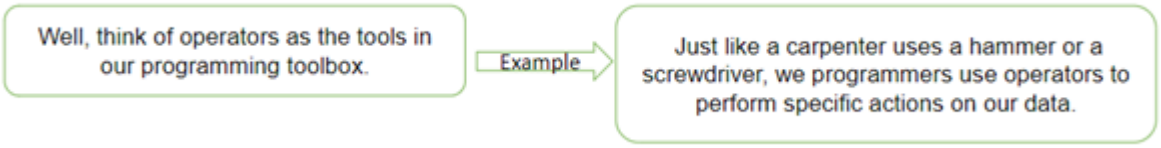
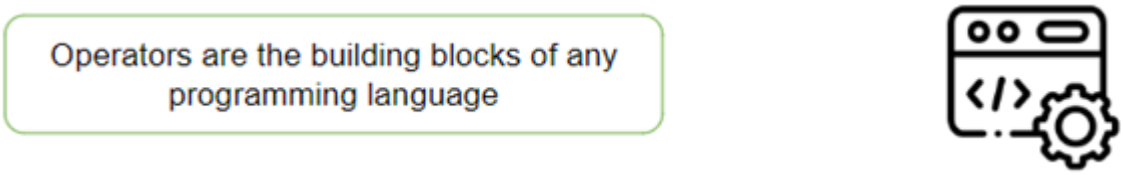
What are Operators?

Operators are the tools in our programming toolbox.

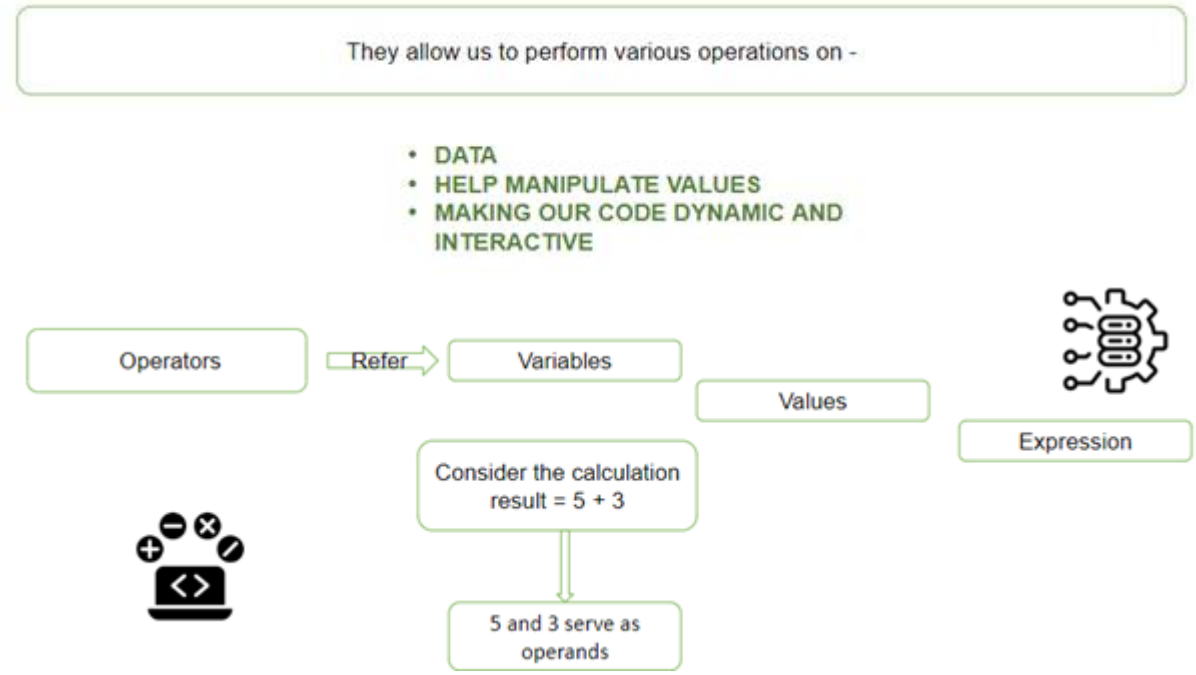


Operators are the building blocks of any programming language, including Python.



They are symbols or special characters that are used in programming languages and mathematics to perform various operations on operands. Operands can refer to variables, values, or expressions.

For instance, consider the calculation result = 5 + 3, where both 5 and 3 serve as operands, representing the numbers being added together, and the result is stored in the variable result.



Similarly, in the expression result = x * 2, x and 2 act as operands, illustrating the multiplication of the variable x by 2 to obtain a result.

In Python, we have different types of operators, each serving a unique purpose.

The first operator is the Arithmetic Operators.

1. Arithmetic Operators : These operators are used to perform basic mathematical calculations.

Let's take a closer look at the different Arithmetic Operators and see how they work. Here are some examples:

1. **Addition Operator (+): This operator is used to add two numbers together. For instance, if we have the variables x and y, we can use the addition operator to add them and store the result in another variable, like this:**

   x = 5

   y = 3

   result1 = x + y

   print(result1)

   # Output: 8

Here, we add the values of x which is equal to 5, and y which is equal to 3 using the addition operator (+) and store the result in the variable result1. When we print result1, we get the output 8.

1. **Subtraction Operator (-): This operator is used to subtract one number from another. Let's see an example:**

   a = 10

   b = 4

   result2 = a - b

   print(result2)

   # Output: 6

In this code snippet, we subtract the value of ‘b’ from ‘a’ using the subtraction operator (-) and store the result in the variable result2. The output of result2 is 6.

1. **Multiplication Operator (*): This operator is used to multiply two numbers. Consider the following example:**

   p = 6

   q = 7

   result3 = p * q

   print(result3)

   # Output: 42

In this code, we multiply the values of p and q using the multiplication operator (*) and store the result in the variable result3. The output of result3 is 42.

1. **Division Operator (/): This operator is used to divide one number by another.**

   m = 20

   n = 5

   result4 = m / n

   print(result4)

   # Output: 4.0

Here, we divide the value of m which is 20 by n which is equal to 5 using the division operator (/) and store the result in the variable result4. The output of result4 is 4.0. Notice that when we divide two integers, the result is automatically converted to a floating-point number.

1. **Modulo Operator (%): This operator returns the remainder when one number is divided by another.**

p = 17

q = 5

result5 = p % q

print(result5)

 # Output: 2

In this code snippet, we use the modulo operator (%) to find the remainder when p which is equal to 17 divided by q equal to 5. The output of result is 2.

Now, if we are interested in obtaining the quotient, which is the whole number result of the division, we can use the floor division operator (//).

1. **floor division operator (//): This operator returns the quotient of the division, discarding the remainder.**

p = 17

q = 5

result6 = p // q

print(result6)

 # Output: 3

17 two forward slash 5 results in 3 because it discards the remainder and returns only the whole number part of the quotient after division. Last from the Arithmetic operator we have the Exponentiation Operator(**)

1. **Exponentiation Operator(**): This double asterisk is your shortcut to raising a number to the power of another.**

base = 2

power = 3

result7 = base ** power

print(result7)

#Output:8

Here, the value of result7 will be 8 because 2 raised to the power of 3 is 8.

So these are the Arithmetic Operators in Python. They allow us to perform basic mathematical operations and are incredibly useful in many programming tasks.

2. Assignment operators : The Assignment Operator (=) is used to assign a value to a variable. It takes the value on the right-hand side and assigns it to the variable on the left-hand side.

x = 5

print(x)

Output: 5

In this code snippet, we use the assignment operator (=) 'equals to' to assign the value 5 to the variable x. When we print the value of x, we get the output 5.

In addition to the basic assignment operator, Python also provides compound assignment operators. These operators combine assignment with arithmetic operations. For example, we have += to add and assign, -= to subtract and assign, and *= to multiply and assign and some more.

x = 10

x += 5 # which is Equivalent to x = x + 5

print(x)

Output: 15

In this code, we first assign the value 10 to the variable x. Then, we use the compound assignment operator (+=), which adds the value on the right-hand side (5) to the existing value of x and assigns the result back to x. As a result, the value of x becomes 15.

- z = 7

  z -= 2 #which is Equivalent to z = z - 2

  print(z) # Output: 5

- w = 3

  w *= 4 # Equivalent to w = w * 4

  print(w) # Output: 12

- y=6

  y/=2 # Equivalent to y = y / 2

  print(y) #the output will be 3.0

- p = 17

  p%=2 # Equivalent to p = p %2, recall that p is the modulo operator

  print(p) # the output will be 1

- q = 17

  q//=2 # Equivalent to q = q //2, recall that // is the floor division operator

  print(q)

  #Output is 8

Assignment Operators are essential in programming as they allow us to store and manipulate data effectively.

3. Relational operators: The relational operators also known as Comparison Operators are operators that allow us to compare values and determine their relationship. They are extremely useful in decision-making and control flow statements .

These operators compare two values and return a Boolean result: either True or False. They help us determine if one value is greater than, less than, equal to, or not equal to another value.

1. **Greater than Operator (>): This operator checks if the value on the left-hand side is greater than the value on the right-hand side.**

   x = 5

   y = 3

   result = x > y

   print(result)

   # Output: True

In this code snippet, we use the greater than operator (>) to compare the values of x which is 5, and y which is equal to 3. The result is True because 5 is indeed greater than 3.

1. **Less than Operator (<): This operator checks if the value on the left-hand side is less than the value on the right-hand side.**

   a = 10

   b = 15

   result = a < b

   print(result)

   # Output: True

In this code, we use the less than operator (<) to compare the values of a (10) and b (15). The output is True because 10 is indeed less than 15.

1. **Equal to Operator (==): This operator checks if two values are equal.**

p = 5

q = 5

result = p == q

print(result)

 # Output: True

Here, we use the equal to operator (==) to compare the values of p which is 5 and q which is also 5. The result is True because both variables hold the same value.

1. **Not equal to Operator (!=)Exclamation and equal to sign: This operator checks if two values are not equal.**

   m = 8

   n = 3

   result = m != n

   print(result)

   # Output: True

In this code snippet, we use the not equal to operator (!=)Exclamation and equal to sign to compare the values of m and n. The output is True because 8 is not equal to 3.

1. **The greater than or equal to operator (>=): is a relational operator that checks if the value on the left is greater than or equal to the value on the right. It returns true if the condition is satisfied and false otherwise.**

   For:

   a=10

   b=5

result = a>=b

print(result)

Output is True (because 10 is greater than or equal to 5)

1. **Less than or equal to (<=) checks if the value on the left is less than or equal to the value on the right. It returns true if the condition is satisfied and false otherwise.**

   p = 5

   q = 10

   result = p <= q

   print(result)

Output: True (because 5 is less than 10)

4. Logical Operator

So a logical operator is like a special word or symbol that helps us make decisions based on true or false conditions in Python. It's similar to how we use words like "and," "or," and "not" in everyday language to combine or negate statements. In Python, logical operators allow us to combine and manipulate these true/false conditions to control the flow of our programs and make decisions based on multiple conditions. They help us determine whether certain conditions are met or not, and guide our code accordingly.

There are three types of logical operators in Python:

1. **"and" Operator: The ‘and’ operator returns True if both operands are true, otherwise, it returns False. It can be visualized as a gate that only opens when both conditions are satisfied.**

   x = 5

   y = 10

   result = (x > 0) and (y < 15)

print(result) # Output: True

In this code snippet, we use the ‘and’ operator to combine two conditions. The result is True because both conditions, x > 0 and y < 15, are True.

Let’s see it with a truth table which illustrates the output for all possible combinations of boolean values for the given logical operators. They are useful for understanding how logical expressions evaluate in different scenarios.

Truth Table for AND Operator (and):

X	Y	X and Y
True	True	True
True	False	False
False	True	False
False	False	False

1. **"or" Operator: The ‘or’ operator returns True if at least one of the operands is true, and returns False if both operands are false. It can be visualized as a gate that opens if either condition is satisfied.**

   a = 20

   b = 5

   result = (a > 25) or (b < 10)

   print(result)

   # Output: True

Here, we use the ‘or’ operator to combine two conditions. The output is True because at least one of the conditions, a > 25 or b < 10, is True.

Truth Table for OR Operator is as follows:

pp

X	Y	X or Y
True	True	True
True	False	True
False	True	True
False	False	False

1. **"not" Operator: This operator returns the opposite of the condition. If the condition is True, it returns False, and if the condition is False, it returns True.**

   p = 5

   result = not (p > 10)

   print(result) # Output: True

In this code snippet, we use the ‘not’ operator to negate the condition p > 10. Since p is 5 and the condition p > 10 is False, the output of the NOT operation is True.

Truth Table for NOT Operator is as follows:

X	not X
True	False
False	True

5. Identity Operator:

In Python, when you create a variable and assign a value to it, you're essentially creating an object in the computer's memory. Think of it like putting something in a box.

Imagine you're telling Python, "Hey, Python, I have this number 5, and I want to create a variable called ‘a’ and put this number in it." Python happily creates a small box in its memory, puts the number 5 inside, and labels the box with the name ‘a’. It's like having a box labeled "a" with the number 5 inside it.

Now, let's say you create another variable called ‘b’ and give it the value 5 as well. Since 5 is a simple number, Python might decide, "Hmm, I already have a box with the number 5 in it (the box labeled a), and ‘b’ is also supposed to have 5. Instead of making a new box, I'll just let ‘b’ point to the same box that ‘a’ is pointing to." So now, both ‘a’ and ‘b’ are pointing to the exact same box with the number 5 inside.

On the other hand, if you're dealing with something more complex like a list, Python might think, "Oh, this is more elaborate. I'll create a new box for ‘b’ even though it has the same value as ‘a’." So now, ‘a’ and ‘b’ each have their own separate boxes, even if the contents in the list or object are the same.

Now, the "identity operators" (‘ is’ and ‘ is not’) in Python help us check whether two variables are literally pointing to the exact same box in the computer's memory. It's like asking, "Are these two variables really, truly the same thing in the computer's brain?"

Here's a simple explanation:

1. **"is": If you say ‘a’ is ‘b’, you're asking Python, "Do these two variables (‘a’ and ‘b’) point to the exact same box in the computer's memory?" If they do, Python says "Yes," and if they don't, Python says "No."**
2. **"is not": If you say ‘a’ is not ‘b’, you're asking the opposite question. You're asking, "Are these two variables (a and b) NOT pointing to the exact same box in the computer's memory?" If they are not, Python says "Yes," and if they are, Python says "No."**

Let’s understand this with an example:

- Python:

  ```
  a = 5

  b = 5

  print(id(a)) # Print the memory address of a (show that both have same memory address)

  print(id(b)) # Print the memory address of b
  ```

print(a is b) # True, (First Show its True and then write the comment True)

print(a is not b) # False, (First Show its False and then write the comment False)

The id() function in Python returns the identity (memory address) of an object.

In this example, you'll likely see that id(a) and id(b) have the same value, indicating that both ‘a’ and ’b’ are pointing to the same memory location.

And therefore print(a is b) returns True, because Python used the same box for both ‘a’ and ‘b’

And print(a is not b) returns False, because a and b share the same box.

Now let’s see another example:

- Python:

  x = [1, 2, 3]

  y = [1, 2, 3]

  print(id(x)) # Print the memory address of x (show that both have diff memory address)

  print(id(y)) # Print the memory address of y

  print(x is y) # False

  print(x is not y) # True

  In this example, id(x) and id(y) will likely have different values, indicating that ‘x’ and ‘y’ are stored in different memory locations. Even though their contents are the same, Python created separate memory boxes for them because lists are more complex objects.

6. Membership Operator: The membership operators in Python are used to test whether a value exists in a sequence (like a list, tuple, set, or string) or not. They return True if the value is found in the sequence and False otherwise. Think of the membership operator as a way to check if something is part of a group or a collection.

Membership Operators are of two types:

- ‘ in’ operator
- ‘ not in’ operator

Let’s understand both with an example:

Imagine you have a list of your favorite fruits, like apples, bananas, and oranges. You use the ’ in’ operator to ask, "Is 'apples' in my list of favorite fruits?"

Python:

favorite_fruits = ["apples", "bananas", "oranges"]

print("apples" in favorite_fruits) # This will be True

print("grapes" in favorite_fruits) # This will be False

Now, let's say you're wondering if you have a fruit like watermelon in your favorite fruits list. You can use the ‘not in’ operator to ask, "Is 'watermelon' not in my list of favorite fruits?"

Python:

favorite_fruits = ["apples", "bananas", "oranges"]

print("watermelon" not in favorite_fruits) # This will be True

print("bananas" not in favorite_fruits) # This will be False

7. Bitwise Operator:

Bitwise operators are used to perform operations at the binary level, manipulating individual bits of binary numbers.

In simpler terms, computers store and process data in binary, which is a base-2 numeral system using only 0s and 1s. Bitwise operators allow us to perform operations on these binary representations.

Here are the basic bitwise operators:

1. **AND (&):**

   It compares each bit of two numbers. If both bits are 1, the resulting bit is 1. Otherwise, it's 0.

   0 & 0 = 0

   0 & 1 = 0

   1 & 0 = 0

   1 & 1 = 1

1. **OR (|):**

   If at least one of the bits is 1, the resulting bit is 1. It's 0 only when both bits are 0.

   0 | 0 = 0

   0 | 1 = 1

   1 | 0 = 1

   1 | 1 = 1

1. **XOR (^):**

   Exclusive OR. If the bits are different, the resulting bit is 1. If the bits are the same, it's 0.

   0 ^ 0 = 0

   0 ^ 1 = 1

   1 ^ 0 = 1

   1 ^ 1 = 0

1. **NOT (~):**

   It flips the bits. 0 becomes 1, and 1 becomes 0.

~1 = 0

~0 = 1

1. **Left Shift (<<):**

   Shifts the bits to the left by a specified number of positions. The vacant positions are filled with zeros.

   0010 << 1 = 0100

   Move each bit one position to the left. And Fill the vacant position on the right with a zero.

1. **Right Shift (>>):**

   Shifts the bits to the right by a specified number of positions. The vacant positions are usually filled with the sign bit (0 for positive numbers, 1 for negative numbers).

   0100 >> 1 = 0010

   Move each bit one position to the right. And Fill the vacant position on the left with a zero.

These operations might seem abstract, but they are fundamental in low-level programming, dealing with hardware, and various optimization tasks. They allow efficient data manipulation at the binary level, which is crucial in computer systems.

We explored various types of operators in Python, each serving a specific purpose. From performing basic arithmetic operations to making decisions, comparing values, and manipulating binary data, operators are essential tools in a programmer's toolkit.