Prof R. Madana Mohana

OBJECT ORIENTED PROGRAMMING USING PYTHON

CLASSES & OBJECTS

https://www.youtube.com/c/RASINENIMADANAMOHANA

OUTLINE

- Introduction
- Classes and Objects
- init method
- Constructors and Destructors

Classes and Objects: Introduction

- In all our programs till now, we have been using the procedure-oriented technique in which our programs written using functions or blocks of statements which manipulate data.
- However, another a better style of programming is called **object-oriented programming** in which data and functions are combined to form a **class**.
- Compared with other programming languages, **Python** has a very short and simple way to define and use **classes**.

Classes and Objects: Introduction

- The **class** mechanism supported by **Python** is actually a mixture of that found in **C++** and **Modula-3**.
- **Python** supports all the standard features of **Object-Oriented Programming**.

CLASSES AND OBJECTS

- **Classes** and **objects** are the two main aspects of **object oriented programming**.
- In fact, a **class** is the basic building block of **Python**.
- A **class** creates a new type and object is an instance(or variable) of the **class**.
- **Classes** provides a blueprint or a template using which objects are created.
- In fact, in **Python**, *everything is an object or an instance of some **class**.*

CLASSES AND OBJECTS

- For example, all integer variables that we define in our programs are actually instances of class **int**.
- Similarly, all string variables are objects of class **string**.
- We can find out the type of any object using the **type()** function.
- The **Python Standard Library** is based on the concept of classes and objects.

CLASSES AND OBJECTS

1.Defining Classes:

- A **class** is a user-defined blueprint or prototype from which objects are created.
- **Classes** provide a means of bundling data and functionality together.
- Creating a new class creates a new type of object.
- **Classes** are created by keyword **class**
- Attributes are the variables that belong to **class**.
- Attributes are always **public** and can be accessed using **dot (.)** operator.

```
Eg.: Myclass.Myattribute
```

CLASSES AND OBJECTS

1.Defining Classes:

A collection of variables (data members) and functions (methods) is called a **class**.

Syntax:

```
class class_name:
    list of data members or class variables
    list of methods or class methods
```

CLASSES AND OBJECTS

1.Defining Classes:

Example-1:

```
# To create a class with only variables
class test:
    a=10
    b=20
print(test.a+test.b)
```

OUTPUT:

```
30
```

CLASSES AND OBJECTS

1.Defining Classes:

Example-2:

```
# To create a class with only methods
class wish:
    def mng():
        print('good morning')
    def aft():
        print('good afternoon')
    def evng():
        print('good evening')
    def ngt():
        print('good night')
```

CLASSES AND OBJECTS

1.Defining Classes:

Example-2:

```
wish.mng()
wish.aft()
wish.evng()
wish.ngt()
```

OUTPUT:

```
good morning
good afternoon
good evening
good night
```

CLASSES AND OBJECTS

2. Creating Objects:

- Once a **class** is defined, next job is to create an object (or instances) of that class.
- The object can then access class variables and class methods using the dot operator(**.**).

The syntax to create an object is given as:

```
obect_name = class_name()
```

The syntax for accessing a class member through the class object is:

```
obect_name.class_member_name
```

CLASSES AND OBJECTS

2. Creating Objects:

Example: Program to access class variable using class object

```
class test:
    var = 10 # class variable
obj = test()
print(obj.var) # class variable is accessed
using class object
```

OUTPUT:

```
10
```

CLASSES AND OBJECTS

3. Data Abstraction and Hiding through Classes:

- **Data abstraction** refers to the process by which data and functions are defined in such a way that only essential details are provided to the outside world and the implementation details are **hidden**.
- In **Python** and other object oriented programming languages, **classes** provide methods to the outside world to provide the functionality of the object or to manipulate the object's data.

CLASSES AND OBJECTS

3. Data Abstraction and Hiding through Classes:

- Any entity outside the world does not know about the implementation details of the **class** or **class method**.
- **Data Encapsulation**, also called **Data Hiding**, organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object.

CLASSES AND OBJECTS

3. Data Abstraction and Hiding through Classes:

- **Encapsulation** defines different access levels for data variables and member functions of the **class**. These access levels specifies the access rights.

Public access level:

- In **python**, by default, every member of class is **public**.
- Public members are available publicly. Means can be accessed from any where.
- These attributes we can access from any where either inside the class or from outside of the class.

CLASSES AND OBJECTS

3. Data Abstraction and Hiding through Classes:

Private access level:

- Private attributes can be accessed only within the class i.e. private attributes cannot access outside of the class.
- By convention, we can declare member as **private** type, we write **double underscore**(__) symbol before the identifiers.

CLASSES AND OBJECTS

3. Data Abstraction and Hiding through Classes:

Protected access level:

- Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes.
- By convention, we can declare member as **protected** type, we write **single underscore**(_) symbol before the identifiers.

These attributes should only be used under certain conditions.

The __init()__ method (The Class Constructor)

- The **__init()__** function or method is a built-in function. and has a special significance in **Python classes**.
- The **__init()__** method is automatically executed when an object of a class is created.
- No need of explicit calling.
- The main purpose the **__init()__** method is to initialize the variables of class object.
- The **__init()__** method is prefixed as well as suffixed by **double underscore (__)**.

The __init()__ method (The Class Constructor)

Example-1: Program illustrating the use of __init()__ method

```
class test:
    def __init__(self,val):
        print("In class method!")
        self.val = val
        print("The value is:", val)
obj = test(20)
```

OUTPUT:

```
In class method!
The value is: 20
```

The __init()__ method (The Class Constructor)

Example-2: Program illustrating the use of __init()__ method

```
class person:
    def __init__(self,name):
        print("In class method!")
        self.name = name
    def display(self):
        print("Hello", self.name)
        return
p1 = person('Ram')
p1.display()
```

The __init()__ method (The Class Constructor)

Example-2: Program illustrating the use of __init()__ method

OUTPUT:

```
In class method!
The value is: 20
```

Constructors

- A **constructor** is a class function that instantiates an object to predefined values.
- A **constructor** is a special method. It begins with a **double underscore** (_ _).
- In **Python** constructors are implemented by using the **__init__()**
- The **__init__()** method is called the constructor and is always called when creating an object.

Constructors

Constructors are 2 types:

1. Default constructor
2. Parameterized constructor

Default constructor:

To create an object without parameters is called a **default constructor**.

Constructors

Default constructor: Example-1

```
# Program illustrating Default constructor
class test:
    def __init__(self):
        print('I am test class init
function..')
t1=test()
```

OUTPUT:

I am test class init function..

Constructors

Default constructor: Example-2

```
# Write a program to count no. of objects created for a class
class test:
    count = 0
    def __init__(self):
        test.count = test.count + 1
    def display():
        print('No. of objects = ',test.count)
t1=test()
t2=test()
t3=test()
t4=test()
t5=test()
t6=test()
test.display()
```

Constructors

Default constructor: Example-2

 # Write a program to count no. of objects created for a class

OUTPUT:

```
No. of objects =  6
```

Constructors

Parameterized constructor:

To create an object with parameters is called a **parameterized constructor**.

Constructors

Parameterized constructor: Example-1

```
# Write a program to illustrate parameterized constructor
class users:
    name = ""
    def __init__(self, name):
        self.name = name
    def sayHello(self):
        print("Welcome to Python Developers, " +
self.name)
user1 = users("RAM")
user1.sayHello()
```

Constructors

Parameterized constructor: Example-1

 # Write a program to illustrate parameterized constructor

OUTPUT:

```
Welcome to Python Developers, RAM
```

Constructors

Parameterized constructor: Example-2

```
class car:
    def __init__(self,x,y,z):
        self.color = x
        self.companyname = y
        self.model = z
    def display(self):
        print('Car color: ',self.color)
        print('Car company name: ',self.companyname)
        print('Car model name: ',self.model)
c1 = car('Red','KIA','SELTOS')
c1.display()
```

Constructors

Parameterized constructor: Example-2

OUTPUT:

```
Car color:  Red
Car company name:  KIA
Car model name:  SELTOS
```

Destructors

- **Destructors** are called when an object gets destroyed.
- In **Python**, destructors are not needed as much needed, in **C++** or **Java** , because **Python** has a garbage collector that handles memory management automatically.
- The **__del__()** method is a known as a **destructor** method in **Python**.

Destructors

Syntax of destructor declaration:

```
def __del__(self):
    body of destructor
```

Destructors

```
# Program to demonstrate destructor
class employee:
    def __init__(self):
        print('Employee created..')
    def __del__(self):
        print('Destructor called,Employee  deleted..')
e1 = employee()
del e1
```

OUTPUT:

```
Employee created..
Destructor called,Employee  deleted..
```