.Balance

    def transfer(self, target, amount):
        if(self.Balance - amount < -self.CreditLine):
            # coverage insufficient
            return False
        else:
            self.Balance -= amount
            target.Balance += amount
            return True
```

```
acc = Account("ABC",100,10000)

acc2 = Account("XYZ",101,20000)

acc.transfer(acc2,1000)
print(acc.balance())

print(acc2.balance())
```

9000
21000

Problem Statement: Shopping cart Application

Write a class, Item that represents an item for sale. It should have the following:

- Fields representing the name and price of the item • A constructor that sets those fields,
- A __str__() method that returns a string containing the item name and price, with the price formatted to exactly 2 decimal places

Test the class by creating a new item object and printing it out.

Write a class, ShoppingCart that might be used in an online store. It should have the following:

- A list of Item objects that represents the items in the shopping cart
- A constructor that creates an empty list of items (the constructor should take no arguments except self)
- A method called add() that takes a name and a price and adds an Item object with that name and price to the shopping cart
- A method called total() that takes no arguments and returns the total cost of the items in the cart .
- A method called remove_items() that takes an item name (a string) and removes any Item objects with that name from the shopping cart. It shouldn't return anything.

- Then test out the shopping cart as follows: (1) create a shopping cart; (2) add several items to it; (3) print the cart's total cost (using the total() method); (4) remove one of the items types; (5) print out the cart

Shopping cart Application

Step 1

```
class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __str__(self):
        return '{:s}, {:.2f}'.format(self.name, self.price)

tem = Item('Item1', 12.40)
#print(item)
```

Step 2

```
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

    def total(self):
        return sum(item.price for item in self.items)

    def remove_items(self, name):
        self.items = [item for item in self.items if item.name != name]

    def __str__(self):
        return '\n'.join(str(item) for item in self.items)
```

Step 3

```
cart = ShoppingCart()
cart.add(Item('Item1', 150.55))
cart.add(Item('Item2', 200.75))
cart.add(Item('Item3', 100.00))
print(cart)
cart.remove_items('Item1')
print(cart.total())
print(cart)
```

```
Item1, 150.55
Item2, 200.75
Item3, 100.00

300.75
Item2, 200.75
Item3, 100.00
```