MODULE – 4

4.1 CLASSES AND OBJECTS

Python Python is an interpreted, object-oriented, high-level programming language with dynamic semantics Class is a user-defined data type which binds data and functions together into single entity. Class is just a prototype (or a logical entity/blue print) which will not consume any memory. An object is an instance of a class and it has physical existence. One can create any number of objects for a class. A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).

4.1.1 Programmer-defined Types

A class in Python can be created using a keyword class. The syntax for creating class is shown below. The keyword pass can be used when you don't need any class members within it.

```
class Point:
    pass
print(Point)
```

or

Defining a class named Point creates a **class object**.

```
>>> Point
```

The output would be –

```
<class '__main__.Point'>
```

The term __main__ indicates that the class Point is at the top level while executing the program.

Now, you can create any number of objects of this class.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0x003C1BF0>
```

Blank assigns the reference to a Point object. Creating a new object is called **instantiation**, and the object is an **instance** of the class. When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

4.1.2 Attributes

An object can contain named elements known as ***attributes***. One can assign values to these attributes(instance) using dot operator. The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number. The syntax for two attributes x and y for the object blank of a class Point is as below

```
blank.x = 10.0
blank.y  = 20.0
```

A state diagram that shows an object and its attributes is called as ***object diagram.*** For the object blank, the object diagram is shown in Figure 4.1.

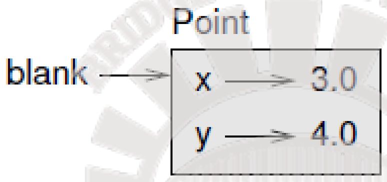


Fig 4.1 Object diagram

The diagram indicates that a variable (i.e. object) p refers to a Point object, which contains two attributes. Each attributes refers to a floating point number.

One can access attributes of an object as shown –

```
>>> print(blank.x)
      10.0
>>> print(blank.y)
      20.0
```

Here, p.x means *"Go to the object blank refers to and get the value of x"*. Attributes of an object can be assigned to other variables –

```
>>> x= blank.x
>>> print(x)
      10.0
```

Here, the variable x is nothing to do with attribute x. There will not be any name conflict between normal program variable and attributes of an object.

Program: Write a class Point representing a point on coordinate system. Implement following functions (Programs written in IDLE)

- A function read_point() to receive x and y attributes of a Point object as user input
- A function distance() which takes two objects of Point class as arguments and computes the Euclidean distance between them.
- A function print_point() to display one point in the form of ordered-pair.

```
import math
class Point:
    """ This is a class Point representing a
     coordinate Point
    """

def  read_point(p):
     p.x=float(input("x coordinate"))
     p.y=float(input("y coordinate:"))

def  print_point(p):
       print("(%g,%g)"%(p.x, p.y))

def  distance(p1,p2):
       d=math.sqrt((p1.x-p2.x)**2+(p1.y-p2.y)**2)
       return d

p1=Point()                      #create first object
print("Enter First point:")
read_point(p1)                  #read x and y for p1
p2=Point()                      #create second object
print("Enter Second point:")
read_point(p2)                  #read x and y for p2
dist=distance(p1,p2)            #compute distance
print("First point is:")
print_point(p1)                 #print p1
print("Second point is:")
print_point(p2)                 #print p2

print("Distance is: %g" %(distance(p1,p2)))
```

The sample output of above program would be –

```
Enter First point:
x  coordinate:10
y  coordinate:20


Enter Second point:
x  coordinate:3
y  coordinate:5
First point is: (10,20)
Second point is:(3,5)
Distance is: 16.5529
```

Explaination of the above program

The class Point contains a string enclosed within 3 double-quotes. This is known as ***docstring***. Comments are written within 3 consecutive double-quotes inside a class, module or function definition. It is an important part of documentation which helps in understanding the code.
The function read_point(p1) takes one argument of type Point object. The parameter p of the function definition will act as an alias for the argument p1. Python understands that the object p1 has two attributes x and y.

Similarly, print_point() also takes one argument and with the help of format-strings and printing the attributes x and y of the Point object as an ordered-pair (x,y).

The Euclidean distance between two points (x1,y1) and (x2,y2) is

$$\sqrt{x1\ \ x2^{\,2}\quad y1\ \ y2^{\,2}}$$

In this program, Point objects (p1.x, p1.y) and (p2.x, p2.y) is applied to the formula on these points by passing objects p1 and p2 as parameters to the function distance().

Thus, the above program gives an idea of defining a class, instantiating objects, creating attributes, defining functions that takes objects as arguments and finally, calling (or invoking) such functions whenever and wherever necessary.

NOTE: User-defined classes in Python have two types of attributes viz. ***class attributes*** and ***instance attributes***. Class attributes are defined inside the class (usually, immediately after class header). They are common to all the objects of that class. That is, they are shared by all the objects created from that class. But, instance attributes defined for individual objects. They are available only for that instance (or object).

4.1.3 Rectangles

It is possible to make an object of one class as an attribute to other class. To illustrate this, consider an example of creating a class called as Rectangle. A rectangle can be created using any of the following data –

- ➔ width and height of a rectangle and one corner point (ideally, a bottom-left corner) in a coordinate system

Program: Write a class Rectangle containing numeric attributes width and height. This class should contain another attribute *corner* which is an instance of another class Point. Implement following functions –

A function to print corner point as an ordered-pair
A function *find_center()* to compute center point of the rectangle
A function *resize()* to modify the size of rectangle

```
class Point:
    """  This is a class Point representing
         coordinate point
    """

class Rectangle:
    """ This is a class Rectangle.
    Attributes: width, height and Corner Point
    """

def  find_center(rect):
    p=Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p

def  resize(rect,w, h):
    rect.width +=w
    rect.height +=h


def  print_point(p):

    print("(%g,%g)"%(p.x, p.y))

box=Rectangle()             #create Rectangle object
box.corner=Point()          #define an attribute corner for box
box.width=100               #set attribute   width to box
box.height=200              #set attribute   height to box
box.corner.x=0              #corner itself   has two attributes x and y
box.corner.y=0                    #initialize x and y to 0

print("Original Rectangle is:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of rectangle is:")
print_point(center)

resize(box,50,70)
print("Rectangle after resize:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of resized rectangle is:")
print_point(center)
```

A sample output would be:

```
Original Rectangle is: width=100, height=200
The center of rectangle is: (50,100)
Rectangle after resize: width=150, height=270
The center of resized rectangle is: (75,135)
```

Program explaination:

Two classes Point and Rectangle have been created with suitable docstrings. box=Rectangle() statement instantiates an object of Rectangle class.

The statement box.corner=Point() indicates that corner is an attribute for the object box and this attribute itself is an object of the class Point. The following statements indicate that the object box has two more attributes – box.width=100 and box.height=200

In this program, the corner points box.corner.x=0 and box.corner.y=0 are origin in coordinate system.

Based on all above statements an object diagram can be drawn as –

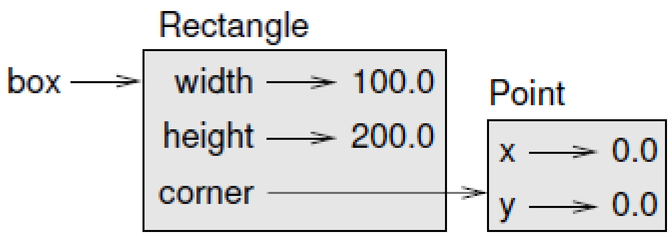


Fig 4.2 Object diagram

The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."

4.1.4 Instances as return values:

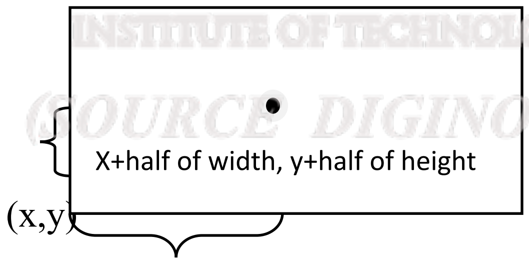
Functions can return instances. In the above program, find_center(rect) takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

Here is an example that passes box as an argument and assigns the resulting Point to center:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

The function find_center() takes an object rect as an argument. So, when a call is made using the statement, center=find_center(box), the object rect acts as an alias for the argument box.

A local object p of type Point has been created inside this function. The attributes of p are x and y, which takes the values as the coordinates of center point of rectangle. Center of a rectangle can be computed with the help of following diagram.



The function find_center() returns the computed center point. Note that, the return value of a function here is an instance of some class. That is, one can have an ***instance as return values*** from a function.

4.1.5 Objects are mutable:

The function resize() in the above program, takes three arguments: rect – an instance of Rectangle class and two numeric variables w and h. The values w and h are added to existing attributes width and height. This clearly shows that ***objects are mutable***. State of an object can be changed by modifying any of its attributes. When this function is called with a statement – resize(box,50,70) the rect acts as an alias for box. Hence, width and height modified within the function will reflect the original object box.

Thus, the above program illustrates the concepts: ***object of one class is made as attribute for object of another class, returning objects from functions*** and ***objects are mutable.***

→ write function grow_rectangle() that modify objects. grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
        rect.width += dwidth
        rect.height += dheight

>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

→ write a function named move_rectangle that takes a Rectangle and two numbers named dx and dy. It should change the location of the rectangle by adding dx to the x coordinate of corner and adding dy to the y coordinate of corner.

```
def move_rectangle(rect, dx, dy):
   """Move the Rectangle by modifying its corner object.
   rect: Rectangle object.
   dx: change in x coordinate (can be negative).
   dy: change in y coordinate (can be negative).
   """
   rect.corner.x += dx
   rect.corner.y += dy


move_rectangle(box, 50, 100)
print(box.corner.x)
print(box.corner.y)
```

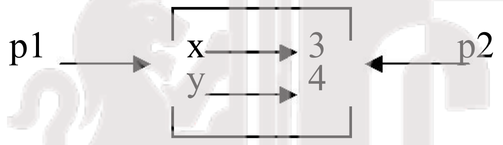
```
output :50, 100
```

4.1.4 Copying

An object will be aliased whenever an object is assigned to another object of same class. For ex:

```
>>> class Point:
            pass
>>> p1=Point()
>>> p1.x=3
>>> p1.y=4
>>> p2=p1
>>> print(p1)
    <__main__.Point object at 0x01581BF0>
>>> print(p2)
    <__main__.Point object at 0x01581BF0>
```

Observe that both p1 and p2 objects have same physical memory. It is clear that the object p2 is an alias for p1. So, we can draw the object diagram as below –



Hence, if we check for equality and identity of these two objects, we will get following result.

```
>>> p1 is p2 True
>>> p1==p2 True
```

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object. It is not a good practice always.
Therefore Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:
To do this, Python provides a module called ***copy*** and a method called ***copy()***

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> import copy
>>> p2 = copy.copy(p1)
```

p1 and p2 contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> print(p1)
    <__main__.Point object at 0x01581BF0>
```

```
>>> print(p3)
     <__main__.Point object at 0x02344A50>
>>> print(p3.x,p3.y)
 3.0  4.0
>>> p1 is p2
False
>>> p1 == p2
False
```

Physical address of the objects p1 and p3 are different. But, values of attributes x and y are same. The is operator indicates that p1 and p2 are not the same object, which is what we expected. But, == operator is generating False though the contents of two objects are same. The reason is the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent.

The ***copy()*** method of ***copy*** module duplicates the object. The content (i.e. attributes) of one object is copied into another object. But when we use copy.copy to duplicate a Rectangle, then it copies the Rectangle object but not the embedded Point.

```
import copy
 class Point:
        """  This is a class Point representing
             coordinate point
        """

class Rectangle:
        """ This is a class Rectangle.
        Attributes: width, height and Corner Point
        """

box1=Rectangle()
box1.corner=Point()
box1.width=100
box1.height=200
box1.corner.x=0
box1.corner.y=0
box2=copy.copy(box1)
print(box1 is box2)                          #prints   False
print(box1.corner is box2.corner)            #prints   True
```

Here, box1.corner and **box2.corner** are same objects, even though **box1** and **box2** are different. Whenever the statement box2=copy.copy(box1) is executed, the contents of all the attributes of box1 object are copied into the respective attributes of box2 object. That is, box1.width is copied into box2.width, box1.height is copied into box2.height. Similarly, box1.corner is copied into box2.corner. Now, recollect the fact that corner is not exactly the object itself, but it is a reference to the object of type Point. Hence, the value of reference (that is, the physical address) stored in box1.corner is copied into box2.corner. Thus, the physical object to which box1.corner and box2.corner are pointing is only one. This type of copying the objects is known as ***shallow copy.*** To understand this behavior, observe the following diagram

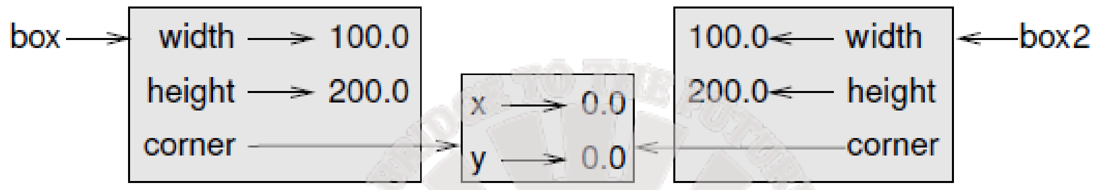


Fig 4.4 Object diagram

Now, the attributes width and height for two objects box1 and box2 are independent. Whereas, the attribute corner is shared by both the objects. Thus, any modification done to box1.corner will reflect box2.corner as well. Obviously, we don't want this to happen, whenever we create duplicate objects. That is, we want two independent physical objects. Python provides a method ***deepcopy()*** for doing this task. This method copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on.

```
box3=copy.deepcopy(box1)
print(box1 is box3)                  #prints   False
print(box1.corner is box3.corner)    #prints   False
```

Thus, the objects box1 and box3 are now completely independent.

4.1.5 Debugging

While dealing with classes and objects, we may encounter different types of errors. For example, if we try to access an attribute which is not there for the object, we will get
AttributeError. For example –

```
>>> p= Point()
>>> p.x = 1
>>> p.y = 2
>>> print(p.z)
    AttributeError: 'Point' object has no attribute 'z'
```

To avoid such error, it is better to enclose such codes within try/except as given below – try:

```
        z = p.x
```

```
    except AttributeError: z = 0
```

When we are not sure, which type of object it is, then we can use ***type()*** as –

```
    >>> type(box1)
            <class '__main__.Rectangle'>
```

Another method ***isinstance()*** helps to check whether an object is an instance of a particular class –

```
    >>> isinstance(box1,Rectangle) True
```

When we are not sure whether an object has a particular attribute or not, use a function ***hasattr()*** –

```
    >>> hasattr(box1, 'width') True
```

Observe the string notation for second argument of the function ***hasattr()***. Though the attribute width is basically numeric, while giving it as an argument to function ***hasattr()***, it must be enclosed within quotes.

4.2 CLASSES AND FUNCTIONS

In this chapter Python presents functional programming style and two types of functions i.e ***pure functions*** and ***modifiers***.

4.2.1 Pure Functions

Pure functions is one that does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To understand the concept of pure functions, let us consider an example of creating a class called Time. An object of class Time contains hour, minutes and seconds as attributes.
Define a class called Time that records the time of day. write a function called print_time that takes a Time object and prints it in the form hour:minute:second.
Add two time objects that yield proper result and hence need to check whether number of seconds exceeds 60, minutes exceeds 60 etc, and take appropriate action.

```
class Time:
    """Represents the time of a day Attributes:
    hour, minute, second """
def printTime(t):
    print("%.2d:%.2d:%.2d"%(t.hour,t.minute,t.second))
def add_time(t1,t2):
    sum=Time()
    sum.hour = t1.hour + t2.hour sum.minute =
    t1.minute + t2.minute sum.second =
    t1.second + t2.second
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1
    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1
    return sum

t1=Time()
t1.hour=10
t1.minute=34
t1.second=25
print("Time1     is:")
```

```
printTime(t1)
t2=Time()
t2.hour=2
t2.minute=12
t2.second=41
print("Time2     is     :")
printTime(t2)
t3=add_time(t1,t2)
print("After    adding    two    time    objects:")
printTime(t3)
```

The output of this program would be –

```
Time1 is: 10:34:25
Time2 is : 02:12:41
After adding two time objects: 12:47:06
```

Here, the function add_ time() takes two arguments of type Time, and returns a Time object, whereas, it is not modifying contents of its arguments t1 and t2. Such functions are called as *pure functions.*

4.2.2 Modifiers

Sometimes it is useful for a function to modify the objects passed as parameters. In that case, the changes are visible to the caller and the parameter value changes during function execution. Functions that work this way are called **modifiers**.

WRITE A FUNCTION increment(), which adds a given number of seconds to a Time object

```
def  increment(t,seconds):
    t.second += seconds

     while   t.second   >=   60:
         t.second     -=     60
         t.minute += 1

     while   t.minute   >=   60:
         t.minute     -=     60
         t.hour += 1
```

In this function, we initially add the argument seconds to t.second. Now, there is a chance that t.second is exceeding 60. So, we will increment minute counter till t.second becomes lesser than 60. Similarly, till the t.minute becomes lesser than 60, we will decrement minute counter. Note that, the modification is done on the argument itself. Thus, the above function is a ***modifier.***

4.2.3 Prototyping v/s Planning

When a problem statement is not known completely , we may write the program initially, and then keep modifying it as and when requirement (problem definition) changes. This methodology is known as ***prototype and patch.*** That is, first design the prototype based on the information available and then perform patch-work as and when extra information is gathered.
An alternative is ***designed development***, in which high-level insight into the problem can make the programming much easier. For example,

In add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem—
we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.
Here is a function that converts Times to integers:

```
def time_to_int(time):
        minutes = time.hour * 60 + time.minute
        seconds = minutes * 60 + time.second
        return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
        time = Time()
        minutes, time.second = divmod(seconds, 60)
        time.hour, time.minute = divmod(minutes, 60)
        return time
```

4.2.4 Debugging

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like valid_time that takes a Time object and returns False if it violates an invariant:

```
        def valid_time(time):
                if time.hour < 0 or time.minute < 0 or time.second < 0:
                        return False
                if time.minute >= 60 or time.second >= 60:
                        return False
                return True
```

Now, at the beginning of add_time() function, we can put a condition as –

```
        def add_time(t1, t2):
                if not valid_time(t1) or not valid_time(t2):
                        raise ValueError('invalid Time object in add_time')

                #remaining statements of add_time() functions
```

Python provides another debugging statement ***assert***. When this keyword is used, Python evaluates the statement following it. If the statement is True, further statements will be evaluated sequentially. But, if the statement is False, then ***AssertionError*** exception is raised. The usage of *assert* is shown here –

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2) #remaining
    statements of add_time() functions
```

The *assert* statement clearly distinguishes the normal conditional statements as a part of the logic of the program and the code that checks for errors.

4.3 CLASSES AND METHODS

The classes considered so far were empty classes without having any definition. But, in a true object oriented programming, a class contains class-level attributes, instance-level attributes, methods etc. There will be a relationship between the object of the class and the function that operate on those objects.

4.3.1 Object-Oriented Features

As an object oriented programming language, Python possess following characteristics:

- ➔ Programs include class and method definitions.
- ➔ Most of the computation is expressed in terms of operations on objects.
- ➔ Objects often represent things in the real world, and methods often correspond to the ways objects in the real world interact.

To establish strong relationship between the object of the class and a function, we must define a function as a member of the class. A function which is associated with a particular class is known as a ***method***. Methods are semantically the same as functions, but there are two syntactic differences:

- ➔ Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- ➔ The syntax for invoking a method is different from the syntax for calling a function.

4.3.2 The __init__() Method

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is __init__ (two underscore characters, followed by init, and then two more underscores). An init method for the Time class might look like this:

```
class Time:
        """    """

        def __init__(self, hour=0, minute=0, second=0):
                self.hour = hour
                self.minute = minute
                self.second = second
```

It is common for the parameters of __init__ to have the same names as the attributes. The statement self.hour = hour stores the value of the parameter hour as an attribute of self.

The parameters are optional, so if you call Time with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```
If you provide two arguments, they override hour and minute.
```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```
And if you provide three arguments, they override all three default values.

4.2.3 The __str__ method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object.

For example, here is a str method for Time objects:
```
class Time:
        “””   “””


        def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the str method:
```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```
It is a good practice to always start by writing __init__, which makes program easier to instantiate objects, and __str__, which is useful for debugging.

this code is example for __str__
```
class Student:
   def __init__(self,rollNo,firstName,lastName):
      self.rollNo = rollNo
      self.firstname = firstName
      self.lastname = lastName

   def __str__(self):
      #convert the int to string
      s_rollNo = str(self.rollNo)
      return s_rollNo+":"+self.firstname+" "+self.lastname

s= Student(34,"chaitra"," rao")
print(s)
```

Consider the previous example and convert all the functions to methods as shown below

```
import math
class Point:
        def  __init__(self,a,b):
                self.x=a
                self.y=b

        def dist(self,p2):
                d=math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2) return d

        def __str__(self):
                return "(%d,%d)"%(self.x, self.y)

p1=Point(10,20)          #__init__() is   called automatically
p2=Point(4,5)            #__init__()  is  called automatically
print("P1    is:",p1)    #__str__()    is  called   automatically
print("P2    is:",p2)    #__str__()    is  called   automatically

d=p1.dist(p2)            #explicit call   for dist()
print("The distance is:",d)
```

The sample output is –

```
P1 is: (10,20)
P2 is: (4,5)
Distance is: 16.15549442140351
```

Program explaination:
Every method of any class must have the first argument as ***self***. By convention, the first parameter of a method is called self. The argument ***self*** is a reference to the current object. That is, it is reference to the object which invoked the method.

```
class Time:
        def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:
• The syntax for a function call, print_time(start), suggests that the function is the active agent.
• In object-oriented programming, the objects are the active agents. A method invocation will be start.print_time()This change in perspective might be more polite, but it is not obvious that it is useful. In the
examples we have seen so far, it may not be. But sometimes shifting responsibility from the
functions onto the objects makes it possible to write more versatile functions (or methods),
and makes it easier to maintain and reuse code.

The method __init__() inside the class is an initialization method, which will be invoked automatically when the object gets created. When the statement like –

```
p1=Point(10,20)
```

is used, the __init__() method will be called automatically. The internal meaning of the above line is –

```
p1.__init__(10,20)
```

Here, p1 is the object which is invoking a method. Hence, reference to this object is created and passed to __init__() as *self*. The values 10 and 20 are passed to formal parameters a and b of __init__() method. Now, inside __init__() method, we have statements

```
self.x=10
self.y=20
```

This indicates, x and y are instance attributes. The value of x for the object p1 is 10 and, the value of y for the object p1 is 20.

When we create another object p2, it will have its own set of x and y. That is, memory locations of instance attributes are different for every object.

Thus, state of the object can be understood by instance attributes.

The method dist() is an ordinary member method of the class Point. As mentioned earlier, its first argument must be self. Thus, when we make a call as –

```
d=p1.dist(p2)
```

a reference to the object p1 is passed as self to dist() method and p2 is passed explicitly as a second argument. Now, inside the dist()method, we are calculating distance between two point (Euclidian distance formula is used) objects. Note that, in this method, we cannot use the name p1, instead we will use self which is a reference (alias) to p1.

4.3.3 Operator Overloading

POLYMORPHISM (Poly= many; Morph = form) : Refers to the ability of an object/method to behave di erently in di erent situations.

Basic operators like +, -, * etc. can be overloaded. To overload an operator, one needs to write a method within user-defined class. Python provides a special set of methods which is to be used for overloading required operator.

When + operator is applied to Time objects, Python invokes __add__method When you print the result, Python invokes __str__, Changing the behavior of an operator so that it works with programmer-defined types
is called operator overloading. For every operator in Python there is a corresponding special method, like __add__.

Let us consider an example of Point class considered earlier. Using operator overloading, we can try to add two point objects. Consider the program given below –

```
class Point:
        def  __init__(self,a=0,b=0):
                self.x=a
                self.y=b

        def  __add__(self,       p2):
                p3=Point()
                p3.x=self.x+p2.x
                p3.y=self.y+p2.y
                return p3

        def __str__(self):
                return "(%d,%d)"%(self.x, self.y)


p1=Point(10,20)
p2=Point(4,5)
print("P1          is:",p1)
print("P2 is:",p2)
p4=p1+p2 #call for __add__() method print("Sum is:",p4)
```

The output would be –

```
P1 is: (10,20)
P2 is: (4,5)
Sum is: (14,25)
```

In the above program, when the statement p4 = p1+p2 is used, it invokes a special method __add__() written inside the class. Because, internal meaning of this statement is

p4 = p1.__add__(p4)

Here, p1 is the object invoking the method. Hence, self inside __add__() is the reference (alias) of p1. And, p4 is passed as argument explicitly.

In the definition of __add__(), we are creating an object p3 with the statement – p3=Point()

The object p3 is created without initialization. Whenever we need to create an object with and without initialization in the same program, we must set arguments of __init__() for some default values. Hence, in the above program arguments a and b of __ init__() are made as default arguments with values as zero. Thus, x and y attributes of p3 will be now zero. In the __add__() method, we are adding respective attributes of self and p2 and storing in p3.x and p3.y. Then the object p3 is returned. This returned object is received as p4 and is printed.

Let us consider a more complicated program involving overloading. Consider a problem of creating a class called Time, adding two Time objects, adding a number to Time object etc. that we had considered in previous section. Here is a complete program with more of OOP concepts.

```
class Time:
    def __init__(self,      h=0,m=0,s=0):
        self.hour=h
        self.min=m
        self.sec=s

    def time_to_int(self):
        minute=self.hour*60+self.min
        seconds=minute*60+self.sec return
        seconds

    def int_to_time(self,       seconds):
        t=Time()
        minutes,     t.sec=divmod(seconds,60)
        t.hour, t.min=divmod(minutes,60) return t

    def __str__(self):
        return "%.2d:%.2d:%.2d"%(self.hour,self.min,self.sec)

    def __eq__(self,t):
        return self.hour==t.hour and self.min==t.min and self.sec==t.sec

    def __add__(self,t):
        if  isinstance(t,  Time):  return
            self.addTime(t)
        else:
            return self.increment(t)

    def addTime(self,  t):  seconds=self.time_to_int()+t.time
        _to_int() return self.int_to_time(seconds)

    def increment(self,  seconds):  seconds  +=
        self.time_to_int()              return
        self.int_to_time(seconds)

    def __radd__(self,t):     return
        self.__add__(t)
```

```
T1=Time(3,40)

T2=Time(5,45)
print("T1 is:",T1)
print("T2 is:",T2)
print("Whether T1 is same as T2?",T1==T2)              #call for __eq__()
T3=T1+T2                                  #call for __add__()

print("T1+T2 is:",T3)
T4=T1+75                                  #call for __add__()
print("T1+75=",T4)
T5=130+T1                                 #call for __radd__()
print("130+T1=",T5)
T6=sum([T1,T2,T3,T4])
print("Using sum([T1,T2,T3,T4]):",T6)
```

The output would be –

```
    T1 is: 03:40:00
    T2 is: 05:45:00
    Whether T1 is same as T2? False
    T1+T2 is: 09:25:00
    T1+75=
    03:41:15
    130+T1=
    03:42:10
    Using sum([T1,T2,T3,T4]): 22:31:15
```

Program explaination

The class Time has __init__() method for initialization of instance attributes hour, min and sec. The default values of all these are being zero.

The method time_to_int() is used convert a Time object (hours, min and sec) into single integer representing time in number of seconds.

The method int_to_time() is written to convert the argument seconds into time object in the form of hours, min and sec. The built-in method ***divmod()*** gives the quotient as well as remainder after dividing first argument by second argument given to it.

Special method __eq__() is for overloading equality (==) operator. We can say one Time object is equal to the other Time object if underlying hours, minutes and seconds are equal respectively. Thus, we are comparing these instance attributes individually and returning either True of False.

When we try to perform addition, there are 3 cases –

- o Adding two time objects like T3=T1+T2.

- o Adding integer to Time object like T4=T1+75
- o Adding Time object to an integer like T5=130+T1

Each of these cases requires different logic. When first two cases are considered, the first argument will be T1 and hence self will be created and passed to __add__() method.
Inside this method, we will check the type of second argument using isinstance() method. If the second argument is Time object, then we call addTime() method. In this method, we will first convert both Time objects to integer (seconds) and then the resulting sum into Time object again. So, we make use time_to_int() and int_to_time() here. When the 2nd argument is an integer, it is obvious that it is number of seconds. Hence, we need to call increment() method.

Thus, based on the type of argument received in a method, we take appropriate action. This is known as ***type-based dispatch.***

In the 3rd case like T5=130+T1, Python tries to convert first argument 130 into self, which is not possible. Hence, there will be an error. This indicates that for Python, T1+5 is not same as 5+T1 (Commutative law doesn't hold good!!). To avoid the possible error, we need to implement ***right-side addition*** method __radd__(). Inside this method, we can call overloaded method __add__().
The beauty of Python lies in surprising the programmer with more facilities!! As we have implemented __add__() method (that is, overloading of + operator), the built-in sum() will is capable of adding multiple objects given in a sequence. This is due to ***Polymorphism*** in Python**.** Consider a list containing Time objects, and then call sum() on that list as –

```
T6=sum([T1,T2,T3,T4])
```

The sum() internally calls __ add__() method multiple times and hence gives the appropriate result. Note down the square-brackets used to combine Time objects as a list and then passing it to sum().