CHAPTER 11

Operator Overloading

Basic Concepts of Operator Overloading • Advantages • Overloading Arithmetic and Logical Operators • Reverse Adding • Overriding __getitem__(), __setitem__(), in operator, and __call__() • Overloading Miscellaneous Functions

11.1 INTRODUCTION

Till now, we have seen that Python is an interesting and easy language. You can build classes with desired attributes and methods. But just think, if you want to add two Time values, where Time is a user-defined class, then how good it would be if we write `T3 = T1 + T2`, where `T1, T2,` and `T3` are all objects of the class Time. As of now, we need to write the same statement as `T3 = T1.add(T2)`.

Basically, the meaning of operators like `+, =, *, /, >, <`, etc. are pre-defined in any programming language. Programmers can use them directly on built-in data types to write their programs. But, for user-defined types like objects, these operators do not work. Therefore, Python allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called operator overloading. *Operator overloading* allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can be also applied to user-defined data types.

You already have a clue of operator overloading. Just give a thought, if you write `5 + 2`, then the integers are added, when you write `str1 + str2`, two strings are concatenated, when you write `List1 + List2`, the two lists are merged, so on and so forth. Thus, we see that the same operator behaves differently with different types.

11.1.1 Concept Of Operator Overloading

With operator overloading, a programmer is allowed to provide his own definition for an operator to a class by overloading the built-in operator. This enables the programmer to perform some specific computation when the operator is applied on class objects and to apply a standard definition when the same operator is applied on a built-in data type.

This means that while evaluating an expression with operators, Python looks at the operands around the operator. If the operands are of built-in types, Python calls a built-in routine. In case, the operator is being applied on user-defined operand(s), the Python compiler checks to see if the programmer has an overloaded operator function that it can call. If such a function whose parameters match the type(s) and number of the operands exists in the program, the function is called, otherwise a compiler error is generated.

Another form of Polymorphism

Like function overloading, operator overloading is also a form of compile-time polymorphism. Operator overloading, is therefore less commonly known as operator *ad hoc polymorphism* since different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

Note Ad hoc polymorphism is a specific case of polymorphism where different operators have different implementations depending on their arguments.

11.1.2 Advantage of Operator Overloading

We can easily write our Python programs without the knowledge of operator overloading, but the knowledge and use of this feature can help us in many ways. Some of them are:

- With operator overloading, programmers can use the same notations for user-defined objects and built-in objects. For example, to add two complex numbers, we can simply write `C1 + C2`.
- With operator overloading, a similar level of syntactic support is provided to user-defined types as provided to the built-in types.
- In scientific computing where computational representation of mathematical objects is required, operator overloading provides great ease to understand the concept.
- Operator overloading makes the program clearer. For example, the statement

`(C1.mul(C2).div(C1.add(C2))` can be better written as `C1 * C2 / C1 + C2`

11.2 IMPLEMENTING OPERATOR OVERLOADING

Just consider the code given below which is trying to add two complex numbers and observe the result.

Example 11.1 Program to add two complex numbers without overloading the + operator

```
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0
    def setValue(self, real, imag):
        self.real = real
        self.imag = imag
    def display(self):
        print("(", self.real, " + ", self.imag, "i)")
C1 = Complex()
C1.setValue(1,2)
C2 = Complex()
C2.setValue(3,4)
C3 = Complex()
C3 = C1 + C2
C3.display()
```

OUTPUT

```
Traceback (most recent call last):
```

```
  File "C:\Python34\Try.py", line 15, in <module>
    C3 = C1 + C2
TypeError: unsupported operand type(s) for +: 'instance' and 'instance'
```

So, the reason for this error is simple. + operator does not work on user-defined objects. Now, to do the same concept, we will add an operator overloading function in our code. For example, look at the code given below which has the overloaded add function specified as `__add__()`.

Example 11.2 Program to overload the + operator on a complex object

```
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0
    def setValue(self, real, imag):
        self.real = real
        self.imag = imag
    def __add__(self, C):
        Temp = Complex()
        Temp.real = self.real + C.real
        Temp.imag = self.imag + C.imag
        return Temp
    def display(self):
        print("(", self.real, " + ", self.imag, "i)")
C1 = Complex()
C1.setValue(1,2)
C2 = Complex()
C2.setValue(3,4)
C3 = Complex()
C3 = C1 + C2
Print("RESULT = ")
C3.display()
```

OUTPUT

```
RESULT =  ( 4  +  6 i)
```

In the program, when we write `C1 + C2`, the `__add__()` function is called on `C1` and `C2` is passed as an argument. Remember that, user-defined classes have no + operator defined by default. The only exception is when you inherit from an existing class that already has the + operator defined.

Note The `__add__()` method returns the new combined object to the caller.

We can also overload the comparison operators to work with class objects. But before we write further programs, let us first have a look at Table 11.1 to know the name of the function for each operator.

Table 11.1 Operators and their corresponding function names

Operator	Function Name	Operator	Function Name
+	`__add__`	+=	`__iadd__`
-	`__sub__`	-=	`__isub__`
*	`__mul__`	*=	`__imul__`
/	`__truediv__`	/=	`__idiv__`
**	`__pow__`	**=	`__ipow__`
%	`__mod__`	%=	`__imod__`
>>	`__rshift__`	>>=	`__irshift__`
&	`__and__`	&=	`__iand__`
|	`__or__`	|=	`__ior__`
^	`__xor__`	^=	`__ixor__`
~	`__invert__`	~=	`__iinvert__`
<<	`__lshift__`	<<=	`__ilshift__`
>	`__gt__`	<=	`__le__`
<	`__lt__`	==	`__eq__`
>=	`__ge__`	!=	`__ne__`

The program given below compares two `Book` objects. Although the class `Book` has three, attributes, comparison is done based on its price. However, this is not mandatory. You can compare two objects based on any of the attributes.

Example 11.3 Program to compare two objects of user-defined class type

```
class Book:
    def __init__(self):
        title = ""
        publisher = ""
        price = 0
    def set(self, title, publisher, price):
        self.title = title
        self.publisher = publisher
        self.price = price
    def display(self):
        print("TITLE : ", self.title)
        print("PUBLISHER : ", self.publisher)
        print("PRICE : ", self.price)
    def __gt__(self, B):
        if self.price > B.price:
            return True
        else:
            return False
B1 = Book()
```

```
B1.set("OOP with C++", "Oxford University Press", 525)
B2 = Book()
B2.set("Let us C++", "BPB", 300)
if B1>B2:
    print("This book has more knowledge so I will buy")
    B1.display()
```

OUTPUT

```
This book has more knowledge so I will buy
TITLE :  OOP with C++ PUBLISHER :  Oxford University Press PRICE :  525
```

PROGRAMMING EXAMPLES

Program 11.1 Write a program that overloads the + operator on a class Student that has attributes name and marks.

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    def display(self):
        print(self.name, self.marks)
    def __add__(self, S):
        Temp = Student(S.name, [])
        for i in range(len(self.marks)):
            Temp.marks.append(self.marks[i] + S.marks[i])
        return Temp
S1 = Student("Nikhil", [87, 90, 85])
S2 = Student("Nikhil", [83, 86, 88])
S1.display()
S2.display()
S3 = Student("",[])
S3 = S1 + S2
S3.display()
```

OUTPUT

```
Nikhil [87, 90, 85]
Nikhil [83, 86, 88]
Nikhil [170, 176, 173]
```

Program 11.2 Write a program that overloads the + operator to add two objects of class Matrix.

```
class Matrix:
    def __init__(self, List):
        self.List = List
    def display(self):
        print(self.List)
    def __add__(self, M):
        Temp = Matrix([])
        for i in range(len(self.List)):
            for j in range(len(self.List[0])):
                Temp.List.append(self.List[i][j] + M.List[i][j])
        return Temp
M1 = Matrix([[1,2],[3,4]])
M2 = Matrix([[3,4],[5,1]])
M3 = Matrix([])
M3 = M1 + M2
print("RESULTANT MATRIX IS : ")
M3.display()
```

OUTPUT

```
RESULTANT MATRIX IS :  [4, 6, 8, 5]
```

Program 11.3 Write a program that overloads the + operator so that it can add two objects of class Fraction.

```
def GCD(num, deno):
    if(deno == 0):
        return num
    else:
        return GCD(deno, num%deno)
class Fraction:
    def __init__(self):
        self.num = 0
        self.deno = 1
    def get(self):
        self.num = int(input("Enter the numerator : "))
        self.deno = int(input("Enter the denominator : "))
    def simplify(self):
        common_divisor = GCD(self.num, self.deno)
        self.num //= common_divisor
        self.deno //= common_divisor
    def __add__(self, F):
        Temp = Fraction()
        Temp.num = (self.num * F.deno) + (F.num * self.deno)
        Temp.deno = self.deno * F.deno
        return Temp
    def display(self):
        self.simplify()
        print(self.num, "/", self.deno)
F1 = Fraction()
```

```
F1.get()
F2 = Fraction()
F2.get()
F3 = Fraction()
F3 = F1 + F2
print("RESULTANT FRACTION IS : ")
F3.display()
```

OUTPUT

```
Enter the numerator : 4
Enter the denominator : 10
Enter the numerator : 2
Enter the denominator : 5
RESULTANT FRACTION IS :  4 / 5
```

Program 11.4 Write a program that overloads the + operator so that it can add a specified number of days to a given date.

```
Dict = {1:31, 3:31, 4:30, 5:31, 6:30, 7:31, 8:31, 9:30, 10:31, 11:30, 12:31}
def chk_Leap_Year(year):
    if (year%4 == 0 and year%100 != 0) or (year%400 == 0):
        return 1
    else:
        return 0
class Date:
    def __init__(self):
        d = m = y = 0
    def get(self):
        self.d = int(input("Enter the day : "))
        self.m = int(input("Enter the month : "))
        self.y = int(input("Enter the year : "))
    def __add__(self, num):
        self.d += num
        if self.m !=2:
            max_days = Dict[self.m]
        elif self.m == 2:
            isLeap = chk_Leap_Year(self.y)
            if isLeap == 1:
                max_days = 29
            else:
                max_days = 28
        while self.d > max_days:
            self.d -= max_days
            self.m += 1
        while self.m > 12:
            self.m -= 12
            self.y += 1
```



```
    def display(self):
        print(self.d, "-", self.m,"-", self.y)
D = Date()
D.get()
num = int(input("How many days to add : "))
D + num
D.display()
```

OUTPUT

```
Enter the day : 25
Enter the month : 2
Enter the year : 2016
How many days to add : 10
6 - 3 - 2016
```

Program 11.5 Write a program that has an overloads the *, /, and > operators so that it can multiply, divide, and compare two objects of class `Fraction`.

```
Dict = {1:31, 3:31, 4:30, 5:31, 6:30, 7:31, 8:31, 9:30, 10:31, 11:30, 12:31}
def chk_Leap_Year(year):
    if (year%4 == 0 and year%100 != 0) or (year%400 == 0):
        return 1
    else:
        return 0
class Date:
    def __init__(self):
        d = m = y = 0
    def get(self):
        self.d = int(input("Enter the day : "))
        self.m = int(input("Enter the month : "))
        self.y = int(input("Enter the year : "))
    def __add__(self, num):
        self.d += num
        if self.m !=2:
            max_days = Dict[self.m]
        elif self.m == 2:
            isLeap = chk_Leap_Year(self.y)
            if isLeap == 1:
                max_days = 29
            else:
                max_days = 28
        while self.d > max_days:
            self.d -= max_days
            self.m += 1
        while self.m > 12:
            self.m -= 12
            self.y += 1
```

```
    def display(self):
        print(self.d, "-", self.m,"-", self.y)
D = Date()
D.get()
num = int(input("How many days to add : "))
D + num
D.display()
```

OUTPUT

```
Enter the numerator : 2
Enter the denominator : 3
Enter the numerator : 4
Enter the denominator : 9
F1 > F2 True
F1 * F2 IS :  8 / 27
F1 / F2 IS :  3 / 2
```

Program 11.6 Write a program that overloads the + operator so that it can add two objects of class Binary.

```
class Binary:
    number = []
    def set(self, bnum):
        self.number = bnum
    def display(self):
        print(self. Number)
    def __add__(self, B):
        Temp = Binary()
        index = len(self.number)
        carry = []
        while len(Temp.number) != index:
            Temp.number.append(-1)
            carry.append(0)
        index -= 1
        while (index)>=0:
            if self.number[index] == 0 and B.number[index] == 0:
                Temp.number[index] = 0 + int(carry[index])
            if self.number[index] == 0 and B.number[index] == 1:
                Temp.number[index] = 1 + int(carry[index])
            if self.number[index] == 1 and B.number[index] == 0:
                Temp.number[index] = 1 + int(carry[index])
            if self.number[index] == 1 and B.number[index] == 1:
                Temp.number[index] = 0 + int(carry[index])
                carry[index-1] = 1
            if Temp.number[index] == 2:
                Temp.number[index] = 0
                if (index-1)>=0:
```



```
                    carry[index-1] = 1
            index -= 1
        return Temp
B1 = Binary()
B1.set([1,1,0,1,1])
B2 = Binary()
B2.set([0,1,1,0,1])
B3 = B1 + B2
B3.display()
```

OUTPUT

```
[0, 1, 0, 0, 0]
```

Program 11.7 Write a program to compare two Date objects.

```
class Date:
    def __init__(self):
        d = m = y = 0
    def get(self):
        self.d = int(input("Enter the day : "))
        self.m = int(input("Enter the month : "))
        self.y = int(input("Enter the year : "))
    def __eq__(self, D):
        Flag = False
        if self.d == D.d:
            if self.m == D.m:
                if self.y == D.y:
                    Flag = True
        return Flag
    def __lt__(self, D):
        Flag = False
        if self.y < D.y:
            if self.m < D.m:
                if self.d < D.d:
                    Flag = True
        return Flag
D1 = Date()
D1.get()
D2 = Date()
D2.get()
print("D1 == D2", D1 == D2)
print("D1 < D2", D1 < D2)
```

OUTPUT

```
Enter the day : 21
Enter the month : 3
```

Programming Tip: The __eq__ function gives NotImplemented as result when left hand argument does not know how to test for equality with given right hand argument.

```
Enter the year : 2017
Enter the day : 21
Enter the month : 3
Enter the year : 2017
D1 == D2 True
D1 < D2 False
```

Program 11.8 Write a program to overload the -= operator to subtract two Distance objects.

```
class Distance:
    def __init__(self):
        self.km = 0
        self.m = 0
    def set(self, km, m):
        self.km = km
        self.m = m
    def __isub__(self, D):
        self.m = self.m - D.m
        if self.m < 0:
            self.m += 1000
            self.km -= 1
        self.km = self.km - D.km
        return self
    def convert_to_meters(self):
        return (self.km*1000 + self.m)
    def display(self):
        print(self.km, "kms", self.m, "mtrs")
D1 = Distance()
D1.set(21, 70)
D2 = Distance()
D2.set(18, 123)
D1 -= D2
print("D1 - D2 = ")
D1.display(),
print("that is", D1.convert_to_meters(), "meters")
```

OUTPUT

```
D1 - D2 =  2 kms 947 mtrs that is 2947 meters
```

11.3 REVERSE ADDING

In a program, we have added a certain number of days to our `Date` object by writing `d + num`. In this case, it is compulsory that the class object will invoke the `__add__()`. But, to provide greater flexibility, we should also be able to perform the addition in reverse order, that is, adding a non-class object to the class object. For this, Python provides the concept of reverse adding. The function to do normal addition on `Date` object is discussed in the following example.

Example 11.4 Program to illustrate adding on `Date` object

```
def __add__(self, num):
        self.d += num
        if self.m !=2:
            max_days = Dict[self.m]
        elif self.m == 2:
            isLeap = chk_Leap_Year(self.y)
            if isLeap == 1:
                max_days = 29
            else:
                max_days = 28
        while self.d > max_days:
            self.d -= max_days
            self.m += 1
        while self.m > 12:
            self.m -= 12
            self.y += 1
```

Programming Tip: Special methods are used for performing operator overloading.

But, had we written the same statement as `num + d`, then the desired task would not have been performed. The simple reason for this is that the `__add__()` takes `self` as the first argument, so the + operator has to be invoked using the `Date` object. But this is not the case when you work with numbers. You can either write `10 + 20` or `20 + 10`, it means the same and the correct result is produced. So, we should also have the same result when we write `d + num` or `num + d`. Python has a solution to this. It has the feature of reverse adding. As you write the `__add__()` function, just write the `__radd__()` function which will do the same task.

Note To overload the += or −= operators, use the `__iadd__()` or `__isub__()` functions.

11.4 OVERRIDING `__getitem__()` AND `__setitem__()` METHODS

Python allows you to override `__getitem__()` and `__setitem__()` methods. We have already seen in Chapter 9 that `__getitem__()` is used to retrieve an item at a particular index. Similarly, `__setitem__()` is used to set value for a particular item at the specified index. Although they are well defined for built-in types like list, tuple, string, etc. but for user-defined classes we need to explicitly write their codes. Consider the program given below which has a list defined in a class. By default, Python does not allow you to apply indexes on class objects but if you have defined the `__getitem__()` and `__setitem__()` methods in the class, then you can simply work with indices as with any other built-in type as shown in the following example.

Example 11.5 Program that overrides `__getitem__()` and `__setitem__()` methods in a class

```
class myList:
    def __init__(self, List):
        self.List = List
    def __getitem__(self,index):
        return self.List[index]
```

```
    def __setitem__(self, index, num):
        self.List[index] = num
    def __len__(self):
        return len(self.List)
    def display(self):
        print(self.List)
L = myList([1,2,3,4,5,6,7])
print("LIST IS : ")
L.display()
index = int(input("Enter the index of List you want to access : "))
print(L[index])
index = int(input("Enter the index at which you want to modify : "))
num = int(input("Enter the correct number : "))
L[index] = num
L.display()
print("The length of my list is : ", len(L))
```

OUTPUT

```
LIST IS :  [1, 2, 3, 4, 5, 6, 7]
Enter the index of List you want to access : 3
4
Enter the index at which you want to modify : 3
Enter the correct number : 40
[1, 2, 3, 40, 5, 6, 7]
The length of my list is :  7
```

11.5 OVERRIDING THE in OPERATOR

We have seen that in is a membership operator that checks whether the specified item is in the variable of built-in type or not (like string, list, dictionary, tuple, etc.). We can overload the same operator to check whether the given value is a member of a class variable or not. To overload the in operator we have to use the function __contains__(). In the program given in the following example, we have created a dictionary that has name of the subjects as *key* and their maximum weightage as *value*. In the main module, we are asking the user to input a subject. If the subject is specified in our dictionary, then its maximum weightage is displayed.

Example 11.6 Program to override the in operator

```
class Marks:
    def __init__(self):
        self.max_marks = {"Maths":100, "Computers":50, "SST":100, "Science":75}
    def __contains__(self, sub):
        if sub in self.max_marks:
            return True
        else:
                return False
    def __getitem__(self, sub):
```



```
        return self.max_marks[sub]
    def __str__(self):
        return "The Dictionary has name of subjects and maximum marks allotted to them"
M = Marks()
print(str(M))
sub = input("Enter the subject for which you want to know extra marks : ")
if sub in M:
    print("Social Studies paper has maximum marks alloted = ", M[sub])
```

OUTPUT

```
The Dictionary has name of subjects and maximum marks allotted to them
Enter the subject for which you want to know extra marks : Computers
Social Studies paper has maximum marks alloted =  50
```

11.6 OVERLOADING MISCELLANEOUS FUNCTIONS

Python allows you to overload functions like long(), float(), abs(), and hex(). Remember that we have used these functions on built-in type variables to convert them from one type to another. We can use these functions to convert a value of one user-defined type (object) to a value of another type.

Example 11.7 Program to overload hex(), oct(), and float() functions

```
class Number:
    def __init__(self, num):
        self.num = num
    def display(self):
        return self.num
    def __abs__(self):
        return abs(self.num)
    def __float__(self):
        return float(self.num)
    def __oct__(self):
        return oct(self.num)
    def __hex__(self):
        return hex(self.num)
    def __setitem__(self, num):
        self.num = num
N = Number(-14)
print("N IS : ", N.display())
print("ABS(N) IS : ", abs(N))
N = abs(N)
print("Converting to float....., N IS : ", float(N))
print("Hexadecimal equivalent of N IS : ", hex(N))
print("Octal equivalent of N IS : ", oct(N))
```

```
OUTPUT
N IS :  -14
ABS(N) IS :  14
Converting to float....., N IS :  14.0
Hexadecimal equivalent of N IS :  0xe
Octal equivalent of N IS :  016
```

Let us take another example in which we have two classes for calculating the distance. One has distance specified in meters and the other has distance in kilometers. There are two functions–`km()` and `mts()`, which takes the argument of class `Distance` and then converts the distance into kilometers and meters respectively.

Example 11.8 Program to illustrate conversion of class objects

```
class Distance_m:
  def __init__(self, m):
    self.m = m
  def display(self):
    print("Distance in meters is : ", self.m)
def mts(D):
    return D.km*1000
class Distance_km:
  def __init__(self, km):
    self.km = km
  def display(self):
    print("Distance in kilometers is : ", self.km)
def km(D):
    return D.m/1000
Dm = Distance_m(12345)
Dm.display()
print("Distance in kilo metres = ", km(Dm))
Dkm = Distance_km(12.345)
Dkm.display()
print("Distance in metres = ", mts(Dkm))
```

```
OUTPUT
Distance in meters is :  12345
Distance in kilo metres =  12
Distance in kilometers is :  12.345
Distance in metres =  12345.0
```

11.7 OVERRIDING THE `__call__()` METHOD

The `__call__()` method is used to overload call expressions. The `__call__()` method is called automatically when an instance of the class is called. It can be passed to any positional or keyword arguments. Like other functions, the `__call__()` method also supports all of the argument-passing modes. The `__call__()` method can be declared as, `def __call__(self, [args...])`

Example 11.9 Program to overload the `__call__()` method

```
class Mult:
  def __init__(self, num):
    self.num = num
  def __call__(self, O):
    return self.num * O
x = Mult(10)
print(x(5))
```

```
OUTPUT
50
```

Summary

- The meaning of operators like `+`, `=`, `*`, `/`, `>`, `<`, etc. are pre- defined in any programming language. So, programmers can use them directly on built in data types to write their programs.
- Operator overloading allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can also be applied to user-defined data types.
- With operator overloading, a programmer is allowed to provide his own definition for an operator to a class by overloading the built-in operator.
- Operator overloading is also known as operator *ad hoc polymorphism* since different operators have different implementations depending on their arguments.
- The `__add__()` method returns the new combined object to the caller.
- By default, Python does not allow you to apply indexes on class objects but if you have defined the `__getitem__()` and `__setitem__()` in the class, then you can simply work with indices as with any other built-in type.

Glossary

Ad hoc polymorphism A specific case of polymorphism where different operators have different implementations depending on their arguments.

Membership operator An operator that checks whether the specified item is present in the instance of an object or not.

Operator Overloading Redefining the meaning of operators when they operate on class objects.

Exercises

Fill In The Blanks

1. _____ allows programmers to redefine the meaning of existing operators.
2. Operator overloading is also known as _____ polymorphism.
3. _____ is a specific case of polymorphism where different operators have different implementations
4. The name of the function to overload ** operator is _____.
5. The `__add__()` method returns _____.
6. To overload the *= operator you will use _____ function.