In [ ]:

```
Q.1. Create two int type variables, apply addition, subtraction,
division and multiplicationsand store the results in variables.
Then print the data in the following format by calling the
variables:

First variable is __ & second variable is __.
Addition: __ + __ = __
Subtraction: __ - __ = __
Multiplication: __ * __ = __
Division: __ / __ = __
```

In [1]:

```
A ) # Create the variables
num1 = 10
num2 = 5

# Perform the arithmetic operations and store the results
addition = num1 + num2
subtraction = num1 - num2
multiplication = num1 * num2
division = num1 / num2

# Print the results in the desired format
print("First variable is", num1, "& second variable is", num2)
print("Addition: {} + {} = {}".format(num1, num2, addition))
print("Subtraction: {} - {} = {}".format(num1, num2, subtraction))
print("Multiplication: {} * {} = {}".format(num1, num2, multiplication))
print("Division: {} / {} = {}".format(num1, num2, division))
```

```
First variable is 10 & second variable is 5
Addition: 10 + 5 = 15
Subtraction: 10 - 5 = 5
Multiplication: 10 * 5 = 50
Division: 10 / 5 = 2.0
```

In [ ]:

```
Q.2. What is the difference between the following operators:
(i) ‘/’ & ‘//’
(ii) ‘**’ & ‘^’


A) (i) / and // operators:

The / operator is the division operator in Python. It performs
floating-point division, which means it returns a floating-point
result, even if the operands are integers.
For example, 7 / 2 would give the result 3.5.
The // operator is the floor division operator in Python.
It performs integer division, which means it returns the
largest integer less than or equal to the division result.
For example, 7 // 2 would give the result 3.
It discards the decimal part of the division result and
returns the floor value.

(ii) ** and ^ operators:

The ** operator is the exponentiation operator in Python.
It raises the left operand to the power of the right operand.
For example, 2 ** 3 would give the result 8, which is 2 raised
to the power of 3.
The ^ operator, known as the bitwise XOR operator, is used for
performing bitwise XOR operation between two integers.
It applies the XOR operation bit by bit on the binary
representation of the numbers. For example, 5 ^ 3 would give
the result 6, which is the bitwise XOR of the binary
representations of 5 (0101) and 3 (0011).

To summarize:

/ performs floating-point division.
// performs integer division (floor division).
** performs exponentiation.
^ performs bitwise XOR operation.
It's important to note that the usage and behavior of
these operators may vary in different programming languages.
The information provided here specifically pertains to their
usage in Python.
```

In [ ]:

```
Q.3. List the logical operators.


In Python, the logical operators are used to perform logical operations
on Boolean values (True or False). The logical operators in Python are:

and: The and operator returns True if both operands are True,
    otherwise it returns False. It performs a logical conjunction.
Example: True and False returns False.

or: The or operator returns True if at least one of the operands is True,
    otherwise it returns False. It performs a logical disjunction.
Example: True or False returns True.

not: The not operator is a unary operator that returns the
    opposite Boolean value of the operand. If the operand is True,
    not returns False, and if the operand is False, not returns True.
Example: not True returns False.

These logical operators can be used to combine and manipulate
Boolean values in conditional statements, logical expressions,
and boolean operations. They are fundamental for controlling the
flow and behavior of programs based on logical conditions.
```

In [ ]:

```
Q.4. Explain right shift operator and left shift operator with examples.

In Python, the right shift (>>) and left shift (<<) operators are
used to perform bitwise shift operations on integers.
These operators shift the bits of an integer to the right or left,
respectively. Here's an explanation of each operator with examples:

Right Shift (>>):
The right shift operator (>>) shifts the bits of an integer to
the right by a specified number of positions. The rightmost bits are
discarded, and the leftmost positions are filled with the sign bit
(for signed integers) or with zeros (for unsigned integers).

Syntax: x >> n

x: The integer to be shifted.
n: The number of positions to shift the bits.
Example:


x = 10  # Binary: 1010
n = 2

result = x >> n  # Right shift x by 2 positions
print(result)   # Output: 2
In this example, the binary representation of x is 1010.
When we right shift x by 2 positions (x >> 2), the result is 10,
which is equivalent to the decimal value of 2.

Left Shift (<<):
The left shift operator (<<) shifts the bits of an integer
to the left by a specified number of positions. Zeros are filled
in from the right, and the leftmost bits are discarded.
Syntax: x << n

x: The integer to be shifted.
n: The number of positions to shift the bits.
Example:


x = 5  # Binary: 101
n = 2

result = x << n  # Left shift x by 2 positions
print(result)    # Output: 20
In this example, the binary representation of x is 101.
When we left shift x by 2 positions (x << 2), the result is 10100,
which is equivalent to the decimal value of 20.

These bitwise shift operators are commonly used in scenarios
where bitwise manipulation of integers is required, such as in
low-level programming, bit-level calculations, and encoding schemes.
```

In [ ]:

```
Q.5. Create a list containing int type data of length 15.
Then write a code to check if 10 ispresent in the list or not.
```

In [2]:

```
# Create a list of integers
my_list = [2, 5, 8, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43]

# Check if 10 is present in the list
if 10 in my_list:
    print("10 is present in the list.")
else:
    print("10 is not present in the list.")
```

```
10 is present in the list.
```

In [ ]: