CLASSES AND OBJECTS

- Python is an object-oriented programming language, and *class* is a basis for any object oriented programming language.
- 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).

OOPS Concepts

Class:

A class in C++ is the building block, that leads to Object-Oriented programming.

- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

A C++ class is like a blueprint for an object.

For Example: Consider the Class of **Cars**.

There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc.

So here, Car is the class and wheels, speed limits, mileage are their properties.

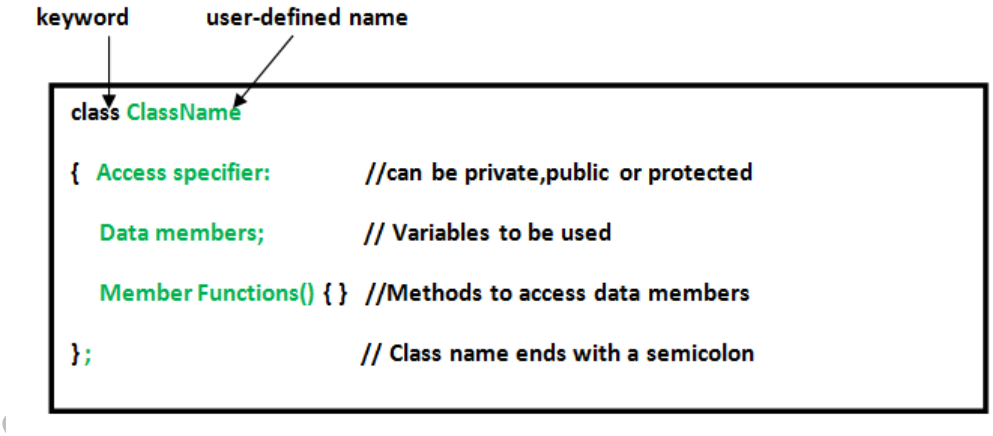
1. A Class is a user defined data-type which has data members and member functions.

2. Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
3. In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc. and member functions can be *apply brakes*, *increase speed* etc.

❖ An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax: ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object.

For example

if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

----------*****----------

Programmer-defined Types

A class in Python can be created using a keyword class. Here, we are creating an empty class without any members by just using the keyword passwithin it.

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

The output would be –

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

The term_main_ indicates that the class Point is in the main scope of the current module. In other words, this class is at the top level while executing the program.

Now, a user-defined data type Point got created, and this can be used to create any number of objects of this class. Observe the following statements –

```
p=Point()
```

Now, a reference (for easy understanding, treat reference as a pointer) to Point object is created and is returned. This returned reference is assigned to the object p.

The process of creating a new object is called as ***instantiation*** and the object is ***instance*** of a class.

When we print an object, Python tells which class it belongs to and where it is stored in the memory.

```
print(p)
```

The output would be –

```
<__main__.Point object at 0x003C1BF0>
```

The output displays the address (in hexadecimal format) of the object in the

memory. It is now clear that, the object occupies the physical space, whereas the class does not.

Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.

The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring.
                I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores __.

For example,

__doc__ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
  "This is a person class"
    age = 10

        def greet(self):
          print('Hello')

                # Output: 10
                 print(Person.age)
```

```
# Output: <function Person.greet>
print(Person.greet)

# Output: 'This is my second class'
print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>  →Hexadecimal
This is a person class
```

Attributes

An object can contain named elements known as ***attributes***. One can assign values to these attributes using dot operator.

For example, keeping coordinate points in mind, we can assign two attributes x and yfor the object pof a class Pointas below –

```
p.x =10.0
p.y =20.0
```

Note: We are assigning values to named elements of an object. These elements are called **attributes**.

A state diagram that shows an object and its attributes is called as ***object diagram.*** For the object p, the object diagram is shown in Figure-1:

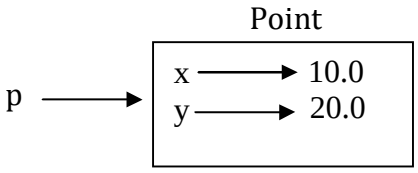


Figure-1: Object Diagram

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

Here, p.x means *"Go to the object p refers to and get the value of x".* In this case, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

You can use dot notation as part of any expression.

For example:

```
>>> print '(%g, %g)' % (p.x, p.y)    (%g-> means for float type with the current precision)
(3.0, 4.0)
>>> distance = math.sqrt(p.x**2 + p.y**2)
>>> print distance
 5.0
```

Attributes of an object can be assigned to other variables –

```
>>> x= p.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.

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 –

- By knowing width and height of a rectangle and one corner point (ideally, a bottom- left corner) in a coordinate system
- By knowing two opposite corner points

- Let us consider the first technique and implement the task: 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

The program is as given below –

```
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))
```

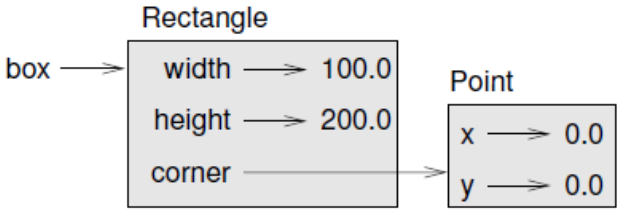
```
center=find_center(box)
print("The center of resized rectangleis:")
print_point(center)
```

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)
```

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."

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



An object that is an attribute of another object is **embedded**

Instances as return values.

Functions can return instances. For example, find_center takes a Rectangle as an argument and

returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(box):
p = Point()
p.x = box.corner.x + box.width/2.0
p.y = box.corner.y + box.height/2.0
return p
```

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

```
>>> center = find_center(box)
>>> print_point(center)
```

(50.0, 100.0)

Objects are mutable

You can change the state of an object by making an assignment to one of its attributes.

For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.width + 100
```

- You can also write functions that modify objects. For example, 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
```

Here is an example that demonstrates the effect:

```
>>> print box.width
100.0
>>> print box.height
200.0
>>> grow_rectangle(box, 50, 100)
>>> print box.width
150.0
>>> print box.height
300.0
```

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

Copying

An object will be aliased whenever there an object is assigned to another object of same class.

This may happen in following situations –

- Direct object assignment (like p2=p1)
- When an object is passed as an argument to a function
- When an object is returned from a function

The last two cases have been understood from the two programs in previous sections.

Let us understand the concept of aliasing more in detail using the following program –

The copy module contains a function called copy that can duplicate any object:

```
>>> 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)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

- The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data.

- In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.

This behavior can be changed—we'll see how later.

If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

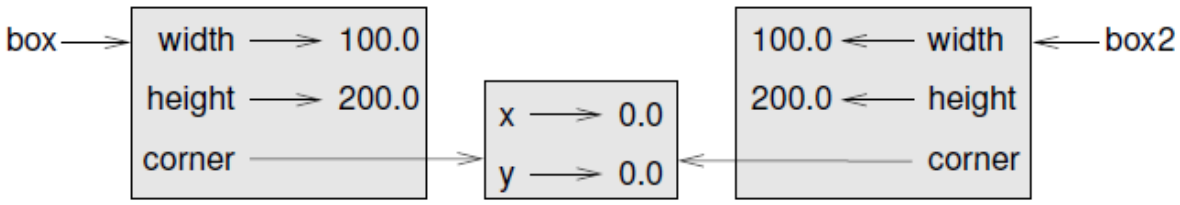
```
>>> box2 = copy.copy(box)
>>> box2 is box
```

False

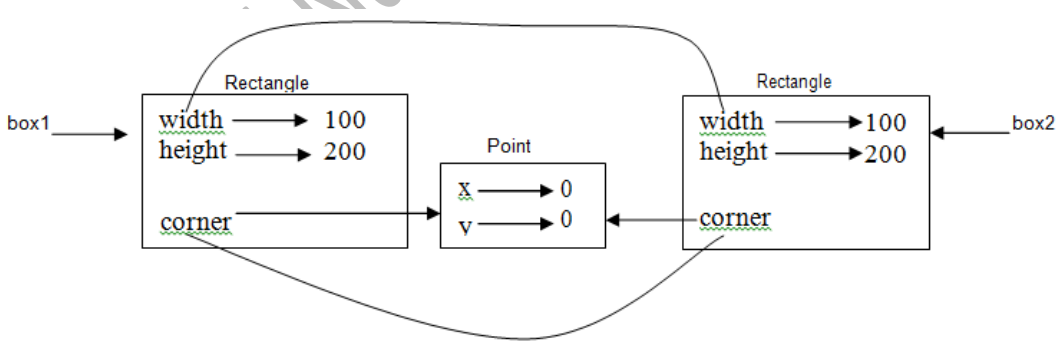
```
>>> box2.corner is box.corner
```

True

Here is what the object diagram looks like:



This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects



- 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.

For most applications, this is not what you want. In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behavior is confusing and error-prone.

Fortunately, 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. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 and box are completely separate objects.

CLASSES AND FUNCTIONS

Though Python is object oriented programming languages, it is possible to use it as functional programming. There are two types of functions viz. ***pure functions*** and ***modifiers***. A pure function takes objects as arguments and does some work without modifying any of the original argument. On the other hand, as the name suggests, modifier function modifies the original argument.

In practical applications, the development of a program will follow a technique called as ***prototype*** and ***patch***. That is, solution to a complex problem starts with simple prototype and incrementally dealing with the complications.

Time

As another example of a user-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
class Time(object):
```

"""represents the time of day.

attributes: hour, minute, second"""

We can create a new Time object and assign attributes for hours, minutes, and seconds:

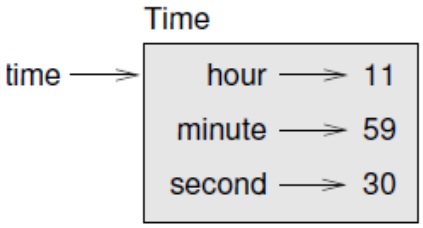
time = Time()

time.hour = 11

time.minute = 59

time.second = 30

The state diagram for the Time object looks like this:



Pure functions

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. Write a function to print time in HH:MM:SS format and another function to add two time objects.

Note that, adding two time objects should yield proper result and hence we 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.*

Modifiers

Sometimes, it is necessary to modify the underlying argument so as to reflect the caller. That is, arguments have to be modified inside a function and these modifications should be available to the caller.

The functions that perform such modifications are known as ***modifier function.*** Assume that, we need to add few seconds to a time object, and get a new time.

Then, we can write a function as below –

```
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 will 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 t itself. Thus, the above function is a ***modifier.***

Prototyping versus planning

Whenever we do not know the complete problem statement, we may write the program initially, and then keep of 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. But, this type of incremental development may end-up in unnecessary code, with many special cases and it may be unreliable too.

An alternative is ***designed development***, in which high-level insight into the problem can make the programming much easier.

For example,

if we consider the problem of adding two time objects, adding seconds to time object etc. as a problem involving numbers with base 60 (as every hour is 60 minutes and every minute is 60 seconds), then our code can be improved. Such improved versions are discussed later in this chapter.

CLASSES AND METHODS

Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming.

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 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.

Printing Objects

In prevous Chapter, we defined a class named Time and you wrote a function named

print_time:

```
class Time(object):
        """represents the time of day.
                        attributes: hour, minute, second"""
def print_time(time):
print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

To call this function, you have to pass a Time object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
```

```
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

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

Now there are two ways to call print_time. The first (and less common) way is to use function

syntax:

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

- In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Another example

Here's a version of increment (from Section 16.3) rewritten as a method:

```
# inside class Time:
        def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

This version assumes that time_to_int is written as a method, note that it is a pure function, not a modifier.

Here's how you would invoke increment:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.

This mechanism can be confusing, especially if you make an error.

For example, if you invoke increment with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

A more complicated example

is_after is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter self and the second parameterother:

```
# inside class Time:
    def is_after(self, other):
    return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

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:

```
# inside 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.)

If you provide two arguments, they override hour, minute and Second

```
>>> time = Time(9, 45, 15)
>>> time.print_time()
09:45:15
```

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:

```
# inside 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:

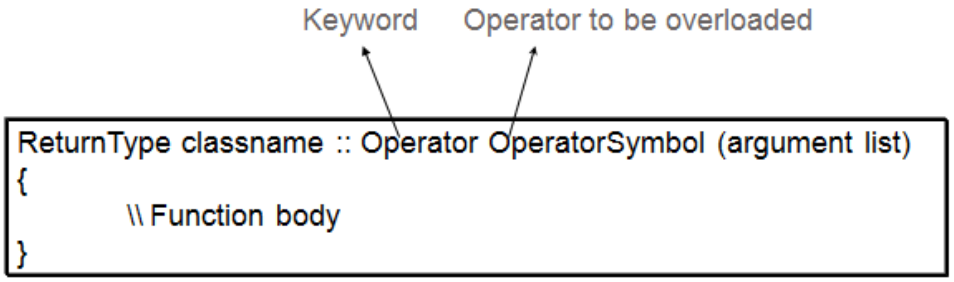
```
>>> time = Time(9, 45)
>>> print time
09:45:00
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

Operator overloading

Ability of an existing operator to work on user-defined data type (class) is known as operator overloading. It is a polymorphic nature of any object oriented programming. 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 have to be used for overloading required operator. The method should consist of the code what the programmer is willing to do with the operator.



Following table shows gives a list of operators and their respective Python methods for overloading.

Operator	Special Function in Python	Operator	Special Function in Python
+	__add__()	<=	__le__()
-	__sub__()	>=	__ge__()
*	__mul__()	==	__eq__()
/	__truediv__()	!=	__ne__()
%	__mod__()	in	__contains_()
<	__lt__()	len	__len__()
>	__gt__()	str	__str__()

For example, if you define a method named _add_ for the Time class, you can use the + operator on Time objects.

Here is what the definition might look like:

```
# inside class Time:
    def __add__(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)
```

And here is how you could use it:

```
    >>> start = Time(9, 45)
```

```
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

When you apply the + operator to Time objects, Python invokes __add__. When you print the result,

Python invokes __str__. So there is quite a lot happening behind the scenes! Changing the behavior of an operator so that it works with user-defined types is called **operator overloading**.

For every operator in Python there is a corresponding special method, like __add__.

Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object.

The following is a version of __add__ that checks the type of other and invokes either

add_time or increment:

```
# inside class Time:
def __add__(self, other):
if isinstance(other, Time):
return self.add_time(other)
else:
return self.increment(other)

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

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

The built-in function isinstance takes a value and a class object, and returns True if the value isan instance of the class.

If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how to do that. But there is a clever solution for this problem: the special method __radd__, which stands for "right-side add."

This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:
def __radd__(self, other):
return self.__add__(other)
```

And here's how it's used:

```
>>> print 1337 + start
10:07:17
```

Polymorphism

What is Polymorphism : The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.

OR

The literal meaning of polymorphism is the condition of occurrence in different forms.

Example of inbuilt polymorphic functions :

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

OUTPUT

5
3

Examples of used defined polymorphic functions :

```
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
        return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

OUTPUT

5
9
