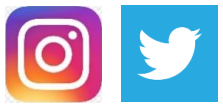
Computational Thinking with Programming

Lecture - 22

Object Oriented in Python

BENNETT UNIVERSITY
TIMES OF INDIA GROUP

@cse_bennett @csebennett



Today…

- Last Session:
  - Introduction to OOPS.
- Today's Session:
  - Constructor.
  - Polymorphism.
- Hands on Session with Jupyter Notebook:
  - We will practice on the objects programming in Jupyter Notebook.

Objects

- Python is an *object-oriented* programming language

- An *object* is a combination of variables (also called *attributes* or *instance variables* or *object variables*) and behaviors (i.e., functions, which are referred to as *methods* in the object context)

- To create an object, you need to create a *class* using the keyword *class* as follows:

```
class Student:
    sname = "Mohammad"
```

An attribute; you can have as many attributes as you want

Creating Objects Out of Classes

- After defining a class, you can create any number of objects out of it

```
s1 = Student()
print(s1.sname)
s2 = Student ()
print(s2.sname)
s3 = Student ()
print(s3.sname)
```

- But, all of the above students have the same name!
  - How can we have student objects with different names?

Class Constructor

- All classes in Python have a function called *__init__()*, which is always executed when the class is being *initiated* (i.e., an object out of it is *created*)

- You can use the __init__() function to assign values to object attribute(s)– as a matter of fact, you can add any code in the __init__() function that you may find necessary for creating objects out of your class

```
class Student:
    def __init__(self, sn):
        self.sname = sn
```

*The __init__() function is called the **constructor** or the **initializer***

Class Constructor

- All classes in Python have a function called *__init__()*, which is always executed when the class is being *initiated* (i.e., an object out of it is *created*)

- You can use the __init__() function to assign values to object attribute(s)– as a matter of fact, you can add any code in the __init__() function that you may find necessary for creating objects out of your class

```
class Student:
    def __init__(self, sn):
        self.sname = sn
```

The constuctor should always have the keyword self as its first parameter

Creating Objects Out of Classes

- You can now create as many students as you like with *different* names

```
s1 = Student12("Khaled")
print(s1.sname)
s2 = Student12("Eman")
print(s2.sname)
s3 = Student12("Omar")
print(s3.sname)
```

- Is there any other way for assigning different names to different students?

Methods

- You can assign different values to attributes through functions/methods, which you can define in your classes

```
class Student:
    sname = ""
    def setSName(self, sn):
        self.sname = sn

s1 = Student12()
s1.setSName("Omar")
print(s1.sname)
```

Every method in a Python class should have self as its first parameter

Methods

- You can assign different values to attributes through functions/methods, which you can define in your classes

```
class Student:
    sname = ""
    def setSName(self, sn):
        self.sname = sn

s1 = Student12()
s1.setSName("Omar")
print(s1.sname)
```

Every attribute in a Python class should be always prefaced with *self.* upon accessing it

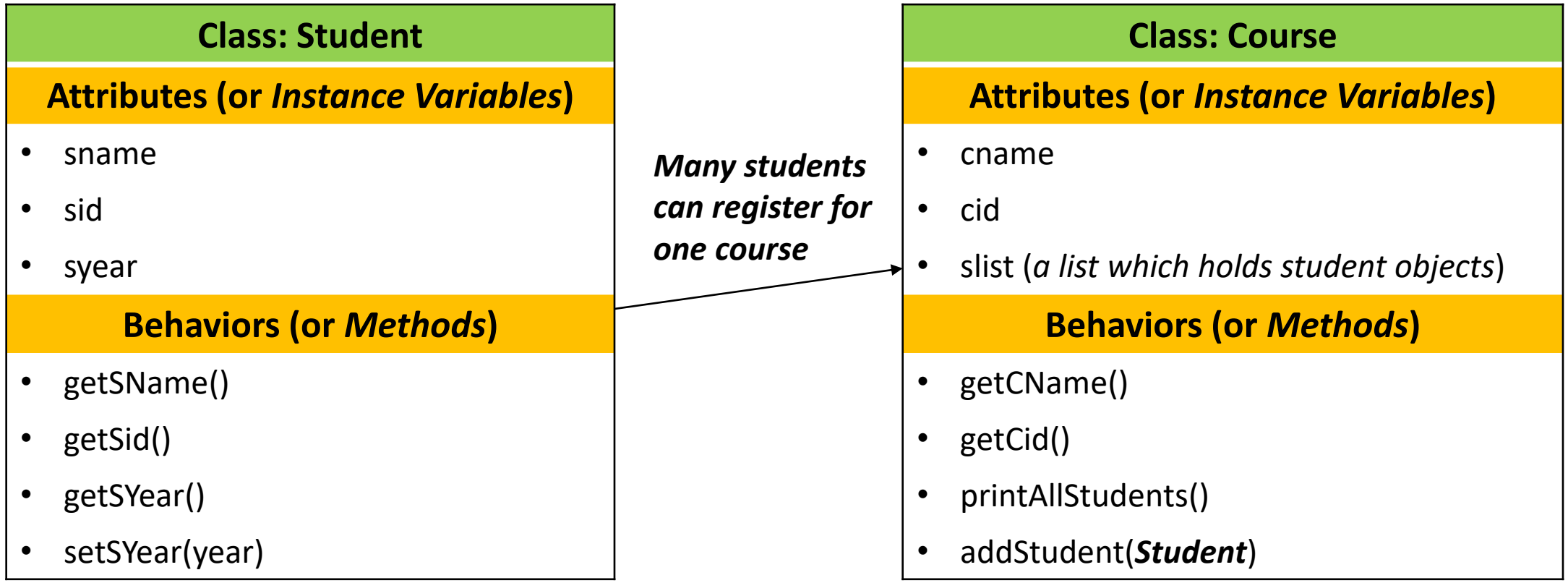
Constructor and Methods

- You can also define __init__() alongside any other function/method

```
class Student12:
    def __init__(self, sn):
        self.sname = sn

    def setSName(self, sn):
        self.sname = sn

    def getSName(self):
        return self.sname

s1 = Student12("Eman")
print(s1.getSName())
s1.setSName("Eman2")
print(s1.getSName())
```

Example: Students and Courses

- Let us write an *object-based* program for the following two classes



Example: Students and Courses

```
class Student:
    def __init__(self, sn, sid, sy):
        self.sname = sn
        self.sid = sid
        self.syear = sy

    def getSName(self):
        return self.sname

    def getSid(self):
        return self.sid
```

Example: Students and Courses

```
    def getSYear(self):
        return self.syear

    def setSYear(self, sy):
        if type(sy) is str and (sy.lower() == "freshman" or sy.lower() ==
"sophomore" or sy.lower() == "junior" or sy.lower() == "senior"):
            self.syear = sy
        else:
            print("You have input an invalid value! The only allowable input are
freshman, sophomore, junior, and senior")
```

Example: Students and Courses

```
class Course:
    def __init__(self, cn, cid, sl):
        self.cname = cn
        self.cid = cid
        self.slist = sl

    def getCName(self):
        return self.cname

    def getCid(self):
        return self.cid
```

Example: Students and Courses

```
def printAllStudents(self):
    for i in self.slist:
        print(i.getSName(), i.getSid(), i.getSYear())

def addStudent(self, st):
    if type(st) is Student:
        for i in self.slist:
            if i is st:
                print("This student is already added to the course!")
                return
        self.slist.append(st)
    else:
        print("Sorry, this is not a student!")
```

Example: Students and Courses

```
c1 = Course("Prinicples of Computing", 15110, [])

s1 = Student("Eman", 100, "Freshman")
s2 = Student("Omar", 101, "Freshman")
s3 = Student("Khaled", 102, "Junior")

c1.addStudent(s1)
c1.addStudent(s2)
c1.addStudent(s3)

c1.printAllStudents()
```

Function Based way Vs oops Way of Writing the code

```
def abc():
    name = input("Enter name")
    roll_number = input("Enter Roll_number")
    age = input("Enter an age")


abc()
```

By function

Task: Write a Program to create a information related to student such as age, roll number and name by oops as well as by the function.

Function Based way Vs oops Way of Writing the code

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

    def get_data(self,name,age,roll_no):
        self.name = name
        self.age =age
        self.roll_no = roll_no

    def display_data(self):
        print(self.name)
        print(self.age)
        print(self.roll_no)
```

```
student1 = student("Anurag", 21, 195)
student1.display_data()
student1.get_data("mohit", 22, 147)
student1.display_data()
```

By oops

```
Anurag
21
195

mohit
22
147
```

Solution

```
class Rectangle():
    def __init__(self, l, w):
        self.length = l
        self.width  = w

    def rectangle_area(self):
        return self.length*self.width

newRectangle = Rectangle(12, 10)
print(newRectangle.rectangle_area())
```

Task: Write a Python class named Rectangle constructed by a length and width and a method which will compute the area of a rectangle.

Encapsulation

- We said that encapsulation:
- hid details of the implementation so that the program was easier to read and write
- modularity, make an object so that it can be reused in other contexts
- providing an interface (the methods) that are the approved way to deal with the class

class namespaces are dicts

- the namespaces in every object and module is indeed a dictionary
- that dictionary is bound to the special variable `__dict__`
- it lists all the local attributes (variables, functions) in the object

private variables in an instance

- many OOP approaches allow you to make a variable or function in an instance ***private***
- private means not accessible by the class user, only the class developer.
- there are advantages to controlling who can access the instance values

Privacy in Python

- Python takes the approach “We are all adults here”. No hard restrictions.
- Provides naming to avoid accidents. Use __ (double underlines) in front of any variable
- this ***mangles*** the name to include the class, namely __var becomes _class__var
- still fully accessible, and the __dict__ makes it obvious

privacy example

```
class NewClass (object):
    def __init__(self, attribute='default', name='Instance'):
        self.name = name                    # public attribute
        self.__attribute = attribute        # a "private" attribute
    def __str__(self):
        return '{} has attribute {}'.format(self.name, self.__attribute)
```

```
>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1)
Monty has attribute Python
>>> print(inst1.name)
Monty
>>> print(inst1.__attribute)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'newClass' object has no attribute '__attribute'
>>> dir(inst1)
'_NewClass__attribute', '__class__', ... , 'name']

>>> print(inst1._NewClass__attribute)
Python
```

Thank You

?