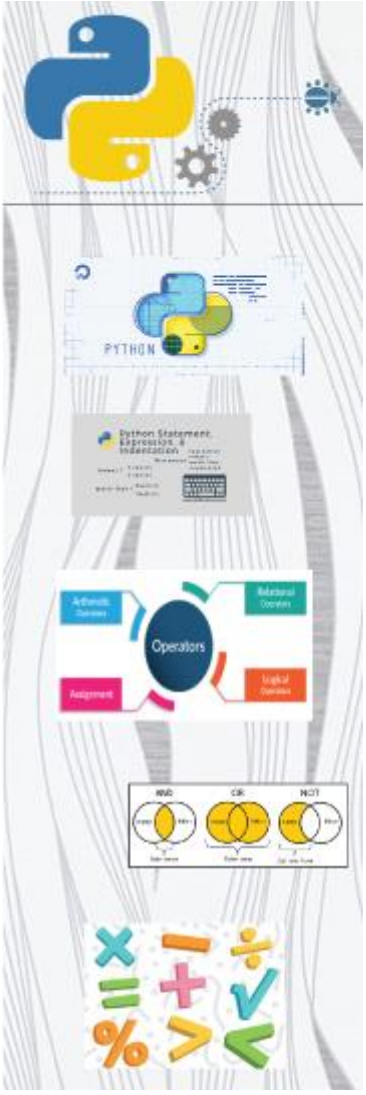
Chapter 3

Expression & Operators



In this chapter, you will learn about

- About Expressions & Operators?
- Python Operators
- Assignment Operators
- Arithmetic Operators
- Relational Operators
- Numeric Comparison
- Conditional Logical Operators
- Bitwise Operators
- Increment & Decrement Operators
- Assignment Operators
- Conditional Operators
- Order of Operator Precedence

Table of Contents

❖ Expressions & Operators

In Python, Expression is the process of any computation which yields a value. In expressions discussions, we often use the term evaluation. For example, we say that an expression evaluates to a certain value. Usually the final value is the only reason for evaluating the expression.

In any programming languages, an operator is a special symbol that specifies a certain process or action is carried out. In programming languages, Operators are taken from mathematics. Developers work with data. The operators are used to process data. An operand is one of the inputs (arguments) of an operator. In any programming languages, the operator instructs the computer what actions to perform. All operators cause actions to be performed on operands. An operand can be a literal, a variable or an expression.

In Python, expressions are the building blocks for computing results. These expressions are built by combining variables and operators together into statements. In Python, an expression is a sequence of operators and operands. This clause defines the syntax, order of evaluation of operands and operators, and meaning of expressions. A Python expression is a build made up of variables, operators, and method invocations, which are constructed according to the syntax of the Python language which evaluates to a single value.

Python provides operators for composing arithmetic, relational, logical, bitwise, and conditional expressions. It also provides operators which produce useful side effects, such as assignment, increment, and decrement.

❖ Python Operators

An *operator* is a symbol that causes Python to take an action. Python provides four types of built-in operators. They are

1. Unary Operators
2. Binary Operators
3. Ternary Operators
4. Other Operators

Unary operators affect a **single expression**. A unary operator is an operator that takes a single operand in an expression or a statement. The unary operators in Python are +, -, !, ~, ++, -- and the cast operator.

Binary operators require **two expressions** to produce a result.

The ternary operator has **three expressions**.

The other operators are represented using Python keywords.

❖ Assignment Operators

There are two types of assignment operators. They are:

1. Simple Assignment Operator
2. Compound Assignment Operator

An *assignment expression* stores a value in the variable designated by the left operand. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed.

The value of the right hand side may be

A literal value	x = 5
Another variable having a value	y = x
Any legal expression that yields a value	z = (5 + 5) * (2 * y);

Name	Operator	Type	Example	Example result
Simple assignment	=	Binary	n = 20	The variable n contains 20.

Simple Assignment Operator

The simple assignment operator has the following form:

<variable> = *<expression>*

The operator stores the value of the right operand *expression* in the variable designated by the left operand *variable*. The value of the expression on *RHS* is placed in the memory location named on the *LHS*.

The left operand must be a modifiable. The type of an assignment operation is the type of the left operand. You can assign multiple variables the same value with one statement:

x = y = z = 5

Correct Assignment Statements

<variable> = <expression>

area = 476

radius = 23

Incorrect Assignment Statements:

<expression> = <variable>

3 = w

w + 3 = h

w + h = 43

Complex Assignment

Python allows you to assign multiple data type literals to different variables. For example

a, b, c = 10, ‘Wisen’, 20

10 is assigned to the first variable a.

WIsen is assigned to the second variable b.

20 is assigned to the third variable c.

❖ Arithmetic Operator

Python supports two types of arithmetic operators. They are

1. Unary Arithmetic Operators
2. Binary Arithmetic Operators

Unary Arithmetic Operators

Python provides two basic unary arithmetic operators. These are summarized in below table. If you remember, unary operators are operators that only take one operand.

Name	Operator	Operator Type	Example	Example result
Unary Plus	+	Unary	+2	2
Unary Minus	-	Unary	-3	-3

The unary arithmetic plus operator determines the value of the operand. In other words, +2 = 2, and +m = m.

The unary arithmetic minus operator determines the operand multiplied by -1. In other words, if n = 3, -n = -3.

Both of these unary arithmetic operators should be placed immediately preceding the operand (eg. -m, not - m).

Do not confuse the unary minus operator with the binary subtraction operator, which uses the same symbol. For example, in the expression m = 7 - -2;, the first minus is the subtraction operator, and the second is the unary minus operator.

❖ Binary Arithmetic Operators

Python provides five basic binary arithmetic operators.

1. Addition Operator
2. Subtraction Operator
3. Multiplication Operator
4. Division Operator
5. Modulus Operator

The binary arithmetic operators are summarized in below table.

Name	Operator	Operator Type	Example	Example result
Addition	+	Binary	2 + 5	7
Subtraction	-	Binary	5 – 2	3
Multiplication	*	Binary	5 * 3	15
Division	/	Binary	10 / 2	5
Floor Division	//	Binary	9 // 2	4
Modulus	%	Binary	11 / 3	2
Power	**	Binary	2 ** 3	8

The // (floor division) operator yield the quotient.

The % (Modulus) operator yield the reminder.

❖ Relational Operators

Python provides six relational operators for comparing values.

The relational operators compare two operands and determine the validity of a relationship. Relational operators evaluate to true or false.

1. **Equality Operator**

   This operator determines true if the value of the left operand is equal to the value of the right operand. Otherwise false.

The equality operator (==) should not be confused with the assignment (=) operator.

2. **Inequality Operator**

   This operator determines true if the value of the left operand is not equal to the value of the right operand. Otherwise false.

3. **Less Than Operator**

   This operator determines true if the value of the left operand is less than the value of the right operand. Otherwise false.

4. **Less Than or Equal Operator**

   This operator determines true if the value of the left operand is less than or equal the value of the right operand. Otherwise false.

5. **Greater Than**

   This operator determines true if the value of the left operand is greater than the value of the right operand. Otherwise false.

6. **Greater Than or Equal**

   This operator determines true if the value of the left operand is greater than or equal the value of the right operand. Otherwise false.

Relational Operators are summarized in the below table

Name	Operator	Type	Example	Example result
Equality	==	Binary	9 == 9	True
Inequality	!=	Binary	9 != 9	False
Less Than	<	Binary	7 < 9	True
Less Than or Equal	<=	Binary	9 <= 9	True
Greater Than	>	Binary	7 > 9	False
Greater Than or Equal	>=	Binary	3 >= 9	False

❖ Numeric Comparison

Some real value numbers cannot be represented exactly in spite of how much precision is available. In the real number system, the fraction 1/3 cannot be represented exactly in decimal notation. Likewise, most decimal fractions (for example, .1) cannot be represented exactly in base 2 or base 16 numbering systems. This is the principle reason for difficulty in storing fractional numbers in floating-point representation.

Imprecision can cause problems with computations. Imprecision can also cause problems with comparisons.

Consider the following example

```
Line 1:   x = 1.0 / 3.0
Line 2:   print(x)
Line 3:   if(x == 0.33333333)
Line 4:             print("Both are Equals for 0.333333")
```

The Line 4 will not be executed at all.

❖ Conditional Logical Operators

Using logical operators, you can combine a series of comparisons into a single logical expression, so you end up needing just one if, virtually regardless of the complexity of the set of conditions, as long as the decision ultimately boils down to a choice between two possibilities(true or false).

Python provides logical operators for combining logical expression.

1. Logical Negation (not)
2. Conditional AND (and)
3. Conditional OR (or)

Logical Negation

Python provides the **not** (**logical NOT**, also called **logical negation**) operator to enable a programmer to "reverse" the meaning of a condition. Unlike the && and || binary operators, which combine two conditions, Logical *negation* is a unary operator, which negates the logical value of its single operand. If its operand is true it produces false, and if it is false it produces true.

The unary logical negation operator is placed before a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is false.

Logical NOT Operator truth table is summarized in the below table

Expression1	not Expression1
False	True
True	False

For example

if x has the value 10, the expression:

not(x > 5)

is false, because x > 5 is true.

You could also apply the not operator in an expression which evaluates Boolean values

not(income > expenditure)

Conditional Logical AND

The and (logical AND) operator indicates whether both operands are true. The and operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to false, the right operand is not evaluated.

The following example uses the short circuited (conditional) logical AND operator to avoid division by zero:

(y != 0) and (x / y == 2)

The expression x / y is not evaluated when y != 0 evaluates to false.

ERROR	Although 5 < x < 9 is a mathematically correct condition. But in Python, this expression does not evaluate as you might expect. To use in Python, you have to specify (5 < x and x < 9) to get the proper evaluation.

Logical AND Operator truth table is summarized in the below table

Expression1	Expression2	Expression1 and Expression2
False	False	False
False	True	False
True	False	False
True	True	True

On Integer Data Type

The & operator evaluates the bitwise logical AND of the two operands. Check Bitwise operator.

Conditional Logical OR

The or (logical OR) operator indicates whether either operand is true. The or operator guarantees left-to-right evaluation of the operands. If the left operand has a true value, the right operand is not evaluated. This is also called as conditional logical OR operator or "short-circuiting" logical OR operator.

Logical OR Operator truth table is summarized in the below table

Expression1	Expression2	Expression1 or Expression2
False	False	False
False	True	True
True	False	True
True	True	True

Like the relational operators, logical operator evaluate to true or false.

TIPS	In expressions using operator &&, if the separate conditions are independent of one another, make the condition most likely to be false the leftmost condition. In expressions using operator ||, make the condition most likely to be true the leftmost condition. This use of short-circuit evaluation can reduce a program's execution time.

❖ Bitwise Operator

The bitwise operators treat their operands as a series of individual bits rather than a numerical value. The bitwise operators are useful in programming hardware devices, where the status of a device is often represented as a series of individual flags (that is, each bit of a byte may signify the status of a different aspect of the device), or for any situation where you might want to pack a set of on-off flags into a single variable.

Python provides six bitwise operators for manipulating the individual bits in an integer values. Out of six, four are bitwise logical operators.

1. Bitwise Negation
2. Bitwise AND
3. Bitwise OR
4. Bitwise XOR
5. Left Shift
6. Right Shift

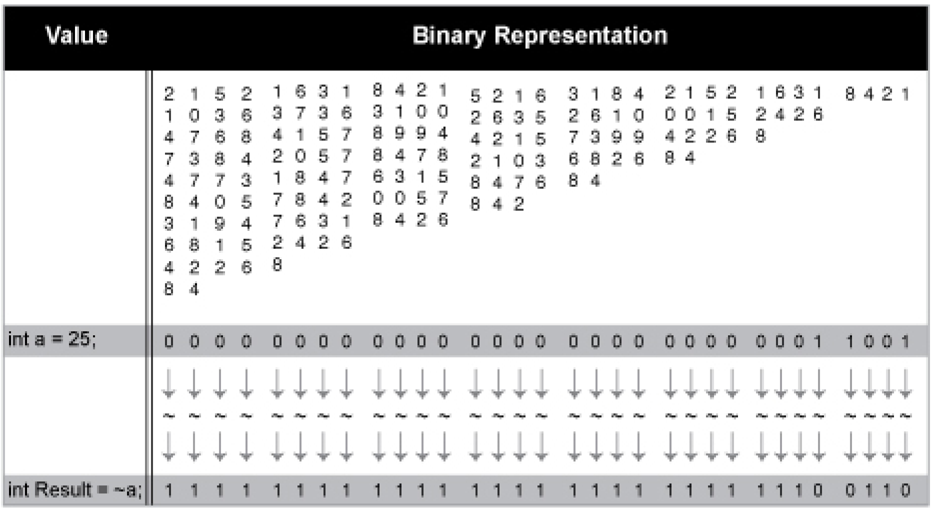
Bitwise Negation

The bitwise NOT, ~, takes a single operand for which it inverts the bits: 1 becomes 0, and 0 becomes 1.

The bitwise negation (NOT) is a three step process:

1. The operand is converted to binary.
2. The binary form is converted into its one’s complement.
3. The one’s complement is converted back to a numeric value.

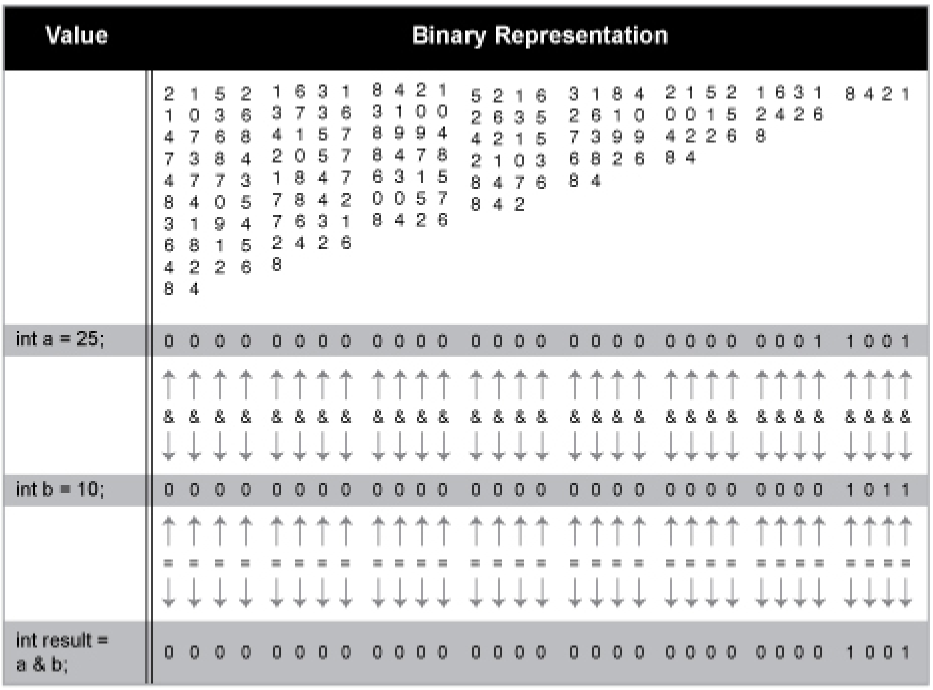
For instance



Value	Binary Representation																															
	2147483648	1073741824	536870912	268435456	134217728	67108864	33554432	16777216	8388608	4194304	2097152	1048576	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
int a = 25;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓	↓ ~ ↓
int Result = ~a;	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0

Bitwise AND

The bitwise AND, &, is a binary operator that combines corresponding bits in its operands in a particular way. If both corresponding bits are 1, the result is a 1 bit, and if either or both bits are 0, the result is a 0 bit.



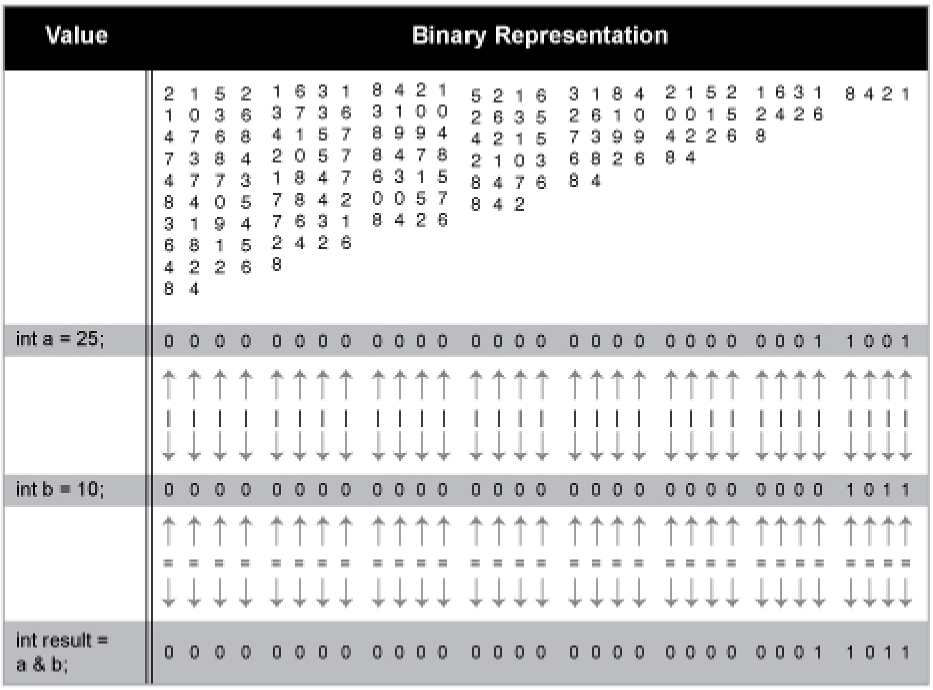
Value	Binary Representation																															
	2147483648	1073741824	536870912	268435456	134217728	67108864	33554432	16777216	8388608	4194304	2097152	1048576	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
int a = 25;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓	↑ & ↓
int b = 10;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓
int result = a & b;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1

In the above example, two bits (bit 0 and bit 3) contain a 1 in both 25 and 3. Because of this, every other bit of the resulting number is set to 0, making the result equal to 9.

Bitwise OR

The **|** (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

On Boolean Values



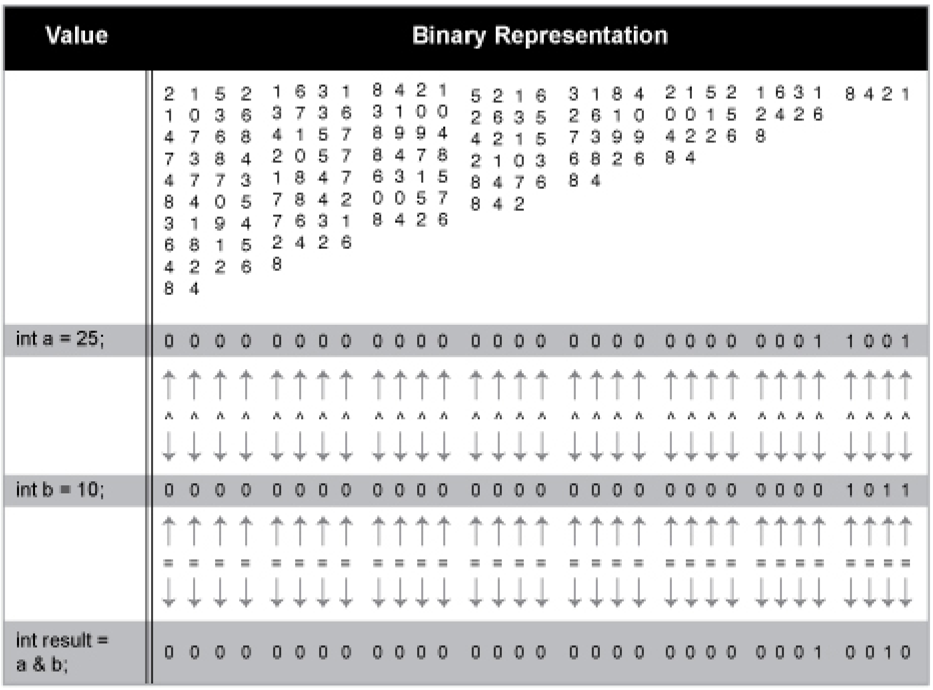
Value	Binary Representation																															
	2147483648	1073741824	536870912	268435456	134217728	67108864	33554432	16777216	8388608	4194304	2097152	1048576	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
int a = 25;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓	↑ | ↓
int b = 10;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓	↑ = ↓
int result = a & b;	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1

As you can see, four bits contain 1 in either number, so these are passed through to the result. The binary code 11011 is equal to 27.

Bitwise Exclusive OR

The bitwise exclusive OR operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1. It is indicated by a caret (^).

As you can see, two bits contain 1 in either number, so these are passed through to the result. The binary value is equal to 18.



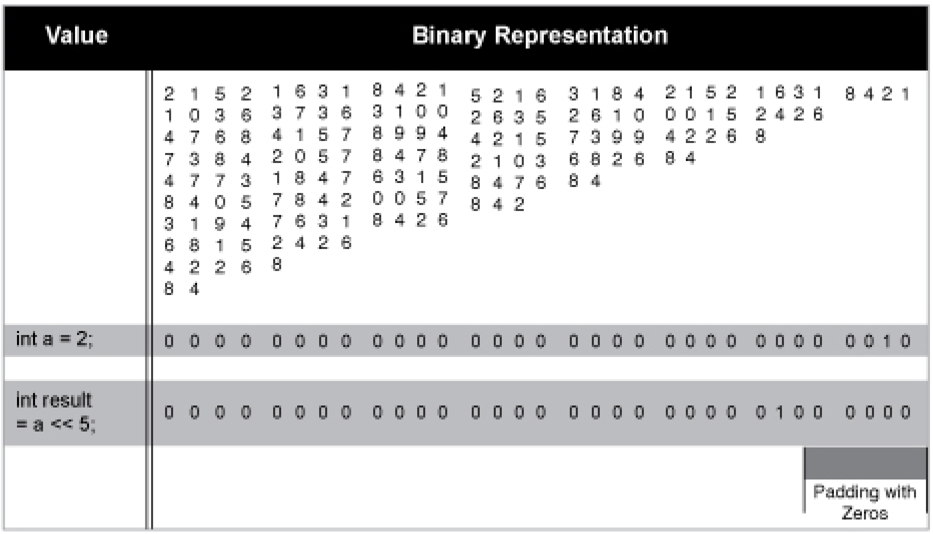
Value	Binary Representation
	2147483648 1073741824 536870912 268435456 134217728 67108864 33554432 16777216 8388608 4194304 2097152 1048576 524288 262144 131072 65536 32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
int a = 25;	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1
	↑ ^ ↓ ↓
int b = 10;	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1
	↑ = ↓ ↓
int result = a & b;	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0

Left Shift Operator

The bitwise shift operators move the bit values to. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted left.

For instance, consider the number 2 (which is equal to 10 in binary) and shifted it 5 bits to the left, the result is 64 (which is equal to 1000000 in binary):

```
myValue = 2;                #equal to binary 10
result = myValue << 5       #equal to binary 1000000 which is integer 64
```



Value	Binary Representation
	2147483648 1073741824 536870912 268435456 134217728 67108864 33554432 16777216 8388608 4194304 2097152 1048576 524288 262144 131072 65536 32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
int a = 2;	0000 0000 0000 0000 0000 0000 0000 0010
int result = a << 5;	0000 0000 0000 0000 0000 0000 0100 0000
	Padding with Zeros

Note that when the bits are shifted, five empty bits remain to the right of the number.

Right Shift Operator

The bitwise shift operators move the bit values to right. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted right.

For instance, consider the number 64 (which is equal to 1000000 in binary) and shifted it 5 bits to the right, the result is 2 (which is equal to 10 in binary):

```
myValue = 64               #equal to binary 1000000
result = myValue >> 5      #equal to binary 10 which is integer 2
```

❖ Compound Assignment Operator

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and store the result of that operation into the left operand.

An assignment operation is itself an expression whose value is the value stored in its left operand. An assignment operation can therefore be used as the right operand of another assignment operation. Any number of assignments can be concatenated in this fashion to form one expression. For example:

```
x = y = z = 10;            # Denotes: x = (y = (z = 10));
x = (y = z = 10) + 2       # Denotes: x = (y = (z = 10)) + 2;
```

This is equally applicable to other forms of assignment. For example:

```
x = 10
x += y = z = 10            # means: x = x + (y = z = 10);
```

Name	Operator	Type	Example	Example result

Addition assignment	+=	Binary	n = 20
			n += 10
				30

Subtraction assignment	-=	Binary	n = 20
			n -= 10
				10

Multiplication assignment	*=	Binary	n = 20 n *= 10	200
Division assignment	/=	Binary	n = 20; n /= 10	2
Modulus assignment	%=	Binary	n = 21 n %= 10	1
AND assignment	&=	Binary	n = 25; n &= 3	1
OR assignment	|=	Binary	n = 25 n |= 3	27
Exclusive OR assignment	^=	Binary	n = 25 n ^= 3	26
Shift Left assignment	<<=	Binary	n = 2 n >>= 5	64
Shift Right assignment	>>=	Binary	n = 64 n <<= 5	2

❖ Conditional Operators

The ternary operator, also called the conditional operator, is very unique. It effectively provides you with an “if…then” statement to use with operations.

The conditional operator takes three expressions. It has the general form:

expression1 ? expression2 : expression3

First *expression1* is evaluated, which is treated as a logical condition. If the result is true then *expression2* is evaluated and its value is the final result. Otherwise, *expression3* is evaluated and its value is the final result.

For example:

```
x = 1

y = 2

max = (x > y ? x : y)        # max receives 2
```

❖ Order of Precedence

In any programming languages, Precedence is very important. Python is no exception. The *order of precedence* indicates which operator should be evaluated first.

Operator	Process	Description
(expressions...), [expressions...], {key: value...}, {expressions...}	Left to Right	Tuple, List display, Dictionary, Set display
x[index], x[index:index], x(arguments...), x.attribute	Left to Right	Subscription, slicing, call, attribute reference
await x	Left to Right	The await expression

**	Right to Left	The exponent Operator
+x, -x, ~x	Left to Right	Positive, negative, bitwise NOT
*, @, /, //, %	Left to Right	Multiplication, matrix multiplication division, Modulus
+, -	Left to Right	Addition and subtraction
<<, >>	Left to Right	Shift Operators
&	Left to Right	Bitwise AND
^	Left to Right	Bitwise XOR
|	Left to Right	Bitwise OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Left to Right	Comparisons, including membership tests and identity tests
not x	Left to Right	Boolean NOT
and	Left to Right	Boolean AND
or	Left to Right	Boolean OR
if – else	Left to Right	Conditional expression
lambda	Left to Right	Lambda expression

Unless parentheses are used, the operations in an expression take place from left to right in the order of precedence
To specify the sequence of operations, you can use parentheses. Then, parenthetical expressions can be nested and are evaluated from inner to outer sets.

When an expression constitutes of multiple operators, Operator Precedence controls the order in which the individual operators are calculated.

In Python, the operator plus (+) and minus (-) has equal weight, so it evaluates from left to right. But this "left to right" evaluation can cause the problems. Change the plus into a minus in your code, and the minus into a plus. So change it to this:

```
a = 30
b = 20
c = 5
result = a - b + c
```

When Python evaluates the expression (a – b + c), the answer is 15 which is assigned to the variable result instead of 5. This is because of left to right evaluation. The Python performs the calculation as below

1. Left to right. So first a – b i-e 30 – 20 is 10
2. The above result + c i-e 10 + 5 is 15

What if you wanted a – the result of b + c? In this case, you have use parentheses like below

```
result = a – (b + c)\
```

Always use parentheses to inform Python exactly how you want your code to be evaluated.

❖ Summary

In Python, Expression is the process of any computation which yields a value. In any programming languages, an operator is a special symbol that specifies a certain process or action is carried out.

Python provides four types of built-in operators. They are

1. Unary Operators
2. Binary Operators
3. Ternary Operators
4. Other Operators

An assignment expression stores a value in the variable designated by the left operand. There are two types of assignment operators. They are:

1. Simple Assignment Operator
2. Compound Assignment Operator

Python supports two types of arithmetic operators. They are

1. Unary Arithmetic Operators
2. Binary Arithmetic Operators

Python provides two basic unary arithmetic operators.

1. Unary Plus
2. Unary Minus

Python provides five basic binary arithmetic operators.

1. Addition Operator
2. Subtraction Operator
3. Multiplication Operator
4. Division Operator
5. Modulus Operator

Python provides six relational operators for comparing values.

1. Equality Operator
2. Inequality Operator
3. Less Than Operator
4. Less Than or Equal Operator
5. Greater Than
6. Greater Than or Equal

Some real value numbers cannot be represented exactly in spite of how much precision is available.

Python provides logical operators for combining logical expression.

1. Logical Negation (!)
2. Logical AND (&)
3. Logical OR (|)
4. Conditional AND (&&)

5. Conditional OR (||)
6. Exclusive OR (^)

Python provides six bitwise operators for manipulating the individual bits in an integer values. Out of six, four are bitwise logical operators.

1. Bitwise Negation
2. Bitwise AND
3. Bitwise OR
4. Bitwise XOR
5. Left Shift
6. Right Shift