CHAPTER

4

OPERATORS IN PYTHON

The general purpose of a program is to accept data and perform some operations on the data. The data in the program is stored in variables. The next question is how to perform operations on the data. For example, when a programmer wants to add two numbers, he can simply type ' + ' symbol. This symbol performs the addition operation. Such symbols are called *operators*. Let's now learn more about these types of operators.

Operator

An operator is a symbol that performs an operation. An operator acts on some variables called *operands*. For example, if we write a + b, the operator ' + ' is acting on two operands 'a' and 'b'. If an operator acts on a single variable, it is called *unary* operator. If an operator acts on two variables, it is called *binary* operator. If an operator acts on three variables, then it is called *ternary* operator. This is one type of classification. We can classify the operators depending upon their nature, as shown below:

- Arithmetic operators
- Assignment operators
- Unary minus operator
- Relational operators
- Logical operators
- Boolean operators
- Bitwise operators

- Membership operators
- Identity operators

Arithmetic Operators

These operators are used to perform basic arithmetic operations like addition, subtraction, division, etc.

There are seven arithmetic operators available in Python. Since these operators act on two operands, they are called 'binary operators' also. Let's assume a = 13 and b = 5 and see the effect of various arithmetic operators in the Table 4.1:

Table 4.1: Arithmetic Operators in Python

Operator	Meaning	Example	Result
+	Addition operator. Adds two values.	a+b	18
-	Subtraction operator. Subtracts one value from another.	a-b	8
*	Multiplication operator. Multiplies values on either side of the operator.	a*b	65
/	Division operator. Divides left operand by the right operand.	a/b	2.6
%	Modulus operator. Gives remainder of division.	a%b	3
**	Exponent operator. Calculates exponential power value. a ** b gives the value of a to the power of b.	a**b	371293
//	Integer division. This is also called Floor division. Performs division and gives only integer quotient.	a//b	2

When there is an expression that contains several arithmetic operators, we should know which operation is done first and which operation is done next. In such cases, the following order of evaluation is used:

1. First parentheses are evaluated.
2. Exponentiation is done next.
3. Multiplication, division, modulus and floor divisions are at equal priority.
4. Addition and subtraction are done afterwards.
5. Finally, assignment operation is performed.

Let's take a sample expression: d = (x+y)*z**a//b+c. Assume the values of variables as: x=1; y=2; z=3; a=2; b=2; c=3. Then, the given expression d = (1+2)*3**2//2+3 will evaluate as:

1. First parentheses are evaluated. d = 3*3**2//2+3.
2. Exponentiation is done next. d = 3*9//2+3.

3. Multiplication, division, modulus and floor divisions are at equal priority. d = 27//2+3 and then d = 13+3.
4. Addition and subtraction are done afterwards. d = 16.
5. Finally, assignment is performed. The value 16 is now stored into 'd'.

Hence, the total value of the expression becomes 16 which is stored in the variable 'd'.

Using Python Interpreter as Calculator

It is interesting to know that we can use Python interpreter as a simple calculator that can perform basic arithmetic calculations. Open Python IDLE graphics window and type some arithmetic operations, as shown in Figure 4.1:

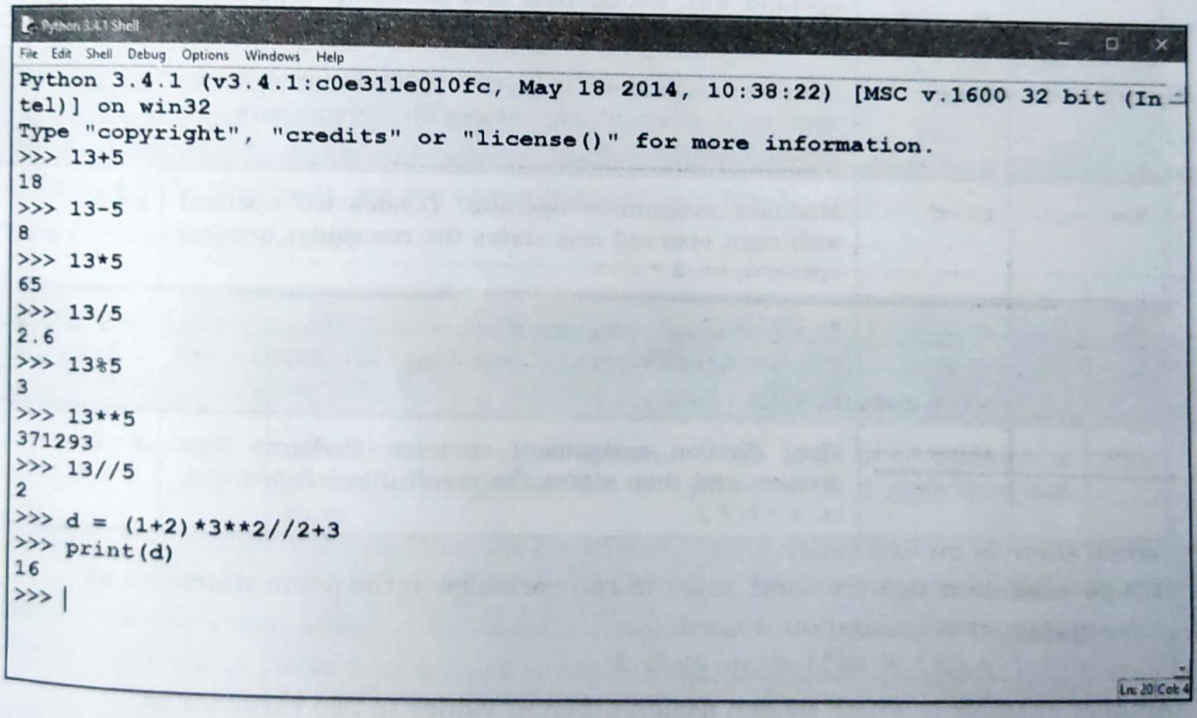


Figure 4.1: Python Interpreter as Calculator

Assignment Operators

These operators are useful to store the right side value into a left side variable. They can also be used to perform simple arithmetic operations like addition, subtraction, etc., and then store the result into a variable. These operators are shown in Table 4.2. In Table 4.2, let's assume the values x = 20, y = 10 and z = 5:

Table 4.2: Assignment Operators

Operator	Example	Meaning	Result
=	z = x+y	Assignment operator. Stores right side value into left side variable, i.e. x+y is stored into z.	z = 30
+=	z+=x	Addition assignment operator. Adds right operand to the left operand and stores the result into left operand, i.e. z = z+x.	z = 25
-=	z-=x	Subtraction assignment operator. Subtracts right operand from left operand and stores the result into left operand, i.e. z = z-x.	z = -15
*=	z*=x	Multiplication assignment operator. Multiplies right operand with left operand and stores the result into left operand, i.e. z = z *x.	z = 100
/=	z/=x	Division assignment operator. Divides left operand with right operand and stores the result into left operand, i.e. z = z/x.	z = 0.25
%=	z%=x	Modulus assignment operator. Divides left operand with right operand and stores the remainder into left operand, i.e. z = z%x.	z = 5
**=	z**=y	Exponentiation assignment operator. Performs power value and then stores the result into left operand, i.e. z = z**y.	z= 9765625
//=	z//=y	Floor division assignment operator. Performs floor division and then stores the result into left operand, i.e. z = z// y.	z = 0

It is possible to assign the same value to two variables in the same statement as:

```
a=b=1
print(a, b) # will display 1 1
```

Another example is where we can assign different values to two variables as:

```
a=1; b=2
print(a, b) # will display 1 2
```

The same can be done using the following statement:

```
a, b = 1, 2
print(a, b) # will display 1 2
```

A word of caution: Python does not have increment operator (++) and decrement operator (--) that are available in C and Java.

Unary Minus Operator

The unary minus operator is denoted by the symbol minus (-). When this operator is used before a variable, its value is negated. That means if the variable value is positive, it will be converted into negative and vice versa. For example, consider the following statements:

```
n = 10
print(-n) # displays -10

num = -10
num = - num
print(num) # displays 10
```

Relational Operators

Relational operators are used to compare two quantities. We can understand whether two values are same or which one is bigger or which one is lesser, etc. using these operators. These operators will result in True or False depending on the values compared, as shown in Table 4.3. In this table, we are assuming a =1 and b = 2.

Table 4.3: Relational Operators

Operator	Example	Meaning	Result
>	a>b	Greater than operator. If the value of left operand is greater than the value of right operand, it gives True else it gives False.	False
>=	a>=b	Greater than or equal operator. If the value of left operand is greater or equal than that of right operand, it gives True else False.	False
<	a<b	Less than operator. If the value of left operand is less than the value of right operand, it gives True else it gives False.	True
<=	a<=b	Less than or equal operator. If the value of left operand is lesser or equal than that of right operand, it gives True else False.	True
==	a==b	Equals operator. If the value of left operand is equal to the value of right operand, it gives True else False.	False
!=	a != b	Not equals operator. If the value of the left operand is not equal to the value of right operand, it returns True else it returns False.	True

Relational operators are generally used to construct conditions in if statements. For example,

```
a=1; b=2
if (a>b):
    print("Yes")
else:
    print("No")
```

will display 'No'. Observe the expression (a>b) written after if. This is called a 'condition'. When this condition becomes True, if statement will display 'Yes' and if it becomes False, then it will display 'No'. In this case, (1>2) is False and hence 'No' will be displayed.

Relational operators can be chained. It means, a single expression can hold more than one relational operator. For example,

```
x=15
10<x<20  # displays True
```

Here, 10 is less than 15 is True, and then 15 is less than 20 is True. Since both the conditions are evaluated to True, the result will be True.

```
10>=x<20 # displays False
```

Here, 10 is greater than or equal to 15 is False. But 15 is less than 20 is True. Since we get False and True, the result will be False.

```
10<x>20 # displays False
```

Here, 10 is less than 15 is True. But 15 is greater than 20 is False. Since we are getting True and False, the total result will be False. So, the point is this: in the chain of relational operators, if we get all True, then only the final result will be True. If any comparison yields False, then we get False as the final result. Thus,

```
1<2<3<4 # will give True

1<2>3<4 # will give False

4>2>=2>1 # will give True
```

Logical Operators

Logical operators are useful to construct compound conditions. A compound condition is a combination of more than one simple condition. Each of the simple condition is evaluated to True or False and then the decision is taken to know whether the total condition is True or False. We should keep in mind that in case of logical operators, False indicates 0 and True indicates any other number. There are 3 logical operators as shown in Table 4.4. Let's take x = 1 and y=2 in this table.

Table 4.4: Logical Operators

Operator	Example	Meaning	Result
And	x and y	And operator. If x is False, it returns x, otherwise it returns y.	2
Or	x or y	Or operator. If x is False, it returns y, otherwise it returns x.	1
Not	not x	Not operator. If x is False, it returns True, otherwise True.	False

Let's take some statements to understand the effect of logical operators.

```
x = 100
y = 200
print(x and y)  # will display 200

print(x or y)  # will display 100

print (not x) # will display False

x=1;  y=2; z=3
if(x<y and y<z): print('Yes')
else: print('No')
```

In the above statement, observe the compound condition after if. That is x <y and y<z. This is a combination of two simple conditions, x<y and y<z. When 'and' is used, the total condition will become True only if both the conditions are True. Since both the conditions became True, we will get 'Yes' as output. Observe another statement below:

```
if(x>y or y<z): print('Yes')
else: print('No')
```

Here, x>y is False but y<z is True. When using 'or' if any one condition is True, it will take the total compound condition as 'True' and hence it will display 'Yes'.

Boolean Operators

We know that there are two 'bool' type literals. They are True and False. Boolean operators act upon 'bool' type literals and they provide 'bool' type output. It means the result provided by Boolean operators will be again either True or False. There are three Boolean operators as mentioned in Table 4.5. Let's take x = True and y = False in Table 4.5:

Table 4.5: Boolean Operators

Operator	Example	Meaning	Result
and	x and y	Boolean and operator. If both x and y are True, then it returns True, otherwise False.	False
or	x or y	Boolean or operator. If either x or y is True, then it returns True, else False.	True
not	not x	Boolean not operator. If x is True, it returns False, else True.	False

In Figure 4.2, we can observe the effect of Boolean operators:

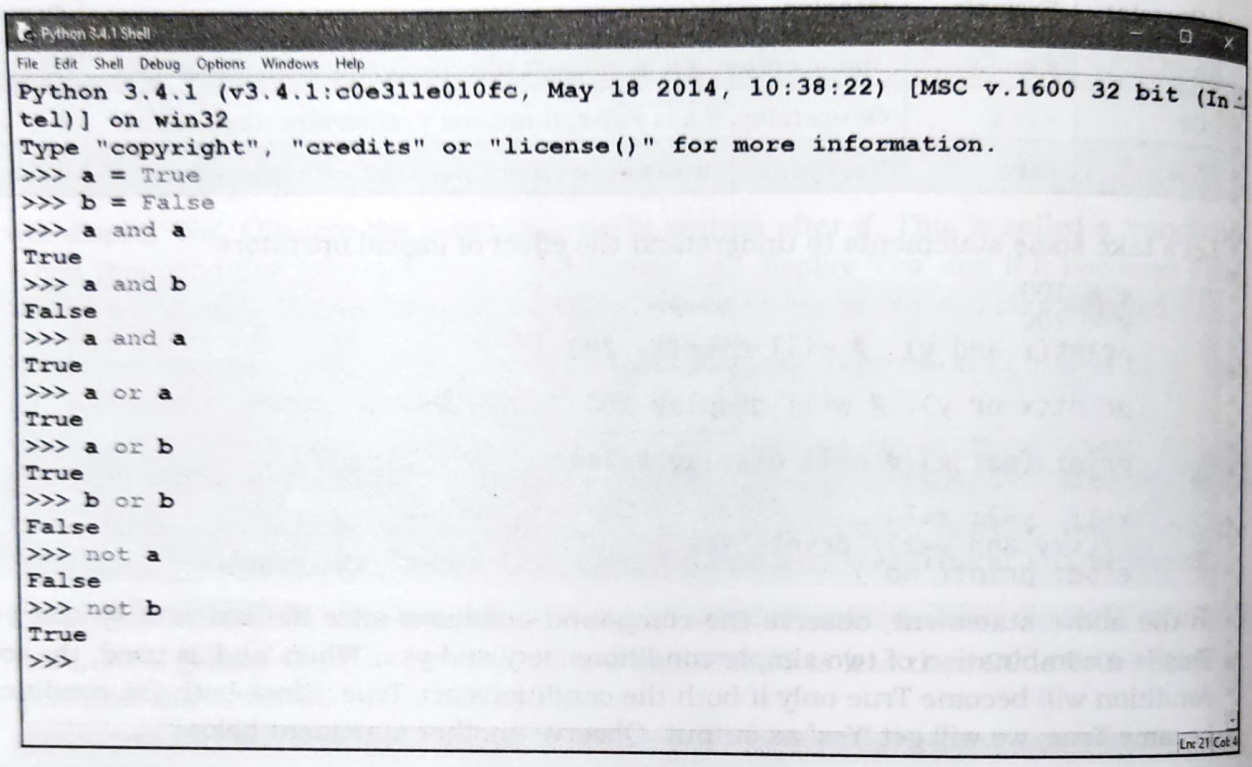


Figure 4.2: Boolean Operators and Their Usage

Bitwise Operators

These operators act on individual bits (0 and 1) of the operands. We can use bitwise operators directly on binary numbers or on integers also. When we use these operators on integers, these numbers are converted into bits (binary number system) and then bitwise operators act upon those bits. The results given by these operators are always in the form of integers.

We use decimal number system in our daily life. This number system consists of 10 digits from 0 to 9. We count all numbers using these 10 digits only. But in case of binary number system that is used by computers internally, there are only 2 digits, i.e. 0 and 1 which are called *bits* (binary digits). All values are represented only using these two bits. It is possible to convert a decimal number into binary number and vice versa.

Example 1: Converting 45 into binary number system.

Rule: Divide the number successively by 2 and take the remainders from bottom to top, as shown in Figure 4.3. The decimal number 45 is represented as 101101 in binary. If we use 8 bit representation, we can write it as: 0010 1101.

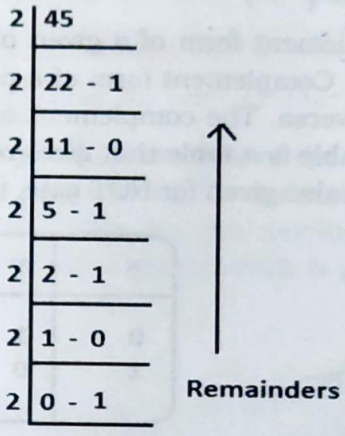


Figure 4.3: Converting into Binary

Example 2: Converting binary number 0010 1101 into decimal number.

Rule: Multiply the individual bits by the powers of 2 and take the sum of the products, as shown in Figure 4.4. Here, the sum is coming to 45. So 0010 1101 in binary is equal to 45 in decimal number system.

$$\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

$$0 + 0 + 32 + 0 + 8 + 4 + 0 + 1 = 45$$

Figure 4.4: Converting from Binary to Decimal

There are 6 types of bitwise operators as shown below:

- Bitwise Complement operator (~)
- Bitwise AND operator (&)
- Bitwise OR operator (|)
- Bitwise XOR operator (^)
- Bitwise Left shift operator (<<)
- Bitwise Right shift operator (>>)

We will discuss each of these operators one by one.

Bitwise Complement Operator (~)

This operator gives the complement form of a given number. This operator symbol is ~, which is pronounced as *tilde*. Complement form of a positive number can be obtained by changing 0's as 1's and vice versa. The complement operation is performed by NOT gate circuit in electronics. Truth table is a table that gives relationship between the inputs and the output. The truth table is also given for NOT gate, as shown in Figure 4.5:

NOT gate

x	y
0	1
1	0

truth table for NOT gate

Figure 4.5: NOT gate that Performs Complement of Bits

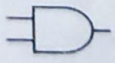
```
If x = 10, find the ~x value.
x = 10 = 0000 1010.
```

By changing 0's as 1's and vice versa, we get 1111 0101. This is nothing but -11(in decimal). So, ~x = -11.

Bitwise AND Operator (&)

This operator performs AND operation on the individual bits of numbers. The symbol for this operator is &, which is called *ampersand*. To understand the bitwise AND operation, see the truth table given in Figure 4.6:

```
x = 10 = 0000 1010
y = 11 = 0000 1011
-------------------
x & y = 0000 1010
```



AND gate

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

truth table for AND gate

Figure 4.6: AND Operation

From the truth table, we can conclude that by multiplying the input bits, we can get the output bit. The AND gate circuit present in the computer chip will perform the AND operation.

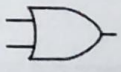
```
If x = 10, y = 11. Find the value of x&y.
x = 10 = 0000 1010.
y = 11 = 0000 1011.
```

From the truth table, by multiplying the bits, we can get x&y = 0000 1010. This is nothing but 10 (in decimal).

Bitwise OR Operator (|)

This operator performs OR operation on the bits of the numbers. The symbol of bitwise OR operator is |, which is called *pipe* symbol. To understand this operation, see the truth table given in Figure 4.7. From the table, we can conclude that by adding the input bits, we can get the output bit. The OR gate circuit, which is present in the computer chip will perform the OR operation.

```
x = 10 = 0000 1010
y = 11 = 0000 1011
-------------------
x | y = 0000 1011
```



OR gate

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

truth table for OR gate

Figure 4.7: OR Operation

```
If x = 10, y = 11, find the value of x|y.
x = 10 = 0000 1010.
y = 11 = 0000 1011.
```

The truth table shows that by adding the bits, we can get x|y = 0000 1011. This is nothing but 11 (in decimal).

Bitwise XOR Operator (^)

This operator performs *exclusive or (XOR)* operation on the bits of numbers. The symbol is ^, which is called *cap*, *carat*, or *circumflex* symbol. To understand the *XOR* operation, see the truth table given in Figure 4.8. From the table, we can conclude that when we have odd number of 1's in the input bits, we can get the output bit as 1. The *XOR* gate circuit of the computer chip will perform this operation.

```
x = 10 = 0000 1010
y = 11 = 0000 1011
-------------------
x ^ y = 0000 0001
```

XOR gate

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

truth table for XOR gate

Figure 4.8: XOR Operation

```
If x = 10, y = 11, find the value of x^y.
x = 10 = 0000 1010.
y = 11 = 0000 1011.
```

From the truth table, when odd number of 1's are there, we can get a 1 in the output. Thus, x^y = 0000 0001 is nothing but 1 (in decimal).

Bitwise Left Shift Operator (<<)

This operator shifts the bits of the number towards left a specified number of positions. The symbol for this operator is <<, read as *double less than*. If we write x<<n, the meaning is to shift the bits of x towards left n positions.

```
If x = 10, calculate x value if we write x<<2.
```

Shifting the value of x towards left 2 positions will make the leftmost 2 bits to be lost. The value of x is 10 = 0000 1010. Now, x<<2 will be 0010 1000 = 40 (in decimal). The procedure to do this is explained, as shown in Figure 4.9:

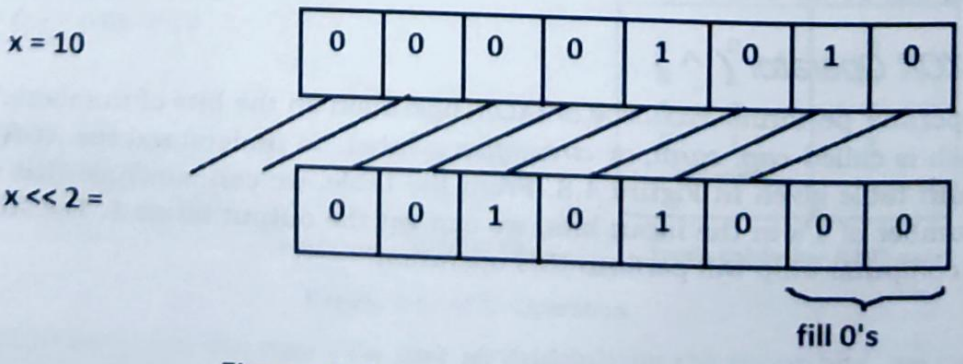


Figure 4.9: Shifting bits towards left 2 times

Bitwise Right Shift Operator (>>)

This operator shifts the bits of the number towards right a specified number of positions. The symbol for this operator is >>, read as *double greater than*. If we write x>>n, the meaning is to shift the bits of x towards right n positions.

>> shifts the bits towards right and also preserves the sign bit, which is the leftmost bit. Sign bit represents the sign of the number. Sign bit 0 represents a positive number and 1 represents a negative number. So, after performing >> operation on a positive number, we get a positive value in the result also. If right shifting is done on a negative number, again we get a negative value only.

```
If x = 10, then calculate x>>2 value.
```

Shifting the value of x towards right 2 positions will make the rightmost 2 bits to be lost. x value is 10 = 0000 1010. Now x>>2 will be: 0000 0010 = 2 (in decimal) (Figure 4.10).

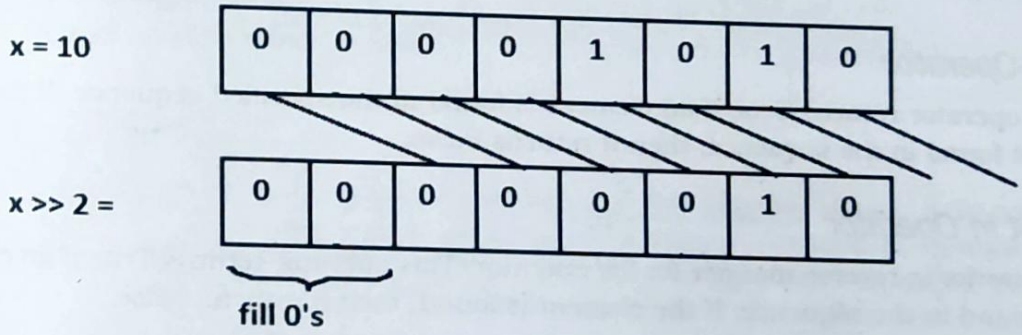


Figure 4.10: Shifting bits towards right 2 times

Let's check the effects of various bitwise operators so far discussed. We will use IDLE window and type the Python statements and confirm the results, as shown in Figure 4.11:

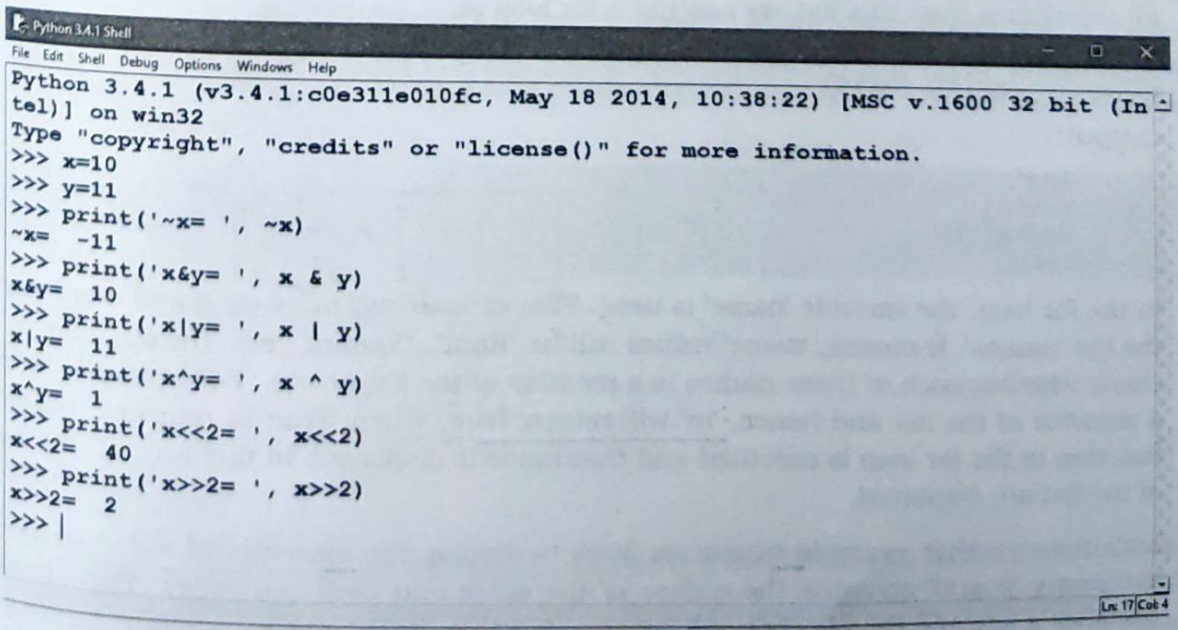


Figure 4.11: Using the Python Bitwise Operators

Membership Operators

The membership operators are useful to test for membership in a sequence such as strings, lists, tuples or dictionaries. For example, if an element is found in the sequence or not can be asserted using these operators. There are two membership operators as shown here:

- in
- not in

The in Operator

This operator returns True if an element is found in the specified sequence. If the element is not found in the sequence, then it returns False.

The not in Operator

This works in reverse manner for 'in' operator. This operator returns True if an element is not found in the sequence. If the element is found, then it returns False.

Let's take a group of strings in a list. We want to display the members of the list using a for loop where the 'in' operator is used. The list of names is given below:

```
names = ["Rani", "Yamini", "Sushmita", "Veena"]
```

Here the list name is 'names'. It contains a group of names. Suppose, we want to retrieve all the names from this list, we can use a for loop as:

```
for name in names:
    print (name)
```

Output:

```
Rani
Yamini
Sushmita
Veena
```

In the for loop, the variable 'name' is used. This variable will store each and every value of the list 'names'. It means, 'name' values will be "Rani", "Yamini", etc. The operator 'in' will check whether each of these names is a member of the list or not. For example, "Rani" is a member of the list and hence, 'in' will return True. When 'True' is returned, the print() function in the for loop is executed and that name is displayed. In this way, all the names of the list are displayed.

Let's take another example where we want to display the elements of a dictionary. The dictionary 'postal' contains the names of the cities and their pin codes. The city name becomes a key and the pin code will become its value. For example,

```
postal = {'Delhi': 110001, 'Chennai': 600001, 'Kolkata': 700001,
'Bangalore':560001}
```

Now, we want to retrieve each key and its corresponding value. For this purpose, we take a variable 'city' in for loop. Every time in the for loop, 'city' will get the name of the city. To get the corresponding value, we can use postal[city]. So, the following for loop will display the cities and their pin codes from 'postal' dictionary:

```
for city in postal:
    print(city, postal[city])
```

Output:

```
Kolkata 700001
Bangalore 560001
Delhi 110001
Chennai 600001
```

Identity Operators

These operators compare the memory locations of two objects. Hence, it is possible to know whether the two objects are same or not. The memory location of an object can be seen using the id() function. This function returns an integer number, called the *identity number* that internally represents the memory location of the object. For example, id(a) gives the identity number of the object referred by the name 'a'. See the following statements:

```
a = 25
b = 25
```

In the first statement, we are assigning the name (or identifier) 'a' to the object 25. In the second statement, we are assigning another name 'b' to the same object 25. In Python, everything is considered as an object. Here, 25 is the object for which two names are given. If we display an identity number of these two variables, we will get same numbers as they refer to the same object.

```
id(a)
1670954952
id(b)
1670954952
```

There are two identity operators:

- is
- is not

The is Operator

The 'is' operator is useful to compare whether two objects are same or not. It will internally compare the identity number of the objects. If the identity numbers of the objects are same, it will return True; otherwise, it returns False.

The is not Operator

This is not operator returns True, if the identity numbers of two objects being compared are not same. If they are same, then it will return False.

The 'is' and 'is not' operators do not compare the values of the objects. They compare the identity numbers or memory locations of the objects. If we want to compare the value of the objects, we should use equality operator (==).

```
a = 25
b = 25
if(a is b):
    print("a and b have same identity")
else:
    print("a and b do not have same identity")
```

Output:

```
a and b have same identity
```

As another example, we will take two lists with 4 elements each as:

```
one = [1,2,3,4]
two = [1,2,3,4]
if(one is two):
    print("one and two are same")
else:
    print("one and two are not same")
```

Output:

```
one and two are not same
```

In the preceding example, the lists one and two are having same elements or values. But the output is "one and two are not same". The reason is this: 'is' operator does not compare the values. It compares the identity numbers of the lists. Let's see the identity numbers of these two lists by using the id() function:

```
id(one)
51792432
id(two)
51607192
```

Since both the lists are created at different memory locations, we have their identity numbers different. So, 'is' operator will take the lists 'one' and 'two' as two different lists even though their values are same. It means, 'is' is not comparing their values. We can use equality (==) operator to compare their values as:

```
if(one == two):
    print("one and two are same")
else:
    print("one and two are not same")
```

Output:

```
one and two are same
```

Operator Precedence and Associativity

An expression or formula may contain several operators. In such a case, the programmer should know which operator is executed first and which one is executed next. The sequence of execution of the operators is called *operator precedence*. The following table summarizes the operators according to their precedence. The table shows precedence in descending order. It means the operators which are having highest precedence are listed at the top of the table, shown in Table 4.6:

Table 4.6: Precedence of Operators in Python

Operator	Name
()	Parenthesis
**	Exponentiation
-, ~	Unary minus, Bitwise complement
*, / , // , %	Multiplication, Division, Floor division, Modulus
+ , -	Addition, Subtraction
<< , >>	Bitwise left shift, Bitwise right shift
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
>, >=, <, <= , == , !=	Relational (comparison) operators
= , %= , /= , //= , -= , += , *= , **=	Assignment operators
is , is not	Identity operators
in , not in	Membership operators
not	Logical not
or	Logical or
and	Logical and

Precedence represents the priority level of the operator. The operators with higher precedence will be executed first than of lower precedence.

Suppose an expression contains operators having same precedence, then which operator is executed first is another question. It means knowing whether the execution is from left to right or right to left. This is called *associativity*. 'Associativity' is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity in Python. Let's take an expression value = 3/2*4+3+(10/4)**3-2 to understand how these precedence rules can be applied. The final value of this expression will be 22.625, as shown in Table 4.7:

Table 4.7: Evaluation of an Expression

Expression	Explanation
value = 3/2*4+3+(10/4)**3-2	
value = 3/2*4+3+2.5**3-2	The expression in ( ) is evaluated first.
value = 3/2*4+3+15.625-2	Exponentiation ** is next.
value = 1.5*4+3+15.625-2 value = 6.0+3+15.625-2	* and / have equal precedence. They are evaluated from left to right (associativity). So, first / and then *.
value = 9.0+15.625-2 value = 24.625-2 value = 22.625	+ and – have equal precedence. They are evaluated from left to right (associativity). So, first + and then -.

Mathematical Functions

While operators are very handy when dealing with fundamental operations in a program, we can use built-in functions given in Python to perform various advanced operations. For example, to calculate square root value of a number, we need not develop any logic. We can simply use the sqrt() function that is already given in the 'math' module. For example, to calculate square root of 16, we can call the function as: sqrt(16). This function returns the positive square root value of 16, i.e. 4.0.

In Python, a module is a file that contains a group of useful objects like functions, classes or variables. 'math' is a module that contains several functions to perform mathematical operations. If we want to use any module in our Python program, first we should import that module into our program by writing 'import' statement. For example, to import 'math' module, we can write:

```
import math
```

Once this is done, we can use any of the functions available in math module. Now, we can refer to sqrt() function from math module by writing module name before the function as: math.sqrt().

```
x = math.sqrt(16)
```

Here, 'x' value will become 4.0.

We can also use import statement as:

```
import math as m
```

In this case, we are importing 'math' module and naming it as 'm' in our program. Hence, hereafter, 'm' represents 'math'. So to calculate square root of 16, we can write as:

```
x = m.sqrt(16)
```

Import math statement is useful to import all the functions from math module. On the other hand, if the programmer wants to import only one or two functions from math module, he can write as:

```
from math import sqrt
```

This will make only sqrt() function available to our program. The programmer can use only that function from the math module but not other functions in his program. Similarly, to import more than one function, one can write:

```
from math import factorial, sqrt
```

Here, the programmer is importing two functions by the names factorial() and sqrt(). The programmer can use these functions without using module name before them, as:

```
x = sqrt(16)
y = factorial(5)
```

Here, 'x' value will be 4.0 and 'y' value will be 120. The following table (Table 4.8) summarizes important mathematical functions from math module. Please note that these functions cannot be used with complex numbers. To use these functions with complex numbers, a separate module by the name 'cmath' is available in Python. Also, Table 4.9 gives the available constants in math module.

Table 4.8: Important Math Functions

Function	Description
ceil(x)	Raises x value to the next higher integer. If x is integer, then same value is returned. Ex: ceil(4.5) gives 5
floor(x)	Decreases x value to the previous integer value. If x integer, then same value is returned. Ex: floor(4.5) gives 4
degrees(x)	Converts angle value x from radians into degrees. Ex: degrees(3.14159) gives 179.9998479605043
radians(x)	Converts x value from degrees into radians. Ex: radians(180) gives 3.141592653589793
sin(x)	Gives sine value of x. Here x value is given in radians. Ex: sin(0.5) gives 0.479425538604203

Function	Description
cos(x)	Gives cosine value of x. Here x value is given in radians. Ex: cos(0.5) gives 0.8775825618903728
tan(x)	Returns tangent value of x, given x value in radians. Ex: tan(0.5) gives 0.5463024898437905
exp(x)	Returns exponentiation of x. This is same as e **x. Ex: exp(0.5) returns 1.6487212707001282
fabs(x)	Gives the absolute value (or positive quantity) of x. Ex: fabs(-4.56) gives 4.56
factorial(x)	Returns factorial value of x. Raises 'ValueError' if x is not integer or is negative. Ex: factorial(4) gives 24
fmod(x, y)	Returns remainder of division of x and y. fmod() is preferred to calculate modulus for float values than '%' operator. '%' operator works well for integer values. Ex: fmod(14.5, 3) gives 2.5
fsum(values)	Returns accurate sum of floating point values. Ex: fsum([1.5, 2.4, -3.3]) returns 0.6000000000000001
modf(x)	Returns float and integral parts of x. Ex: modf(2.56) gives (0.56, 2.0)
log10(x)	Returns base-10 logarithm of x. Ex: log10(5.2345) gives 0.7188752041406328
log(x, [, base])	Returns the natural logarithm of x of specified base. Ex: log(5.5, 2) gives 2.4594316186372973
sqrt(x)	Returns positive square root value of x. Ex: sqrt(49) gives 7.0
pow(x, y)	Raises x value to the power of y. Ex: pow(5, 3) returns 125.0

Function	Description
gcd(x, y)	Gives greatest common divisor of x and y. Available from Python 3.5 onwards. Ex: gcd(25, 30) gives 5
trunc(x)	The real value of x is truncated to integer value and returned. Ex: trunc(15.5676) gives 15
isinf(x)	Returns True if x is a positive or negative infinity, and False otherwise. Ex: num = float('Inf') # num is a float number that indicates infinity. Isinf(num) gives True
isnan(x)	Returns True if x is a NaN (not a number), and False otherwise. Ex: num = float('NaN') # convert NaN into a float representation. isnan(num) gives True

Table 4.9: Constants in Math Module

Constant	Description
pi	The mathematical constant π = 3.141592..., with high precision.
e	The mathematical constant e = 2.718281..., with high precision.
inf	A floating-point positive infinity. (For negative infinity, use -math.inf.) Equivalent to the output of float('inf')
nan	A floating-point "not a number" (NaN) value. Equivalent to the output of float('nan').

Let's write a Python program to calculate the area of a circle. We know that the formula to calculate the area of a circle is pi * r *r. This logic is used in our program. Of course, to refer to the 'pi' value, we can use the constant in 'math' module. Hence, we should import 'math' module.

This program is executed in all the three environments: using IDLE window, using command line window and from system prompt. Let's see how to execute the program in these three environments.

Using IDLE Window

Click on 'Python IDLE window' icon available at task bar. This will open the Python Shell window where we should type the program. When last statement is typed and Enter is pressed, the result will be displayed, as shown in Figure 4.12:

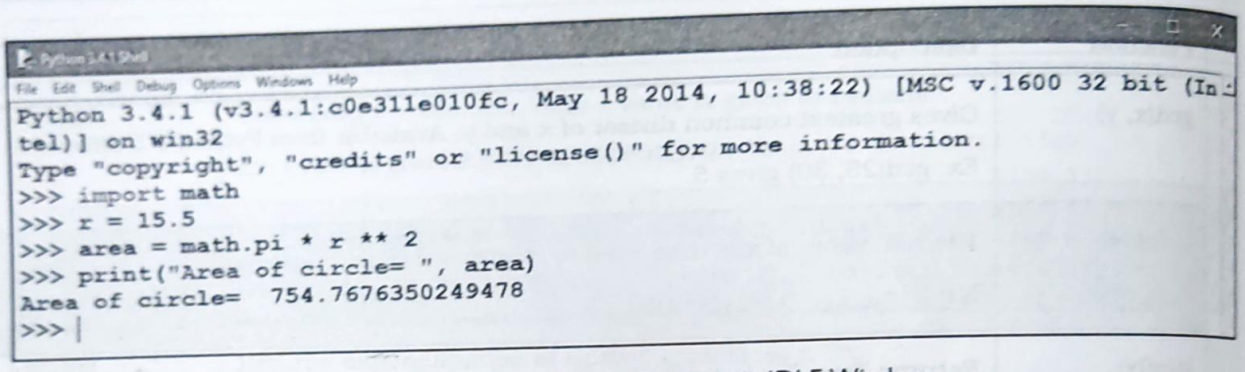


Figure 4.12: Executing the Program in IDLE Window

Using Command Line Window

Click on 'Python command line window' icon available at task bar. This will open the Python command line window where we should type the program. When last statement is typed and Enter is pressed, the result will be displayed, as shown in Figure 4.13:

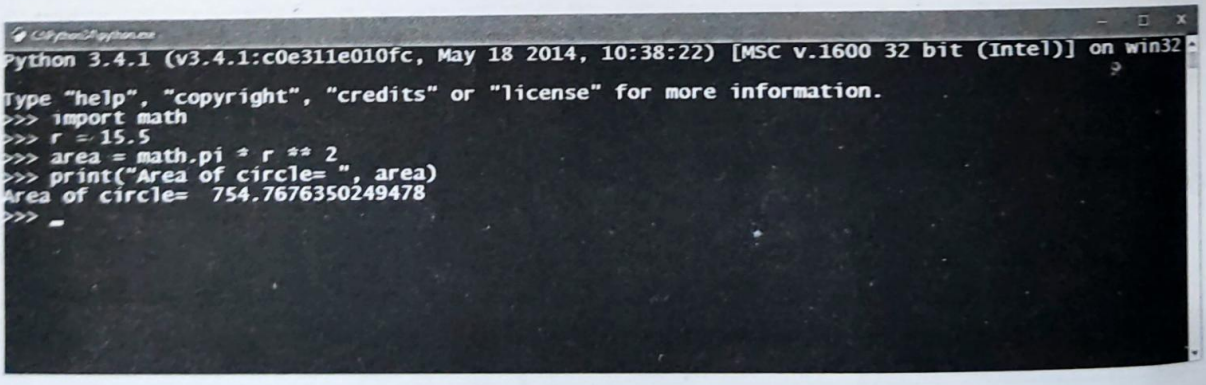


Figure 4.13: Executing the Program in Command Line Window

Executing at System Prompt

Open a text editor like Notepad or Edit Plus and then type the program. Save the program as "circle.py" in a directory. Open command prompt window and go to that directory where the program is stored. Then invoke python interpreter by typing:

```
F:\PY>python area.py
```

Then the result is displayed, as shown in Figure 4.14:

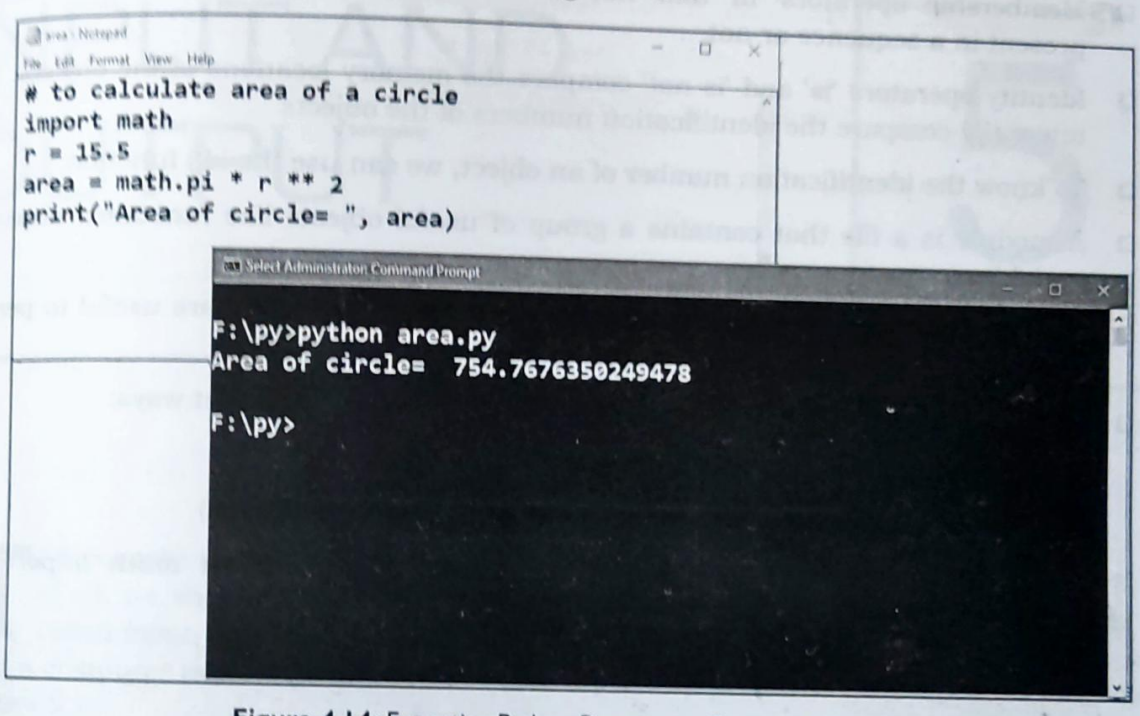


Figure 4.14: Executing Python Program at Command Prompt

Points to Remember

- An operator is a symbol that performs an operation.
- An operator acts on variables which are called operands.
- It is possible to use Python interpreter as a calculator.
- Arithmetic operators perform addition, subtraction, multiplication, division, floor division, modulus, and exponentiation operations.
- Assignment operators are useful to assign a value to the left side variable.
- Unary minus operator is useful to negate a value.
- Relational operators are useful to compare two quantities. They are used to create simple conditions which may evaluate to True or False.
- Logical operators are useful to create compound conditions. A compound condition is a combination of more than one simple condition.
- Boolean operators act on 'bool' type literals. They give again 'bool' type result.
- Bitwise operators act on individual bits (1 and 0) of the numbers.

- Membership operators 'in' and 'not in' are useful to test whether an element is present in a sequence or not.
- Identity operators 'is' and 'is not' compare the memory locations of the objects. They internally compare the identification numbers of the objects.
- To know the identification number of an object, we can use the id() function.
- A module is a file that contains a group of useful objects like functions, classes or variables.
- The 'math' module in Python contains several functions which are useful to perform various mathematical calculations.
- To import a module into a Python program, we can use 3 different ways:
  - import modulename (Ex: import math)
  - import modulename as anothername (Ex: import math as m)
  - from modulename import object1, object2, ... (Ex: from math import sqrt, factorial)