Operators in general are used to perform operations on values and variables.
OPERATORS: These are the special symbols. Eg- + , * , /, etc.
OPERAND: It is the value on which the operator is applied.

```
# The Walrus operator (:=), introduced in Python 3.8, ****allows you to assign a value to a variable as part of an expression**
# This can make your code more concise and can eliminate the need for additional lines of code just to assign and then use a va
while (data := input("Enter something: ")) != "quit":
    print(data)

numbers = [4, 8, 15, 16, 23, 42]
filtered = [square for num in numbers if (square := num ** 2) > 100]
print(filtered)  # Output: [225, 256, 529, 1764]

if((text_lenght := len("Hello World!")) > 5):
  print(f"Text has {text_lenght} characters")

def callme():
  return 100

if((res:=callme())>50):
    print("Pass")
```

```
Enter something: quit
[225, 256, 529, 1764]
Text has 12 characters
Pass
```

```
# Membership Operators
# The Python membership operators test for the membership of an object in a sequence, such as strings, lists, or tuples.
# in            True if value is found in the sequence
# not in        True if value is not found in the sequence

# Membership operators are used to test whether a value exists within a sequence such as a list, tuple, or string.

# in operator and not in operator

list1 = [1, 2, 3, 4, 5, "kite"]
str1 = "Hello World"
set1 = {1, 2, 3, 4, 5}
dict1 = {1: "love", 2:"for", 3:"love"}
tup1 = (1, 2, 3, 4, 5)

# checking an integer in a list
print(2 in list1)

# checking an string in a list
print("kite" in list1)

# checking a character in a string
print('O' in str1)

# checking an integer in a set
print(6 in set1)

# checking for a key in a dictionary
print(1 in dict1)

# checking for an integer in a tuple
print(9 in tup1)

dict1 = {1: 'apple', 2: 'banana', 3: 'cherry'}

# Check if value 'banana' exists
print('banana' in dict1.values())  # Output: True

# Check if value 'grape' exists
print('grape' in dict1.values())   # Output: False


# Similarly we can go with the not in operator
# The ‘not in’ Python operator evaluates to true if it does not find the variable in the specified sequence and false otherwise
```

```
True
True
False
False
True
False
True
False
```

```
# Identity Operators
# The Python Identity Operators are used to compare the objects if both the objects are actually of the same data type and shar

# is operator and is not operator

# Python optimizes small integers by reusing existing objects.
num1 = 5
num2 = 5
print(num1 is num2)
num1 = num1+3
num3 = 8
print(num1 is num3)


# Lists in Python are mutable, so even if lst1 and lst2 have the same content, they are different objects in memory. Thus, lst1
# However, lst3 is explicitly assigned to lst1, meaning they point to the same object, so lst1 is lst3 returns True.
lst1 = [1, 2, 3]
lst2 = [1, 2, 3]
lst3 = lst1
print(lst1 is lst2)
print(lst1 is lst3)
print(lst2 is lst3)

# tuple
tup1 = (1,2,3)
tup2 = (1,2,3)
tup3 = tup1

print("tup1 is tup2",tup1 is tup2)
print("tup1 is tup3",tup1 is tup3)

# For strings, it depends on whether Python decides to intern the string or not.
str1 = "hello world"
str2 = "hello world"

print(str1 is str2)

# Similrly we can go with the is not operator
# is not - Returns True if objects are different

# Note
# Difference between ‘==’ and ‘is’ Operator
# While comparing objects in Python, the users often gets confused between the Equality operator and Identity ‘is’ operator.
# The equality operator is used to compare the value of two variables, whereas the identities operator is used to compare the m
print("lst1 == lst2",lst1 == lst2)
print("lst1 is lst2",lst1 is lst2)

s1 = "hello"
s2 = "hello"
print(s1 is s2)

s3 = "hello world"
s4 = "hello world"
print(s3 is s4)

import sys
s5 = sys.intern("hello world")
s6 = sys.intern("hello world")
print(s5 is s6)
```

```
True
True
False
True
False
tup1 is tup2 False
tup1 is tup3 True
False
```

```
lst1 == lst2 True
lst1 is lst2 False
True
False
True
```

```
# Ternary Operator
print("Greater than" if 5 > 6 else "less than")
# Nested Ternary Operator
print("Positive" if -15 > 0 else "Negative" if -15 < 0 else "Zero")
```

```
less than
Negative
```

```
# Arithematic operators
# Used for mathematical operations.
x = 5 + 3
print("add = ",x)
x = 5 - 3
print("sub = ",x)
x = 5 * 3
print("mul = ",x)
x = 5 / 3
print("div = ",x)
x = 5 % 3
print("mod = ",x)
x = 5 // 3
print("floor div = ",x)
x = 5 ** 3
print("expo = ",x)
```

```
add =  8
sub =  2
mul =  15
div =  1.6666666666666667
mod =  2
floor div =  1
expo =  125
```

```
# Comparison Operators (Relational Operator)
# These operators compare the values of two operands and return a boolean (True or False).

# equal
# In Python, you use == to check for equality and is to check for identity (if two references point to the same object).
x = 5 == 5
print(x)
y = 5 == 3
print(y)

# notEqual
x = 5 != 5
print(x)
y = 5 != 3
print(y)

# greater than
x = 5 > 5
print(x)
y = 5 > 3
print(y)
z = 5 > 8
print(z)

# less than
x = 5 < 5
print(x)
y = 5 < 3
print(y)
z = 5 < 8
print(z)

# greater than or equal
x = 5 >= 5
print(x)
y = 5 >= 3
print(y)
z = 5 >= 8
```

```
print(z)

# less than or equal
x = 5 >= 5
print(x)
y = 5 >= 3
print(y)
z = 5 >= 8
print(z)
```

```
True
False
False
True
False
True
False
False
False
True
True
True
False
True
True
False
```

```
# Logical Operators
# These operators are used to combine two or more conditional expressions and based on the logical rules, it writtens result as
x = (5 > 3) and (3 < 4) # and: Returns True if both statements are true.
print(x)
x = (5 > 3) or (3 > 4) # or: Returns True if at least one of the statements is true.
print(x)
x = not(5 > 3)  # not: Reverses the result, returns False if the result is true.
print(x)
```

```
True
True
False
```

```
# Bitwise Operators
# These operators are used to perform bit-level operations.

# 5 = 0101
# 3 = 0011
# 8 = 1000

# & (AND): Sets each bit to 1 if both bits are 1.
x = 5 & 3
print(x)
# | (OR): Sets each bit to 1 if one of the bits is 1.
x = 5 | 3
print(x)
# ^ (XOR): Sets each bit to 1 if only one of the bits is 1.
x = 5 ^ 3
print(x)
# ~ (NOT): Inverts all the bits.  (basically = -(n+1))
x = ~15 # -(1111)+0001 = -(10000) = -16
print(x)
# << (Zero fill left shift): Shifts bits to the left, filling with zeros.
x = 5 << 1
print(x)
x = -5 << 1
print(x)
# >> (Signed right shift): Shifts bits to the right.
# Convert 5 to binary:
# 5  = 0000 0101 (in 8-bit representation)
# Shift right by 1 bit:
# 0000 0101  >> 1  →  0000 0010  (which is 2 in decimal)
x = 5 >> 1
print(x)
# Negative numbers in binary are stored in two’s complement form.
# Convert -5 to binary (using 8-bit representation):
# -5  = 1111 1011  (two’s complement of 5)
# Shift right by 1 bit:
# 1111 1011  >> 1  →  1111 1101
# 1111 1101 is still a negative number in two’s complement, which is -3 in decimal.
```

```
x = -5 >> 1
print(x)
```

```
1
7
6
-16
10
-10
2
-3
```

```
# Assignment Operators
# Initial assignment
x = 10
print("Initial value of x:", x)

# += : Add and Assign
x += 5  # Equivalent to x = x + 5
print("After x += 5:", x)

# -= : Subtract and Assign
x -= 3  # Equivalent to x = x - 3
print("After x -= 3:", x)

# *= : Multiply and Assign
x *= 2  # Equivalent to x = x * 2
print("After x *= 2:", x)

# /= : Divide and Assign
x /= 4  # Equivalent to x = x / 4
print("After x /= 4:", x)

# %= : Modulus and Assign
x %= 3  # Equivalent to x = x % 3
print("After x %= 3:", x)

# //= : Floor Divide and Assign
x = 7   # Reassign x for demonstration
x //= 2  # Equivalent to x = x // 2
print("After x //= 2:", x)

# **= : Exponent and Assign
x **= 3  # Equivalent to x = x ** 3
print("After x **= 3:", x)

# &= : Bitwise AND and Assign
x = 10  # Reassign x for demonstration
x &= 7  # Equivalent to x = x & 7 (10 & 7 = 2 in binary)
print("After x &= 7:", x)

# |= : Bitwise OR and Assign
x |= 5  # Equivalent to x = x | 5 (2 | 5 = 7 in binary)
print("After x |= 5:", x)

# ^= : Bitwise XOR and Assign
x ^= 3  # Equivalent to x = x ^ 3 (7 ^ 3 = 4 in binary)
print("After x ^= 3:", x)

# <<= : Bitwise Left Shift and Assign
x <<= 2  # Equivalent to x = x << 2 (4 << 2 = 16 in binary)
print("After x <<= 2:", x)

# >>= : Bitwise Right Shift and Assign
x >>= 1  # Equivalent to x = x >> 1 (16 >> 1 = 8 in binary)
print("After x >>= 1:", x)

# := : Walrus operator (Assignment expression)
# This operator is used within an expression and is only available in Python 3.8+
if (y := x + 5) > 10:
    print("y is greater than 10; y:", y)
```

```
Initial value of x: 10
After x += 5: 15
After x -= 3: 12
After x *= 2: 24
After x /= 4: 6.0
```

```
After x %= 3: 0.0
After x //= 2: 3
After x **= 3: 27
After x &= 7: 2
After x |= 5: 7
After x ^= 3: 4
After x <<= 2: 16
After x >>= 1: 8
y is greater than 10; y: 13
```

```
# Parentheses: ()
# Exponentiation: **
# Unary Plus, Minus, and Bitwise NOT: +x, -x, ~x
# Multiplication, Division, Floor Division, and Modulus: *, /, //, %
# Addition and Subtraction: +, -
# Bitwise Shifts: <<, >>
# Bitwise AND: &
# Bitwise XOR: ^
# Bitwise OR: |
# Comparison Operators: ==, !=, >, >=, <, <=, is, is not, in, not in
# Logical NOT: not
# Logical AND: and
# Logical OR: or
# Conditional Expressions: if-else
# Assignment Operators: =, +=, -=, *=, /=, %=, //=, **=, &=, |=, ^=, >>=, <<=
# Lambda Expressions: lambda

# Associativity defines the order in which operators of the same precedence level are evaluated.
# Most operators in Python have left-to-right associativity. However, the exponentiation operator (**) has right-to-left associ

x = 2 ** 3 ** 2  # x = 2 ** (3 ** 2) = 2 ** 9 = 512
y = (2 ** 3) ** 2  # y = (2 ** 3) ** 2 = 8 ** 2 = 64
# In above code x = 2 ** 3 ** 2, the exponentiation is evaluated from right to left because of the right-to-left associativity


x = 2 + 3 * 4  # x = 2 + (3 * 4) = 2 + 12 = 14
# The multiplication (*) has higher precedence than addition (+), so 3 * 4 is evaluated first.

x = -3 ** 2  # x = -(3 ** 2) = -9
y = (-3) ** 2  # y = 9
# The exponentiation (**) has higher precedence than the unary minus (-), so 3 ** 2 is evaluated first, and then the result is


x = True or False and False  # x = True or (False and False) = True or False = True
# The and operator has higher precedence than or, so False and False is evaluated first.


x = 3 < 4 and 4 < 5  # x = (3 < 4) and (4 < 5) = True and True = True
# The comparison operators (<, >, etc.) have higher precedence than the logical operators (and, or, etc.).


x = (2 + 3) * 4  # x = 5 * 4 = 20
# Parentheses can be used to change the order of evaluation, forcing the addition to be evaluated before the multiplication.

print(3 + 5 * 2)   # 13 (Multiplication first)
print((3 + 5) * 2) # 16 (Parentheses first)

# chaining of operators is also there but don't do unless required
```

```
13
16
```

```
# | **Precedence** | **Operators** | **Description** | **Associativity** |
# |----------------|---------------|-----------------|-------------------|
# | **1 (Highest)** | `()` | Parentheses | **Left to Right** |
# | **2** | `x[index]`, `x[index:index]` | Subscription (indexing), slicing | **Left to Right** |
# | **3** | `await x` | Await expression (async operations) | **N/A** |
# | **4** | `**` | Exponentiation (Power) | **Right to Left** |
# | **5** | `+x`, `-x`, `~x` | Unary plus, Unary minus, Bitwise NOT | **Right to Left** |
# | **6** | `*`, `@`, `/`, `//`, `%` | Multiplication, matrix multiplication, division, floor division, modulus | **Left to Rig
# | **7** | `+`, `-` | Addition, Subtraction | **Left to Right** |
# | **8** | `<<`, `>>` | Bitwise Shift Left, Shift Right | **Left to Right** |
# | **9** | `&` | Bitwise AND | **Left to Right** |
# | **10** | `^` | Bitwise XOR | **Left to Right** |
# | **11** | `|` | Bitwise OR | **Left to Right** |
# | **12** | `in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==` | Comparison, Membership, Identity Tests | **Left
# | **13** | `not x` | Boolean NOT | **Right to Left** |
```

```
# | **14** | `and` | Boolean AND | **Left to Right** |
# | **15** | `or` | Boolean OR | **Left to Right** |
# | **16** | `if-else` | Conditional Expression (Ternary Operator) | **Right to Left** |
# | **17** | `lambda` | Lambda Expression | **N/A** |
# | **18 (Lowest)** | `:=` | Walrus Operator (Assignment Expression) | **Right to Left** |
```

```
# Step-by-step Explanation:
# 1. Understanding the >> operator:
# The >> operator is the right shift operator.

# In Python, when you use x >> n, it shifts the bits of x to the right by n positions.

# For positive numbers, it shifts in 0s from the left.

# For negative numbers, Python preserves the sign (arithmetic shift): it shifts in 1s from the left to maintain the sign.

# 2. Binary of -5 in 8-bit two’s complement:
# First, binary of +5 in 8-bit:
# 0000 0101

# Two’s complement of +5 (to get -5):

# Invert bits: 1111 1010

# Add 1: 1111 1011 → This is -5 in 8-bit two's complement.

# 3. Right shifting -5:
# 1111 1011  (which is -5)
# >> 1
# ---------
# 1111 1101  (arithmetic right shift – sign-extended with 1)
# 4. What is 1111 1101 in decimal?
# This is a negative number (leading 1).

# To find its decimal value:

# Invert: 0000 0010

# Add 1: 0000 0011 → which is 3

# So 1111 1101 = -3
```