

PYTHON PROGRAMMING LANGUAGE

OPERATORS

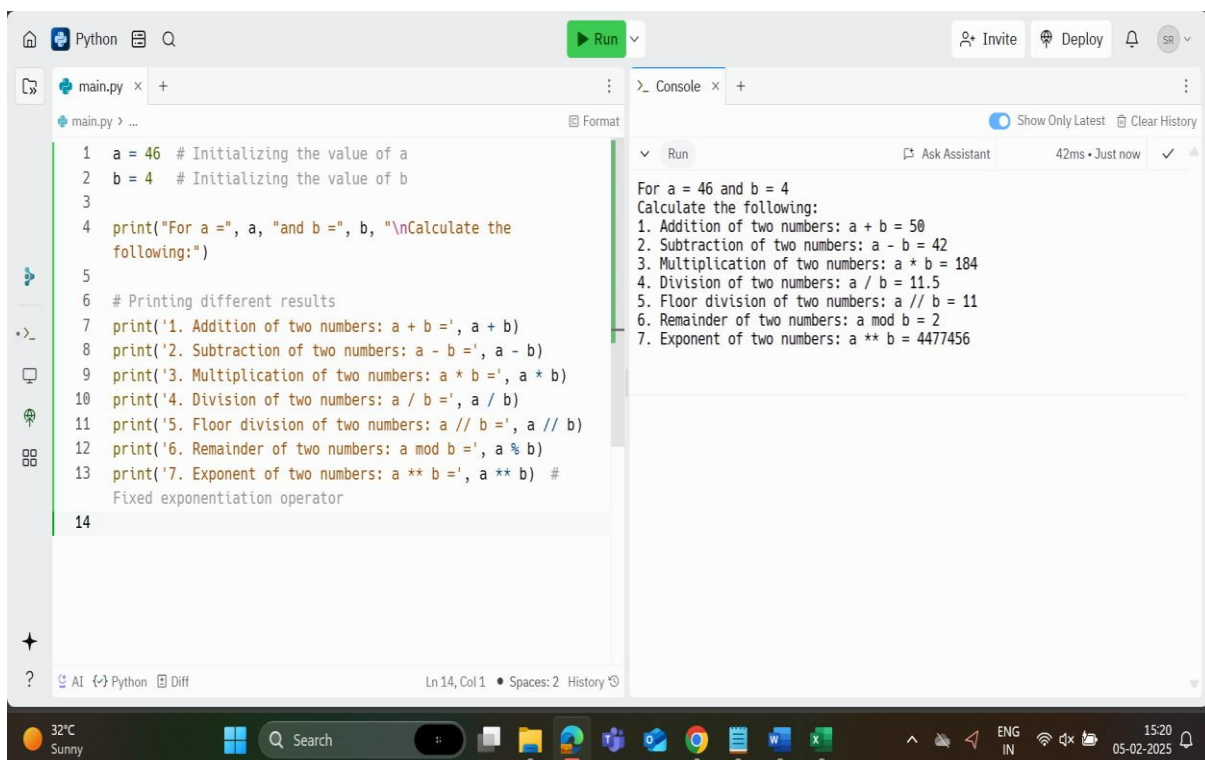
Operators are the symbols used to perform a specific operation on different values and variables. These values and variables are considered as the Operands, on which the operator is applied. Operators serve as the foundation upon which logic is constructed in a program in a particular programming language. In every programming language, some operators perform several tasks.

Arithmetic Operators

- Python Arithmetic Operators are used on two operands to perform basic mathematical operators like addition, subtraction, multiplication, and division.
- There are different types of arithmetic operators available in Python including the '+' operator for addition, '-' operator for subtraction, '*' for multiplication, '/' for division, '%' for modulus, '**' for exponent and '//' for floor division.

Example 1: Using all Arithmetic Operators:

Output:



```
1 a = 46 # Initializing the value of a
2 b = 4  # Initializing the value of b
3
4 print("For a =", a, "and b =", b, "\nCalculate the
   following:")
5
6 # Printing different results
7 print('1. Addition of two numbers: a + b =', a + b)
8 print('2. Subtraction of two numbers: a - b =', a - b)
9 print('3. Multiplication of two numbers: a * b =', a * b)
10 print('4. Division of two numbers: a / b =', a / b)
11 print('5. Floor division of two numbers: a // b =', a // b)
12 print('6. Remainder of two numbers: a mod b =', a % b)
13 print('7. Exponent of two numbers: a ** b =', a ** b) #
   Fixed exponentiation operator
14
```

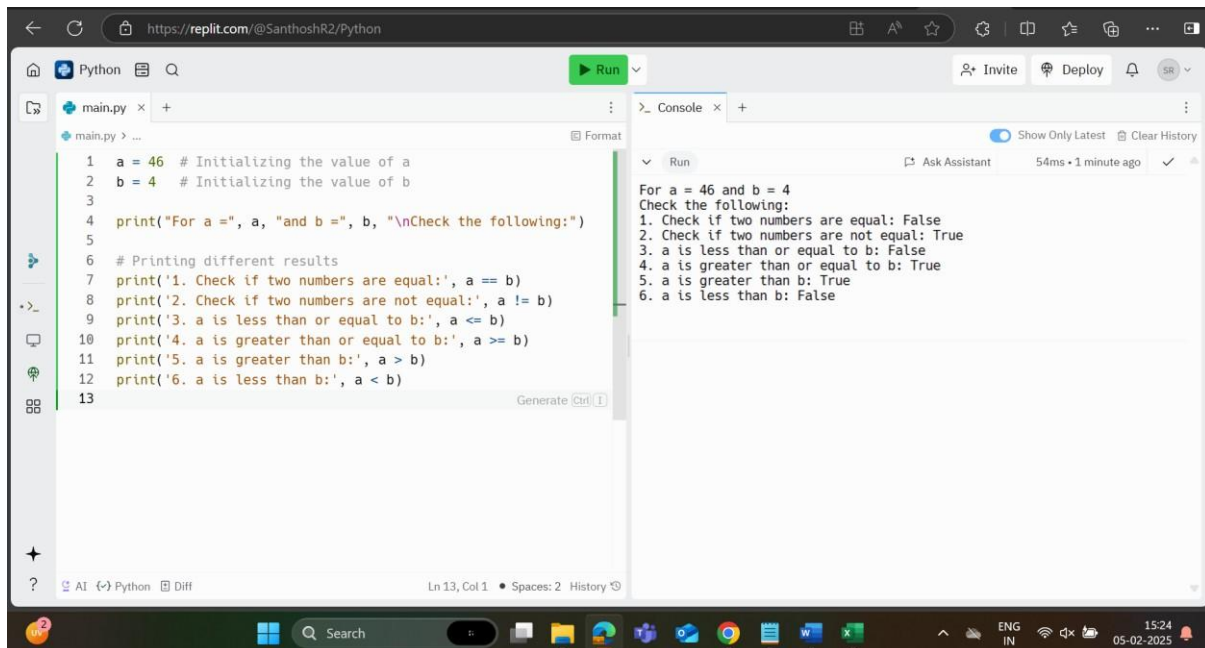
For a = 46 and b = 4
Calculate the following:
1. Addition of two numbers: a + b = 50
2. Subtraction of two numbers: a - b = 42
3. Multiplication of two numbers: a * b = 184
4. Division of two numbers: a / b = 11.5
5. Floor division of two numbers: a // b = 11
6. Remainder of two numbers: a mod b = 2
7. Exponent of two numbers: a ** b = 4477456

Comparison Operators

Python Comparison operators are mainly used for the purpose of comparing two values or variables (operands) and return a Boolean value as either True or False accordingly. There are various types of comparison operators available in Python including the '==', '!=', '<=', '>=', '<', and '>'.

Example 2: Using all Comparison Operators:

Output:



The screenshot shows a Replit Python environment with a file named `main.py` and a console output. The code in `main.py` initializes `a = 46` and `b = 4`, then prints the results of six comparison operations. The console output shows the following results:

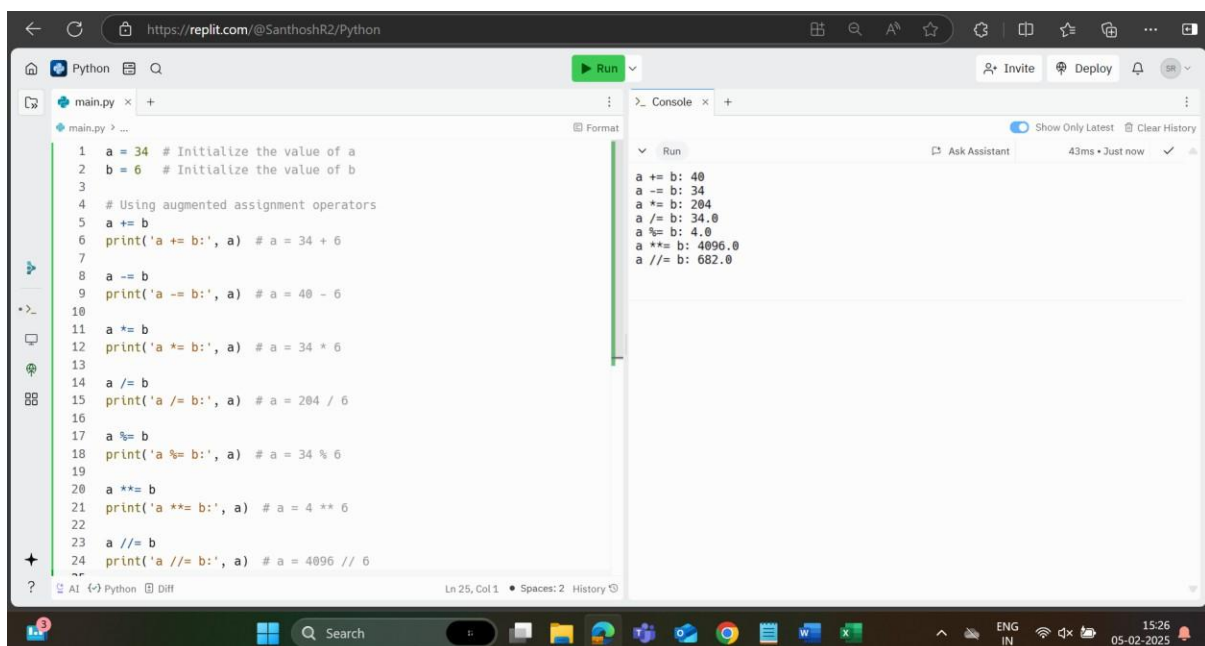
```
For a = 46, and b = 4, \nCheck the following:\n\n# Printing different results\n1. Check if two numbers are equal: False\n2. Check if two numbers are not equal: True\n3. a is less than or equal to b: False\n4. a is greater than or equal to b: True\n5. a is greater than b: True\n6. a is less than b: False
```

Assignment Operators

Using the assignment operators, the right expression's value is assigned to the left operand. Python offers different assignment operators to assign values to the variable. These assignment operators include '=', '+=', '-=', '*=', '/=', '%=', '//=', '**=', '&=', '|=', '^=', '>>=', and '<<='.

Example 3: Using all Assignment Operators:

Output:



The screenshot shows a Replit Python environment with a file named `main.py` and a console output. The code in `main.py` initializes `a = 34` and `b = 6`, then uses various assignment operators to update the value of `a`. The console output shows the following results:

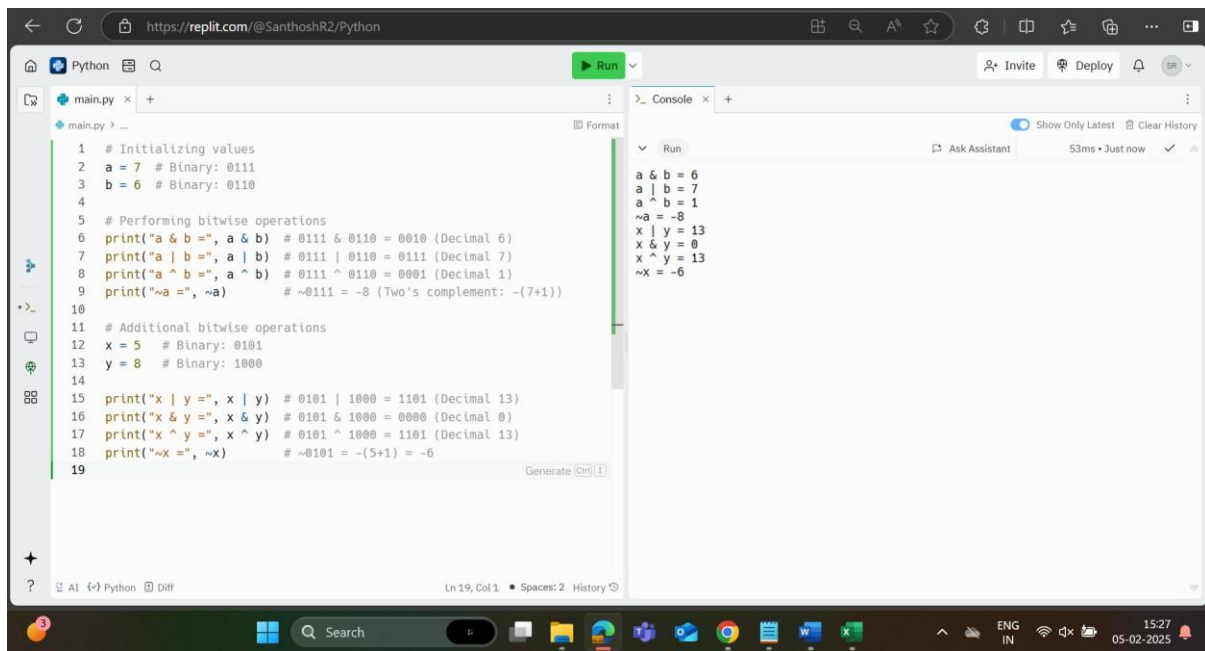
```
a += b: 40\na -= b: 34\na *= b: 204\na /= b: 34.0\na %= b: 4.0\na **= b: 4096.0\na // b: 682.0
```

Bitwise Operators

The two operands' values are processed bit by bit by the bitwise operators. There are various Bitwise operators used in Python, such as bitwise OR (`|`), bitwise AND (`&`), bitwise XOR (`^`), negation (`~`), Left shift (`<<`). Consider the case below.

Example 4: Using all Bitwise Operators:

Output:



The screenshot shows a Replit Python environment with a file named `main.py`. The code performs various bitwise operations on variables `a` and `b`. The output in the console shows the results of these operations, including binary and decimal representations.

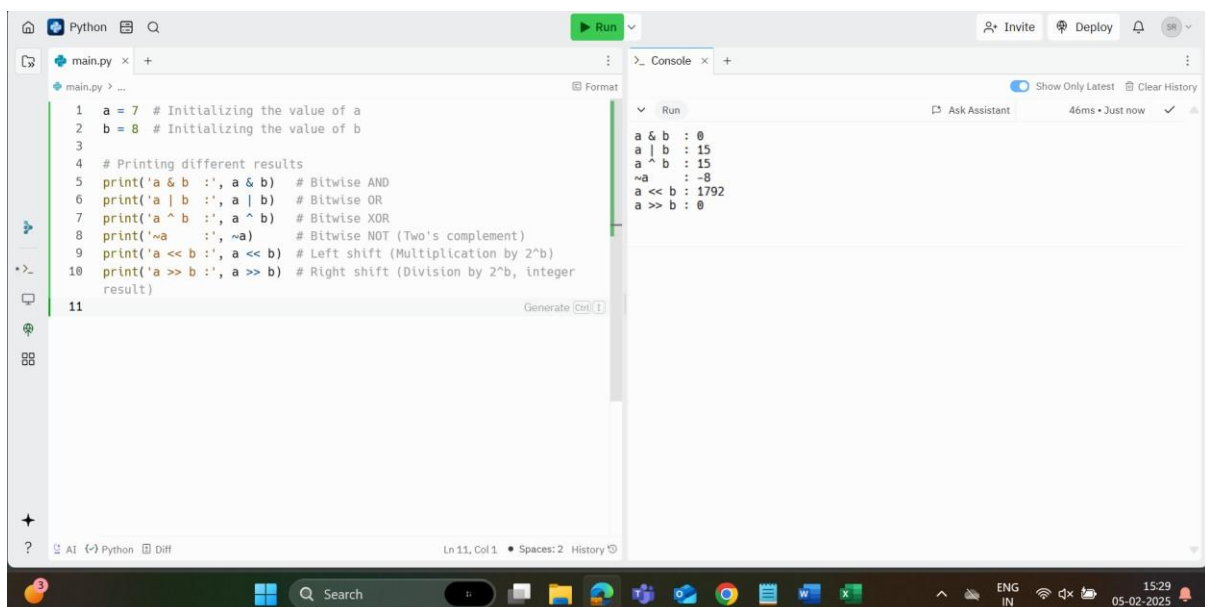
```
1 # Initializing values
2 a = 7 # Binary: 0111
3 b = 6 # Binary: 0110
4
5 # Performing bitwise operations
6 print("a & b =", a & b) # 0111 & 0110 = 0010 (Decimal 6)
7 print("a | b =", a | b) # 0111 | 0110 = 0111 (Decimal 7)
8 print("a ^ b =", a ^ b) # 0111 ^ 0110 = 0001 (Decimal 1)
9 print("~a =", ~a) # ~0111 = -8 (Two's complement: -(7+1))
10
11 # Additional bitwise operations
12 x = 5 # Binary: 0101
13 y = 8 # Binary: 1000
14
15 print("x | y =", x | y) # 0101 | 1000 = 1101 (Decimal 13)
16 print("x & y =", x & y) # 0101 & 1000 = 0000 (Decimal 0)
17 print("x ^ y =", x ^ y) # 0101 ^ 1000 = 1101 (Decimal 13)
18 print("~x =", ~x) # ~0101 = -5 (Two's complement: -(5+1))
19
```

Console Output:

```
a & b = 6
a | b = 7
a ^ b = 1
~a = -8
x | y = 13
x & y = 0
x ^ y = 13
~x = -6
```

Example 5: Using all Bitwise Operators:

Output:



The screenshot shows a Replit Python environment with a file named `main.py`. The code initializes variables `a` and `b` and performs various bitwise operations. The output in the console shows the results of these operations.

```
1 a = 7 # Initializing the value of a
2 b = 8 # Initializing the value of b
3
4 # Printing different results
5 print('a & b :', a & b) # Bitwise AND
6 print('a | b :', a | b) # Bitwise OR
7 print('a ^ b :', a ^ b) # Bitwise XOR
8 print('~a :', ~a) # Bitwise NOT (Two's complement)
9 print('a << b :', a << b) # Left shift (Multiplication by 2^b)
10 print('a >> b :', a >> b) # Right shift (Division by 2^b, integer result)
11
```

Console Output:

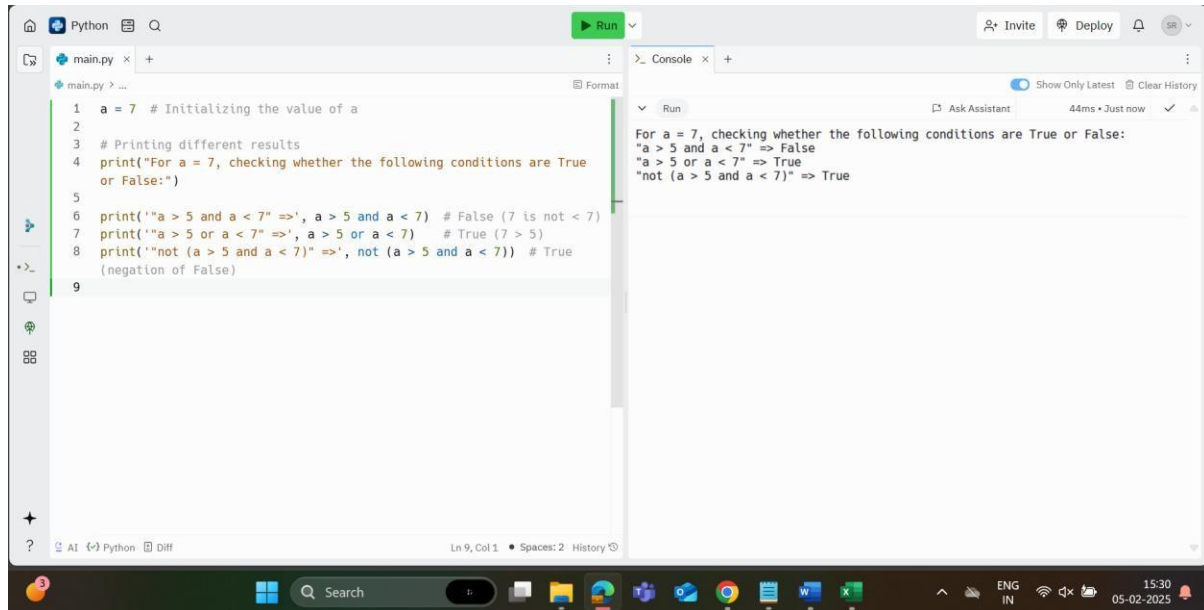
```
a & b : 0
a | b : 15
a ^ b : 15
~a : -8
a << b : 1792
a >> b : 0
```

Logical Operators

The assessment of expressions to make decisions typically uses logical operators. Python offers different types of logical operators such as and, or, and not. In the case of the logical AND, if the first one is 0, it does not depend upon the second one. In the case of the logical OR, if the first one is 1, it does not depend on the second one.

Example 6: Using all Logical Operators:

Output:



```
1 a = 7 # Initializing the value of a
2
3 # Printing different results
4 print("For a = 7, checking whether the following conditions are True or False:")
5
6 print('a > 5 and a < 7' =>, a > 5 and a < 7) # False (7 is not < 7)
7 print('a > 5 or a < 7' =>, a > 5 or a < 7) # True (7 > 5)
8 print('not (a > 5 and a < 7)' =>, not (a > 5 and a < 7)) # True
  (negation of False)
9
```

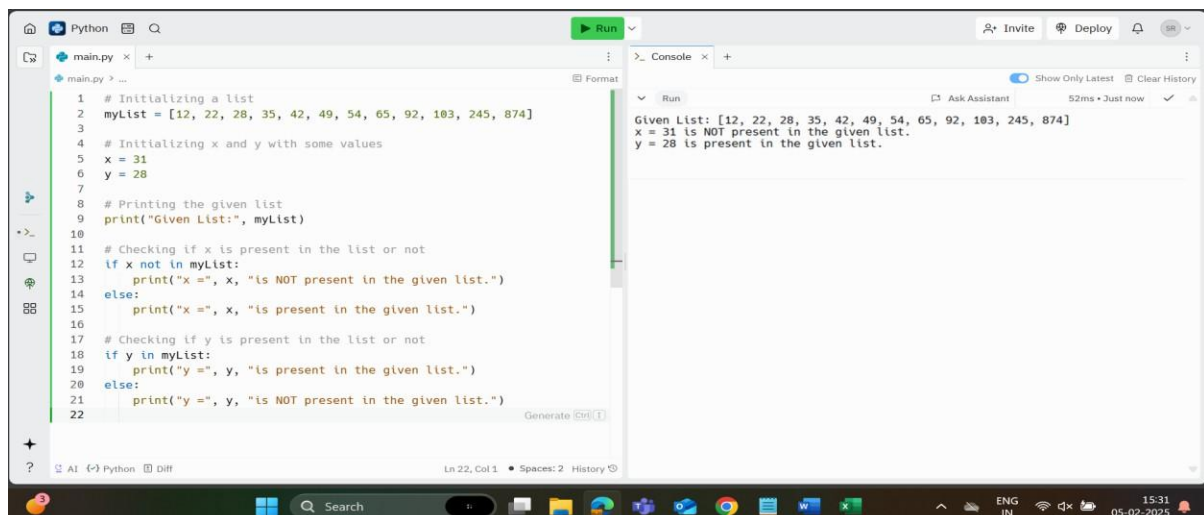
For a = 7, checking whether the following conditions are True or False:
"a > 5 and a < 7" => False
"a > 5 or a < 7" => True
"not (a > 5 and a < 7)" => True

Membership Operators

We can verify the membership of a value inside a Python data structure using the Python membership operators. The result is said to be true if the value or variable is in the sequence (list, tuple, or dictionary); otherwise, it returns false.

Example 7: Using all Membership Operators:

Output:



```
1 # Initializing a list
2 myList = [12, 22, 28, 35, 42, 49, 54, 65, 92, 103, 245, 874]
3
4 # Initializing x and y with some values
5 x = 31
6 y = 28
7
8 # Printing the given list
9 print("Given List:", myList)
10
11 # Checking if x is present in the list or not
12 if x not in myList:
13     print("x =", x, "is NOT present in the given list.")
14 else:
15     print("x =", x, "is present in the given list.")
16
17 # Checking if y is present in the list or not
18 if y in myList:
19     print("y =", y, "is present in the given list.")
20 else:
21     print("y =", y, "is NOT present in the given list.")
22
```

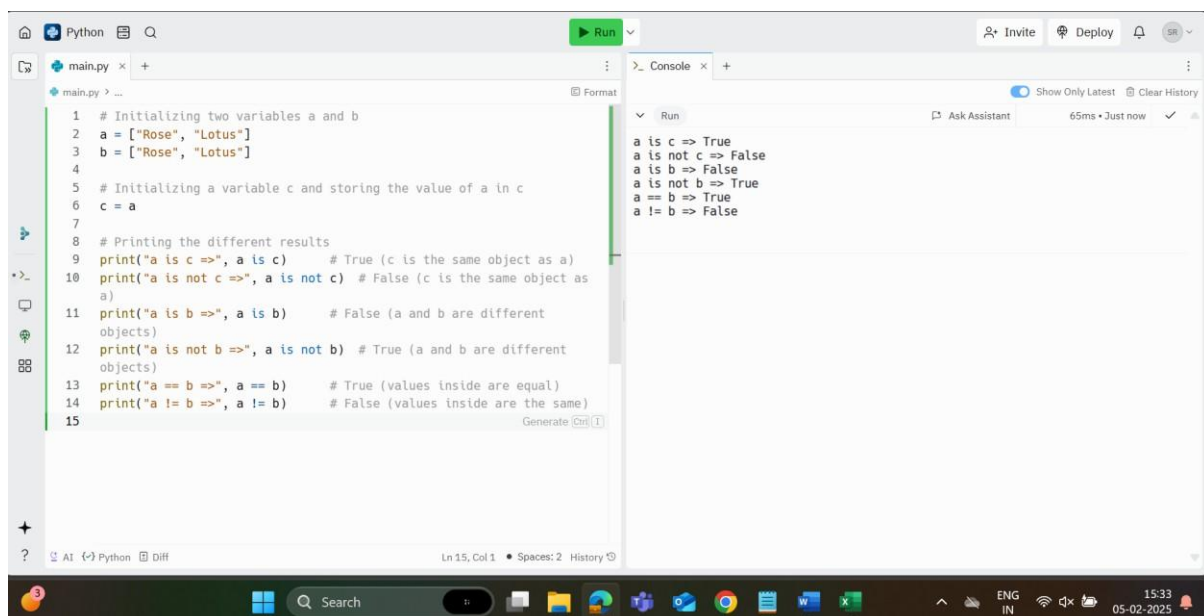
Given List: [12, 22, 28, 35, 42, 49, 54, 65, 92, 103, 245, 874]
x = 31 is NOT present in the given list.
y = 28 is present in the given list.

Identity Operators

Python offers two identity operators as is and is not, that are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

Example 8: Using all Identity Operators:

Output:



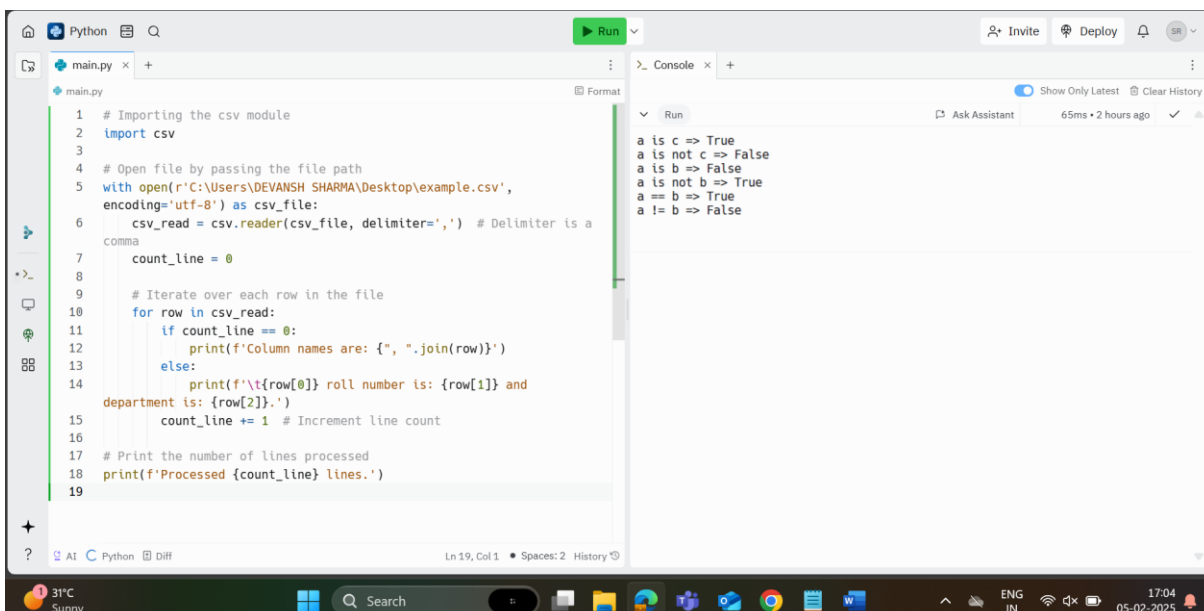
```
1 # Initializing two variables a and b
2 a = ["Rose", "Lotus"]
3 b = ["Rose", "Lotus"]
4
5 # Initializing a variable c and storing the value of a in c
6 c = a
7
8 # Printing the different results
9 print("a is c =>", a is c)      # True (c is the same object as a)
10 print("a is not c =>", a is not c) # False (c is the same object as a)
11 print("a is b =>", a is b)      # False (a and b are different objects)
12 print("a is not b =>", a is not b) # True (a and b are different objects)
13 print("a == b =>", a == b)      # True (values inside are equal)
14 print("a != b =>", a != b)      # False (values inside are the same)
15
```

Run

```
a is c => True
a is not c => False
a is b => False
a is not b => True
a == b => True
a != b => False
```

How to read CSV file in Python

The CSV file stands for a comma-separated values file. It is a type of plain text file where the information is organized in the tabular form. It can contain only the actual text data. The textual data don't need to be separated by the commas (.). There are also many separator characters such as tab (\t), colon(:), and semi-colon(;), which can be used as a separator. Let's understand the following example.



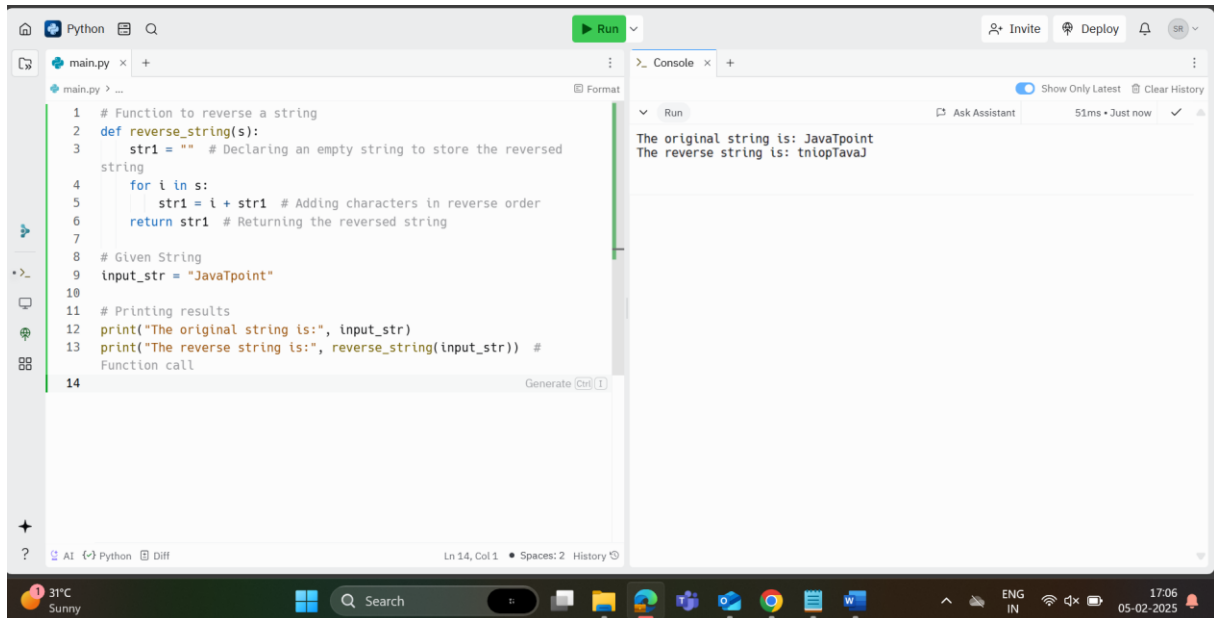
```
1 # Importing the csv module
2 import csv
3
4 # Open file by passing the file path
5 with open(r'C:\Users\DEVANSH SHARMA\Desktop\example.csv',
6           encoding='utf-8') as csv_file:
7     csv_read = csv.reader(csv_file, delimiter=',') # Delimiter is a comma
8     count_line = 0
9
10    # Iterate over each row in the file
11    for row in csv_read:
12        if count_line == 0:
13            print(f'Column names are: {", ".join(row)}')
14        else:
15            print(f'\t{row[0]} roll number is: {row[1]} and
16            department is: {row[2]}')
17            count_line += 1 # Increment line count
18
19    # Print the number of lines processed
20    print(f'Processed {count_line} lines.')
21
```

Run

```
a is c => True
a is not c => False
a is b => False
a is not b => True
a == b => True
a != b => False
```

How to reverse a string in Python

The collection of Unicode characters is Python String. Python has various capabilities for string control, yet Python string library doesn't uphold the in-constructed "switch()" capability. However, there are numerous methods for reversing the string. The following reverse Python String method is being defined.



The screenshot shows a Python IDE with a file named `main.py`. The code defines a function `reverse_string(s)` that reverses a string using a `for` loop. The function iterates over each character in the string and appends it to a new string `str1` in reverse order. The main program calls this function with the input string "JavaTpoint" and prints both the original and reversed strings.

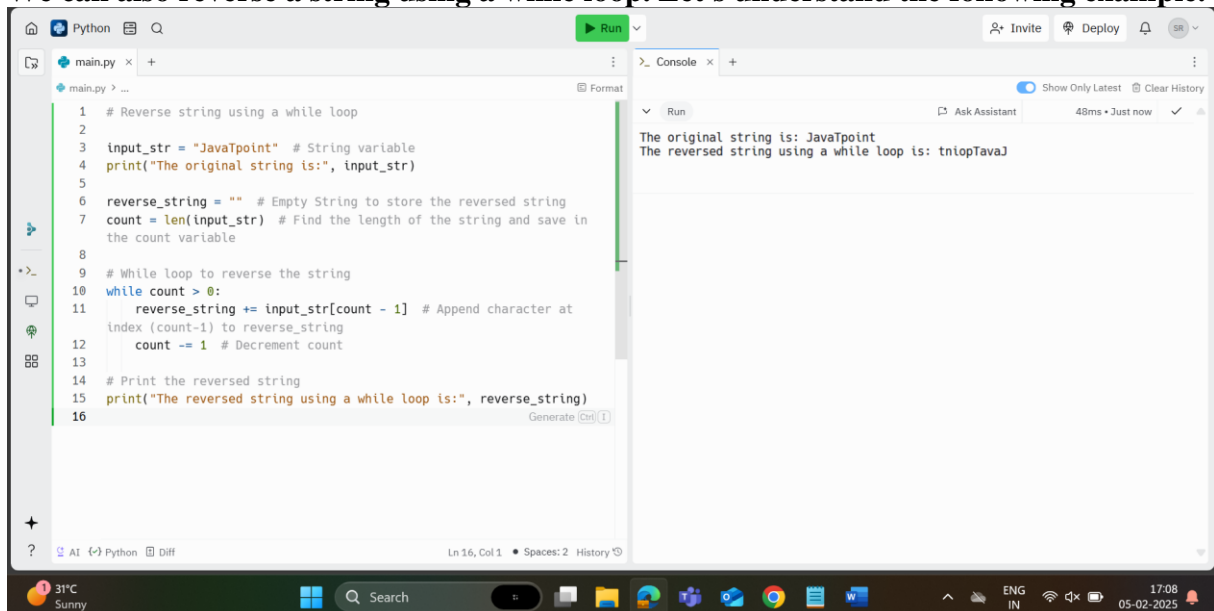
```
1 # Function to reverse a string
2 def reverse_string(s):
3     str1 = "" # Declaring an empty string to store the reversed
    string
4     for i in s:
5         str1 = i + str1 # Adding characters in reverse order
6     return str1 # Returning the reversed string
7
8 # Given String
9 input_str = "JavaTpoint"
10
11 # Printing results
12 print("The original string is:", input_str)
13 print("The reverse string is:", reverse_string(input_str)) #
    Function call
14
```

The console output shows:

```
The original string is: JavaTpoint
The reverse string is: tniopTavaJ
```

Using while loop

We can also reverse a string using a while loop. Let's understand the following example.



The screenshot shows a Python IDE with a file named `main.py`. The code defines a function `reverse_string` that reverses a string using a `while` loop. The function calculates the length of the input string and iterates from the end to the beginning, appending each character to a new string `reverse_string`. The main program calls this function with the input string "JavaTpoint" and prints both the original and reversed strings.

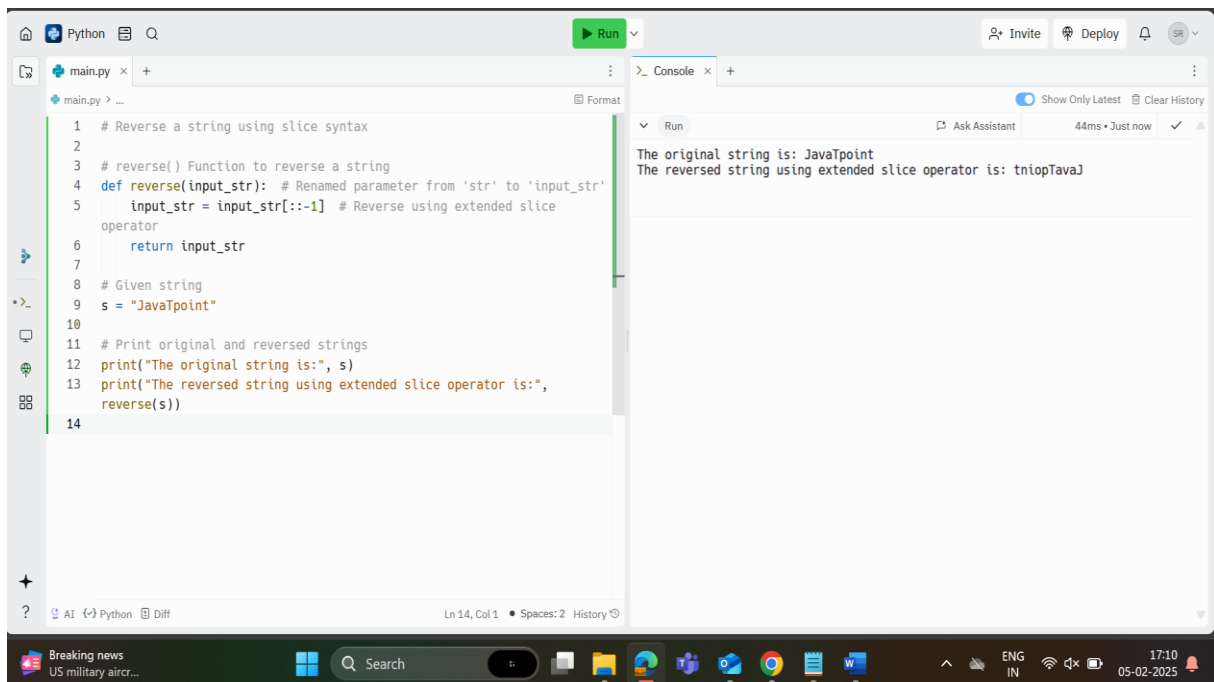
```
1 # Reverse string using a while loop
2
3 input_str = "JavaTpoint" # String variable
4 print("The original string is:", input_str)
5
6 reverse_string = "" # Empty String to store the reversed string
7 count = len(input_str) # Find the length of the string and save in
    the count variable
8
9 # While loop to reverse the string
10 while count > 0:
11     reverse_string += input_str[count - 1] # Append character at
    index (count-1) to reverse_string
12     count -= 1 # Decrement count
13
14 # Print the reversed string
15 print("The reversed string using a while loop is:", reverse_string)
16
```

The console output shows:

```
The original string is: JavaTpoint
The reversed string using a while loop is: tniopTavaJ
```


Using the slice ([]) operator

We can also reverse the given string using the extended slice operator. Let's understand the following example.



The screenshot shows a Python IDE with a file named `main.py`. The code in the editor is as follows:

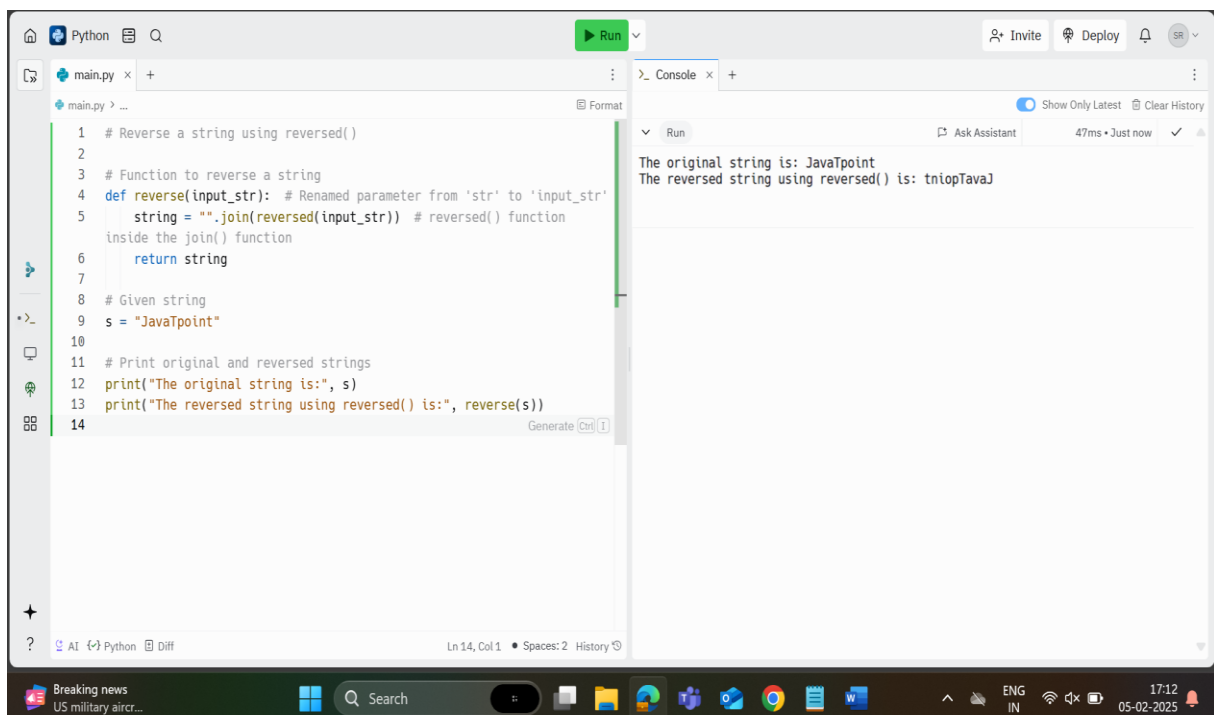
```
1 # Reverse a string using slice syntax
2
3 # reverse() Function to reverse a string
4 def reverse(input_str): # Renamed parameter from 'str' to 'input_str'
5     input_str = input_str[::-1] # Reverse using extended slice
6     return input_str
7
8 # Given string
9 s = "JavaTpoint"
10
11 # Print original and reversed strings
12 print("The original string is:", s)
13 print("The reversed string using extended slice operator is:",
14       reverse(s))
```

The console output on the right shows:

```
The original string is: JavaTpoint
The reversed string using extended slice operator is: tniopTavaJ
```

Using reverse function with join

Python provides the `reversed()` function to reverse the string. Let's understand the following example.



The screenshot shows a Python IDE with a file named `main.py`. The code in the editor is as follows:

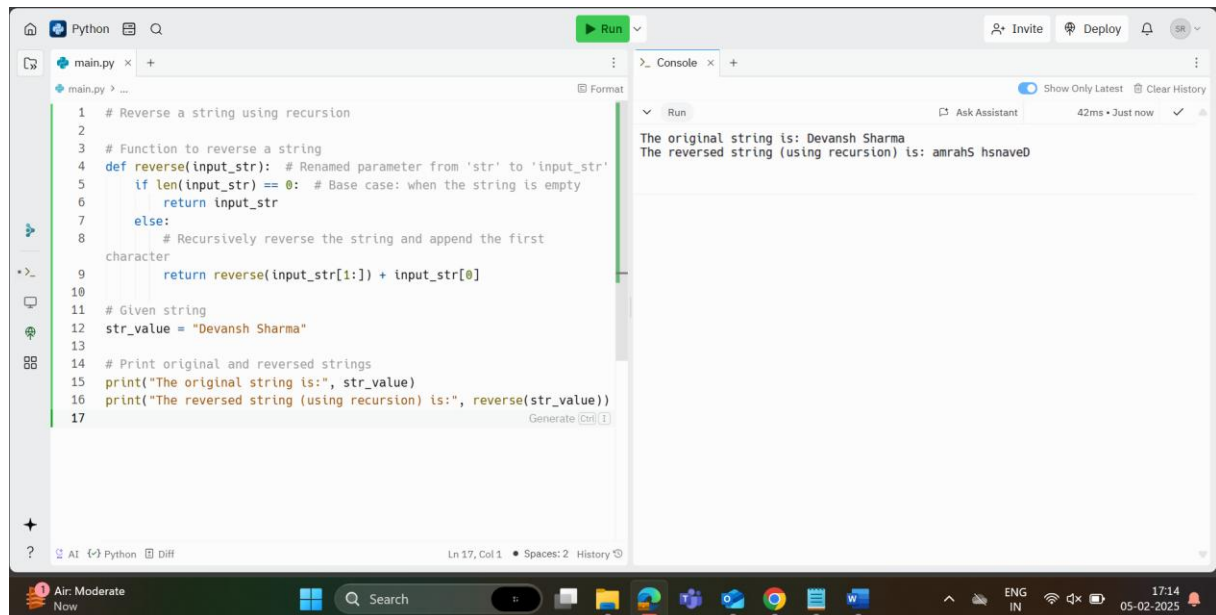
```
1 # Reverse a string using reversed()
2
3 # Function to reverse a string
4 def reverse(input_str): # Renamed parameter from 'str' to 'input_str'
5     string = "".join(reversed(input_str)) # reversed() function
6     return string
7
8 # Given string
9 s = "JavaTpoint"
10
11 # Print original and reversed strings
12 print("The original string is:", s)
13 print("The reversed string using reversed() is:", reverse(s))
14
```

The console output on the right shows:

```
The original string is: JavaTpoint
The reversed string using reversed() is: tniopTavaJ
```

Using recursion()

The recursion can also be used to turn the string around. Recursion is a cycle where capability calls itself. Look at the following example.



The screenshot shows a Python IDE with a file named `main.py`. The code defines a recursive function `reverse` that takes a string `input_str` and returns its reverse. The function uses a base case where if the length of the string is 0, it returns the string. Otherwise, it recursively calls itself with the string excluding the first character and then appends the first character to the result. The main program sets `str_value` to "Devansh Sharma", prints the original string, and prints the reversed string using the `reverse` function.

```
1 # Reverse a string using recursion
2 # Function to reverse a string
3 def reverse(input_str): # Renamed parameter from 'str' to 'input_str'
4     if len(input_str) == 0: # Base case: when the string is empty
5         return input_str
6     else:
7         # Recursively reverse the string and append the first
8         # character
9         return reverse(input_str[1:]) + input_str[0]
10
11 # Given string
12 str_value = "Devansh Sharma"
13
14 # Print original and reversed strings
15 print("The original string is:", str_value)
16 print("The reversed string (using recursion) is:", reverse(str_value))
17
```

The console output shows:

```
The original string is: Devansh Sharma
The reversed string (using recursion) is: amrahS hsnaveD
```

The status bar at the bottom indicates the cursor is at line 17, column 1, with 2 spaces. The system tray shows the date and time as 05-02-2025, 17:14.