Bit Wise Operators

In Python, bitwise operators manipulate individual bits of integers. These operators work at the binary level, performing operations like AND, OR, XOR, and bit shifting. Here are the main bitwise operators along with examples:

### 1. AND Operator (`&`):
- Performs a bitwise AND operation. The result has a 1 bit only if both corresponding bits of the operands are 1.

   ```python
   a = 5  # 101 in binary
   b = 3  # 011 in binary

   result = a & b
   print(result)  # Output: 1 (001 in binary)
   ```

### 2. OR Operator (`|`):
- Performs a bitwise OR operation. The result has a 1 bit if at least one of the corresponding bits of the operands is 1.

   ```python
   a = 5  # 101 in binary
   b = 3  # 011 in binary

   result = a | b
   print(result)  # Output: 7 (111 in binary)
   ```

### 3. XOR Operator (`^`):
- Performs a bitwise XOR (exclusive OR) operation. The result has a 1 bit if the corresponding bits of the operands are different.

   ```python
   a = 5  # 101 in binary
   b = 3  # 011 in binary

   result = a ^ b
   print(result)  # Output: 6 (110 in binary)
   ```

### 4. NOT Operator (`~`):
- Inverts the bits of its operand. It turns 0 bits into 1 and vice versa. The result is dependent on the number of bits used to represent the integer.

   ```python
   a = 5  # 101 in binary
   ```

    result = ~a
    print(result) # Output: -6 (varies depending on the number of bits used)

### 5. Left Shift Operator (`<<`):
- Shifts the bits of the operand to the left by a specified number of positions. The vacant positions on the right are filled with zeros.

    ```python
    a = 5  # 101 in binary

    result = a << 1
    print(result)  # Output: 10 (1010 in binary)
    ```

### 6. Right Shift Operator (`>>`):
- Shifts the bits of the operand to the right by a specified number of positions. The vacant positions on the left are filled based on the sign bit (0 for positive, 1 for negative).

    ```python
    a = 5  # 101 in binary

    result = a >> 1
    print(result)  # Output: 2 (10 in binary)
    ```

### Examples of Combining Bitwise Operators:

Bitwise operators can be combined to perform complex bit manipulations:

```python
# Check if the 3rd bit is set in a binary number
number = 12  # 1100 in binary

if number & (1 << 2):
    print("The 3rd bit is set.")
else:
    print("The 3rd bit is not set.")
```

In this example, the AND operator is used to check if the 3rd bit is set in the binary representation of the `number`.

Bitwise operators are used in low-level programming, optimization, and situations where direct manipulation of bits is required. They are less commonly used in everyday Python programming compared to logical and arithmetic operators.

Here are 15 examples :

### 1. **Bitwise AND (`&`):**
   ```python
   a = 5  # 0b0101 in binary
   b = 3  # 0b0011 in binary

   result = a & b
   print(bin(result))  # Output: 0b0001
   ```
   Explanation: The bitwise AND operator compares each bit of the two numbers. If both bits are 1, the result will have a corresponding bit set to 1.

### 2. **Bitwise OR (`|`):**
   ```python
   a = 5  # 0b0101 in binary
   b = 3  # 0b0011 in binary

   result = a | b
   print(bin(result))  # Output: 0b0111
   ```
   Explanation: The bitwise OR operator compares each bit of the two numbers. If at least one of the bits is 1, the result will have a corresponding bit set to 1.

### 3. **Bitwise XOR (`^`):**
   ```python
   a = 5  # 0b0101 in binary
   b = 3  # 0b0011 in binary

   result = a ^ b
   print(bin(result))  # Output: 0b0110
   ```
   Explanation: The bitwise XOR operator compares each bit of the two numbers. If the bits are different, the result will have a corresponding bit set to 1.

### 4. **Bitwise NOT (`~`):**
   ```python
   a = 5  # 0b0101 in binary

   result = ~a
   print(bin(result))  # Output: -0b0110 (in two's complement form)
   ```
   Explanation: The bitwise NOT operator inverts each bit of the number. Note that the result is represented in two's complement form.

### 5. **Left Shift (`<<`):**
   ```python
   a = 5  # 0b0101 in binary

   result = a << 2
   ```

   print(bin(result)) # Output: 0b10100
   ~~~
   Explanation: The left shift operator shifts the bits of the number to the left by the specified number of positions, filling the vacant positions with zeros.

### 6. **Right Shift (`>>`):**
   ```python
   a = 16  # 0b10000 in binary

   result = a >> 2
   print(bin(result))  # Output: 0b0010
   ~~~
   Explanation: The right shift operator shifts the bits of the number to the right by the specified number of positions. The vacant positions are filled based on the sign bit (for signed integers).

### 7. **Checking if a Bit is Set:**
   ```python
   num = 12  # 0b1100 in binary
   position = 2

   is_set = (num & (1 << position)) != 0
   print(is_set)  # Output: True
   ```
   Explanation: Using bitwise AND with a shifted bit, you can check if a specific bit is set in a number.

### 8. **Setting a Bit:**
   ```python
   num = 5  # 0b0101 in binary
   position = 1

   num |= (1 << position)
   print(bin(num))  # Output: 0b0111
   ```
   Explanation: Using bitwise OR with a shifted bit, you can set a specific bit in a number.

### 9. **Toggling a Bit:**
   ```python
   num = 6  # 0b0110 in binary
   position = 2

   num ^= (1 << position)
   print(bin(num))  # Output: 0b0010
   ```
   Explanation: Using bitwise XOR with a shifted bit, you can toggle a specific bit in a number.

### 10. **Counting Set Bits (Hamming Weight):**
   ```python
   num = 27  # 0b11011 in binary
   ```

   set_bits_count = bin(num).count('1')
   print(set_bits_count) # Output: 4
   ~~~
   Explanation: Counting the number of set bits (1s) in a binary representation, also known as Hamming Weight. In this example, there are four set bits in the binary representation of 27.

### 11. **Multiply by Powers of 2 (Left Shift):**
   ```python
   num = 5

   result = num << 3  # Equivalent to num * 2^3
   print(result)  # Output: 40
   ~~~
   Explanation: Left shifting a number by `n` is equivalent to multiplying the number by `2^n`.

### 12. **Divide by Powers of 2 (Right Shift):**
   ```python
   num = 32

   result = num >> 2  # Equivalent to num // 2^2
   print(result)  # Output: 8
   ```
   Explanation: Right shifting a number by `n` is equivalent to integer dividing the number by `2^n`.

### 13. **Detecting Overflow with Left Shift:**
   ```python
   max_int = 2**31 - 1  # Maximum signed 32-bit integer

   overflowed_value = max_int << 1
   print(overflowed_value)  # Output: -2 (overflowed in two's complement)
   ```
   Explanation: Left shifting a maximum signed 32-bit integer causes overflow, resulting in a negative value.

### 14. **Filling with Zeros (Left Shift):**
   ```python
   value = 7

   result = value << 4  # Left shift by 4 bits
   print(bin(result))  # Output: 0b111000 (filled with zeros on the right)
   ```
   Explanation: Left shifting fills the vacant positions with zeros.

### 15. **Arithmetic Right Shift for Negative Numbers:**
   ```python
   negative_num = -16

   result = negative_num >> 2
   ```

```
print(result)  # Output: -4
```

Explanation: Arithmetic right shift preserves the sign bit, so the result is still a negative number.