# 2
# Operators

Operators are fundamental building blocks in Python that enable you to perform operations on data and variables. They act on one or more operands to produce a result, making it possible to carry out tasks like arithmetic calculations, logical comparisons, and data manipulations. For example, the + operator can add two numbers or concatenate two strings, while the == operator checks if two values are equal.

Python's operators are diverse, covering arithmetic, comparison, logical, bitwise, assignment, membership, and identity operations. This wide range of functionality allows Python to handle complex expressions concisely and intuitively. The dynamic nature of Python's type system also means that operators behave differently based on the data types of their operands, making them versatile and adaptable. For instance, the / operator divides two numbers, but if the operands are integers, the // operator provides a floor division result.

Understanding operators is crucial for writing clear and efficient Python code. They form the foundation for constructing expressions, implementing logic, and manipulating data. Whether you are calculating values, comparing variables, or building complex expressions, operators provide the tools to interact with your data effectively.

In this chapter, we will explore:

- Arithmetic Operators
- Assignment Operators
- Comparison (Relational) Operators
- Logical Operators
- Identity Operators
- Membership Operators
- Conditional Operator (Ternary Operator)
- Bitwise Operators
- Operator Precedence

## Arithmetic Operators

Arithmetic operators in Python are used to perform mathematical operations on numeric values such as integers (int), floating-point numbers (float), and complex numbers. These operators work on two or more operands and return a result.

### Addition (+)

The addition operator adds two numbers.

Example:

```
a = 10
b = 5
result = a + b  # 15
```

It can also be used to concatenate strings:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2  # "Hello World"
```

## Subtraction (-)

The subtraction operator subtracts the second operand from the first.

Example:

```
a = 10
b = 5
result = a - b  # 5
```

## Multiplication (*)

The multiplication operator multiplies two numbers.

Example:

```
a = 10
b = 5
result = a * b  # 50
```

It can also be used with strings to repeat them:

```
text = "Hi "
result = text * 3  # "Hi Hi Hi "
```

## Division (/)

The division operator divides the first operand by the second and always returns a float.

Example:

```
a = 10
b = 4
result = a / b  # 2.5
```

### Floor Division (//)

The floor division operator divides the first operand by the second and truncates the result to the nearest whole number (integer for integers, floor for floats).

Example:

```
a = 10
b = 4
result = a // b  # 2
```

### Modulus (%)

The modulus operator returns the remainder of the division between two numbers.

Example:

```
a = 10
b = 3
result = a % b  # 1
```

### Exponentiation (**)

The exponentiation operator raises the first operand to the power of the second.

Example:

```
a = 2
b = 3
result = a ** b  # 8
```

It also works with fractional exponents for roots:

```
result = 16 ** 0.5  # 4.0 (square root of 16)
```

## Return Type Determination

Python determines the return type of arithmetic operations based on the types of the operands and the operation being performed. If both operands are of the same type, the result generally retains that type. For example, adding two integers results in an integer:

```
result = 5 + 3  # Result is int: 8
```

Adding two floats produces a float:

```
result = 5.5 + 3.0  # Result is float: 8.5
```

When the operands are of mixed types, Python promotes the result to the more general type to preserve precision. For instance, combining an integer and a float produces a float:

```
result = 5 + 3.0  # Result is float: 8.0
```

Division (/) always returns a float, even if both operands are integers:

```
result = 10 / 2  # Result is float: 5.0
```

Floor division (//) truncates the result, and its type depends on the operand types. If both are integers, the result is an integer. If one operand is a float, the result is a float:

```
result = 10 // 3  # int // int, result is int: 3

result = 10.0 // 3  # float // int, result is float: 3.0
```

The modulus operator (%) retains the type of the operands. For integers, the result is an integer, while for floats, the result is a float:

```
result = 10 % 3  # int % int, result is int: 1

result = 10.5 % 3  # float % int, result is float: 1.5
```

Exponentiation (**) follows similar rules. When both operands are integers, the result is an integer for positive exponents, but if the exponent is negative or one operand is a float, the result is a float:

```
result = 2 ** 3  # int ** int, result is int: 8

result = 2 ** -2  # int ** negative int, result is float: 0.25

result = 2.0 ** 3  # float ** int, result is float: 8.0
```

Python's dynamic typing and type promotion provide a balance of simplicity and accuracy for various arithmetic operations.

# Assignment Operators

Assignment operators in Python allow you to assign values to variables. These operators can perform basic assignment or combine assignment with other operations in a concise way.

## Simple Assignment (=)

The = operator assigns the value on the right-hand side to the variable on the left-hand side.

Example:

```
x = 10  # Assigns 10 to x

name = "Alice"  # Assigns "Alice" to name
```

## Compound Assignment Operators

Compound assignment operators combine a mathematical or bitwise operation with assignment, updating the variable in place. They reduce redundancy by eliminating the need to repeat the variable name.

### Add and Assign (+=)

Example:

```
x = 5
```

```
x += 3  # Equivalent to x = x + 3; x becomes 8
```

### Subtract and Assign (-=)

Example:

```
x = 10
x -= 4  # Equivalent to x = x - 4; x becomes 6
```

### Multiply and Assign (*=)

Example:

```
x = 7
x *= 2  # Equivalent to x = x * 2; x becomes 14
```

### Divide and Assign (/=)

Example:

```
x = 15
x /= 3  # Equivalent to x = x / 3; x becomes 5.0
```

Floor Divide and Assign (//=)
Example:

```
x = 10
x //= 3  # Equivalent to x = x // 3; x becomes 3
```

### Modulus and Assign (%=)

Example:

```
x = 10
x %= 3  # Equivalent to x = x % 3; x becomes 1
```

### Exponentiate and Assign (**=)

Example:

```
x = 2
x **= 3  # Equivalent to x = x ** 3; x becomes 8
```

### Use Cases

- Simple assignment (=) is used to initialize or update variables.
- Compound assignment operators are useful when repeatedly updating a variable, such as in loops or iterative algorithms, as they make code more concise and readable.

Understanding these operators helps streamline operations and manage state effectively in Python programs.

# Comparison (Relational) Operators

Comparison operators in Python are used to compare two values. These operators evaluate the relationship between the operands and return a Boolean value: True if the condition is satisfied, and False otherwise. They work with numbers, strings, and other comparable types.

## Equality (==)

The equality operator checks if two values are equal. If the values are the same, it returns True; otherwise, it returns False.

Example:

```
result = 5 == 5  # True

result = "hello" == "world"  # False
```

## Inequality (!=)

The inequality operator checks if two values are not equal. It returns True if the values are different and False if they are the same.

Example:

```
result = 10 != 5  # True

result = 3 != 3  # False
```

## Greater Than (>)

The greater-than operator checks if the value on the left is greater than the value on the right.

Example:

```
result = 10 > 5  # True

result = 2 > 7  # False
```

## Less Than (<)

The less-than operator checks if the value on the left is smaller than the value on the right.

Example:

```
result = 3 < 8  # True

result = 9 < 4  # False
```

## Greater Than or Equal To (>=)

The greater-than-or-equal-to operator checks if the value on the left is greater than or equal to the value on the right.

Example:

```
result = 7 >= 7  # True
```

```
result = 4 >= 6  # False
```

## Less Than or Equal To (<=)

The less-than-or-equal-to operator checks if the value on the left is smaller than or equal to the value on the right.

Example:

```
result = 5 <= 10  # True

result = 12 <= 8  # False
```

Comparison operators are fundamental to Python programming, enabling conditional logic, filtering, and decision-making in code. Understanding their behavior with different types is key to writing clear and accurate programs.

## Range Checking

In Python, you can efficiently check if a value falls within a specific range using chained comparison operators. This is a clean and readable way to perform range checks without using multiple conditions.

Example: Checking if a Number is Within a Range

```
x = 10


if 5 < x < 15:
    print("x is within the range!")
else:
    print("x is out of range.")
```

The condition 5 < x < 15 is equivalent to (5 < x) and (x < 15), but it is more readable and avoids redundant comparisons. This checks if x is strictly greater than 5 and strictly less than 15.

Example: Checking if a Number is Outside a Range

```
y = 20


if y < 5 or y > 15:
    print("y is out of range!")
else:
    print("y is within the range.")
```

Here, y < 5 or y > 15 ensures that y is either too small or too large, making it fall outside the given range.

# Logical Operators

Logical operators in Python are used to combine conditional statements or evaluate multiple expressions to determine a Boolean result (True or False). These operators are essential in control flow, decision-making, and boolean logic.

## and

The and operator returns True if **both** operands are True. If any operand is False, the result is False.

Examples:

```
result = True and True  # True
result = True and False  # False
result = 5 > 3 and 10 < 20  # True, both conditions are True
result = 5 > 10 and 10 < 20  # False, first condition is False
```

Short-circuit behavior: If the first operand is False, Python skips evaluating the second operand because the result is already False.

## or

The or operator returns True if **at least one** of the operands is True. It only returns False if both operands are False.

Examples:

```
result = True or False  # True
result = False or False  # False
result = 5 > 3 or 10 > 20  # True, first condition is True
result = 5 > 10 or 10 > 20  # False, both conditions are False
```

Short-circuit behavior: If the first operand is True, Python skips evaluating the second operand because the result is already True.

## not

The not operator inverts the truth value of the operand. If the operand is True, it returns False, and vice versa.

Examples:

```
result = not True  # False
result = not False  # True
result = not (5 > 3)  # False, because 5 > 3 is True
```

### Truthy and Falsy Values

In Python, logical operators can also work with non-Boolean values based on their truthiness. Truthy values are considered True in a logical context, while falsy values are considered False. Common falsy values include:

- None
- False
- 0 (int or float)
- "" (empty string)
- [], {}, set() (empty collections)

Examples:

```
result = "" or "default"  # "default", because "" is falsy

result = 0 and 100  # 0, because 0 is falsy

result = not []  # True, because [] is falsy
```

# Identity Operators

Identity operators in Python are used to compare the memory locations of two objects. These operators check whether two variables reference the same object in memory rather than comparing their values.

### is

The is operator returns True if both variables refer to the same object in memory. It is useful when checking whether two variables point to the same instance.

Example:

x = [1, 2, 3]

y = x  # y refers to the same object as x

z = [1, 2, 3]  # z is a different object with the same content

print(x is y)  # True, both reference the same list object

print(x is z)  # False, different objects with the same values

### is not

The is not operator returns True if the variables reference different objects in memory.

Example:

x = [4, 5, 6]

```
y = [4, 5, 6]


print(x is not y)  # True, x and y have the same content but are different objects

print(x is y)  # False, they are not the same object
```

Python caches small integers and some immutable objects, so identity comparisons may sometimes yield unexpected results. For example, small integers and strings with the same value may be the same object due to interning.

Example:

```
a = 100

b = 100

print(a is b)  # True, because small integers are cached


a = 1000

b = 1000

print(a is b)  # False, because large integers are not cached
```

Identity operators are particularly useful for checking against None since None is a singleton in Python:

```
def check_value(value):
    if value is None:
        print("Value is None")
    else:
        print("Value is not None")


check_value(None)  # Output: Value is None

check_value(0)  # Output: Value is not None
```

# Membership Operators

Membership operators in Python are used to check whether a value exists within a sequence, such as a list, tuple, string, or dictionary.

## in

The in operator returns True if the specified value is present in the sequence.

Example:

```
fruits = ["apple", "banana", "cherry"]

print("banana" in fruits)  # True

print("grape" in fruits)  # False
```

It also works with strings:

```
text = "Hello, Python!"

print("Python" in text)  # True

print("Java" in text)  # False
```

For dictionaries, in checks whether a key exists:

```
person = {"name": "Alice", "age": 25}

print("name" in person)  # True

print("Alice" in person)  # False, because "Alice" is a value, not a key
```

### not in

The not in operator returns True if the specified value is not present in the sequence.

Example:

```
numbers = [1, 2, 3, 4]

print(5 not in numbers)  # True

print(3 not in numbers)  # False
```

# Ternary (Conditional) Operator

Python provides a compact way to write conditional expressions using the ternary operator, which follows the format:

```
x if condition else y
```

This means that if the condition evaluates to True, x is returned; otherwise, y is returned.

**Example 1: Assigning a value based on a condition**

```
age = 20

status = "Adult" if age >= 18 else "Minor"

print(status)  # Output: Adult
```

**Example 2: Finding the maximum of two numbers**

```
a = 10

b = 15

max_value = a if a > b else b
```

print(max_value)  # Output: 15

**Example 3: Checking if a number is even or odd**

num = 7

result = "Even" if num % 2 == 0 else "Odd"

print(result)  # Output: Odd

Ternary operators make conditional assignments and expressions more concise and readable, allowing for more compact code in simple decision-making scenarios.

# Bitwise Operators *

Bitwise operators in Python allow you to perform operations on the binary representations of integers. Unlike arithmetic operators that work on whole numbers, bitwise operators work at the level of individual bits, making them useful for low-level programming tasks, such as optimizing performance, working with flags, or manipulating bit masks. Python provides a range of bitwise operators, including AND, OR, XOR, NOT, and the bit shifting operators. Understanding these operators can help you manipulate data at the binary level and write more efficient code in certain scenarios.

## Bitwise AND (&)

The bitwise AND operator compares each bit of two integers and returns a new integer whose bits are set to 1 only when both corresponding bits in the operands are 1.

For example:

```
a = 10      # binary: 1010
b = 4       # binary: 0100
result = a & b  # binary: 0000, which is 0
```

## Bitwise OR (|)

The bitwise OR operator sets each bit of the result to 1 if at least one of the corresponding bits in the operands is 1.

For example:

```
a = 10      # binary: 1010
b = 4       # binary: 0100
result = a | b  # binary: 1110, which is 14
```

## Bitwise XOR (^)

The bitwise XOR (exclusive OR) operator compares the bits of two numbers and returns a new integer where each bit is set to 1 only if the corresponding bits in the two operands are different.

For example:

```
a = 10       # binary: 1010

b = 4        # binary: 0100

result = a ^ b  # binary: 1110, which is 14
```

## Bitwise NOT (~)

The bitwise NOT operator inverts the bits of its operand. In Python, the result of applying the bitwise NOT is the two's complement of the number, which is equivalent to -(n + 1).

For example:

```
a = 10       # binary: 1010

result = ~a  # equals -(10 + 1), so result is -11
```

## Left Shift (<<)

Shifts the bits to the left by a specified number of positions, effectively multiplying the number by 2 for each shift.

```
a = 3        # binary: 0011

result = a << 2  # shifts left by 2 positions, resulting in binary: 1100, which is 12
```

## Right Shift (>>)

Shifts the bits to the right by a specified number of positions, effectively performing an integer division by 2 for each shift.

```
a = 16       # binary: 10000

result = a >> 2  # shifts right by 2 positions, resulting in binary: 0100, which is 4
```

## Bitwise Assignment Operators

Similar to compound arithmetic operators, Python provides compound assignment operators for bitwise operations. These operators update the variable in place:

- &= performs bitwise AND and assigns the result.
- |= performs bitwise OR and assigns the result.
- ^= performs bitwise XOR and assigns the result.
- <<= performs left shift and assigns the result.
- >>= performs right shift and assigns the result.

For example:

```
a = 5        # binary: 0101

a <<= 1      # shifts bits left by 1, a becomes 10 (binary: 1010)
```

Bitwise operators offer a powerful way to work directly with the binary representations of integers. They enable fine-grained control over data manipulation, particularly in scenarios where performance and memory usage are critical, such as in systems programming or working with embedded devices. By mastering operators like AND, OR, XOR, NOT, and the bit shift operators, you can enhance your ability to perform low-level data manipulations in Python.

# Operator Precedence

Operator precedence determines the order in which operations are evaluated in expressions. Python evaluates operators with higher precedence first, and when operators have the same precedence, evaluation is left-to-right unless specified otherwise. Below is how precedence applies to Arithmetic, Assignment, Comparison, and Logical operators, listed in descending order of precedence:

## Arithmetic Operators

Arithmetic operators have the highest precedence among the specified types. Operations like exponentiation (**), multiplication (*), division (/), floor division (//), modulus (%), addition (+), and subtraction (-) are evaluated first in this order of precedence:

- Exponentiation (**) has the highest precedence among arithmetic operators.
- Multiplication (*), Division (/), Floor Division (//), and Modulus (%) come next.
- Addition (+) and Subtraction (-) have the lowest precedence in this group.

Example:

```
result = 2 + 3 * 4  # Multiplication first, result is 14

result = (2 + 3) * 4  # Parentheses change precedence, result is 20
```

## Bitwise Operators *

Bitwise operators come next in Python's operator precedence hierarchy and are evaluated in the following specific sequence.

- left shift (<<) and right shift (>>) operators
- bitwise AND (&) operator
- bitwise XOR (^) operator
- the bitwise OR (|) operator is evaluated.

```
result = 5 + 3 << 2 & 12 ^ 7 | 1
print(result)

result = 1 | 7 ^ 12 & 2 << 3 + 5
print(result)
```

Both lines expressions give the same result because of operator precedence

## Comparison Operators

Comparison operators, such as ==, !=, >, <, >=, and <=, are evaluated after arithmetic operations. These operators are used to compare values and return True or False.

Example:

```
result = 5 + 3 > 6  # Addition happens first, then comparison, result is True
```

## Identity and Membership Operators

The identity operators, `is` and `is not` and the membership operators `in` and `not in` are evaluated next.

## Logical Operators

Logical operators not, and, and or come after arithmetic and comparison operators. Among these:

- not has the highest precedence.
- and is evaluated after not.
- or has the lowest precedence.

Example:

```
result = not 3 > 5 and 2 + 2 == 4  # `not` first, then comparison and arithmetic, finally `and`
```

## Ternary Operator

The ternary operator (`x if condition else y`) has very low precedence, even lower than logical operators like `and` and `or`. This means that it is evaluated last, ensuring that the condition and both possible values are fully computed before making a choice.

## Assignment Operators

Assignment operators like =, +=, -=, etc., have the lowest precedence. They are evaluated after all other operators, ensuring that the right-hand side of the expression is fully computed before the assignment.

Example:

```
x = 3 + 5 * 2  # Arithmetic evaluated first, then assignment, x is 13

x += 4 > 3  # Comparison (4 > 3) is True (1), so x is updated to 14
```

## Combining Operators

When multiple operator types appear in an expression, Python follows the precedence rules strictly:

```
result = 3 + 5 > 6 and 2 * 2 == 4
```

# Step-by-step:

# 1. Arithmetic: 3 + 5 = 8, 2 * 2 = 4

# 2. Comparison: 8 > 6 (True), 4 == 4 (True)

# 3. Logical: True and True = True

### Overruling precedence

Parentheses can always be used to override default precedence and make expressions clearer. For example:

```
result = (3 + 5) > (6 and 2)  # Parentheses ensure clarity and enforce specific order
```

Understanding operator precedence ensures that complex expressions are evaluated as intended, preventing logic errors in Python programs.

# Standard Input and Output (stdin and stdout) in Python

In Python, stdin (standard input) and stdout (standard output) are fundamental concepts for interacting with the user or external processes. They allow data to be read from an input source (such as user input from the keyboard) and written to an output destination (such as the console). While they are not operators in the traditional sense, they facilitate communication between the program and its execution environment, much like how operators facilitate transformations of data within a program.

The sys module provides direct access to sys.stdin and sys.stdout, which act as file-like objects for reading and writing data. Normally, input() is used for reading user input, which internally reads from sys.stdin, and print() is used for output, which writes to sys.stdout. However, directly interacting with sys.stdin and sys.stdout allows for more flexibility, such as redirecting input and output or handling data in a more controlled manner.

To read from stdin, you can use sys.stdin.read() or sys.stdin.readline(). The former reads the entire input as a single string, while the latter reads a single line at a time. For example, import sys followed by data = sys.stdin.read() will capture all the input until an end-of-file (EOF) signal is encountered. Similarly, line = sys.stdin.readline() will read only one line at a time, which can be useful when processing large inputs.

For writing output, sys.stdout.write() functions similarly to print(), but without automatically appending a newline. If you use sys.stdout.write("Hello, world!"), it will print "Hello, world!" without a newline, whereas print("Hello, world!") would include a newline by default.

Redirection of stdin and stdout is particularly useful when handling file input and output. By setting sys.stdin to a file object, the program can read input from a file instead of user input. Similarly, redirecting sys.stdout to a file allows output to be written to a file rather than appearing on the console. This can be done using sys.stdin = open("input.txt") to read from a file or sys.stdout = open("output.txt", "w") to write to a file.

In concurrent programming, particularly in Rust, controlling input and output is essential to avoid blocking operations. Similarly, in Python, managing stdin and stdout efficiently is crucial in multi-

threaded programs, where improper handling of shared resources can lead to race conditions or deadlocks. To mitigate such issues, using thread-safe methods like sys.stdout.flush() ensures that output is written immediately, preventing buffering delays when multiple threads are involved.

By leveraging sys.stdin and sys.stdout, Python provides precise control over input and output operations, making it possible to handle a wide range of data processing scenarios, from simple user interactions to complex pipeline-based data flows.

# Summary

Operators in Python are fundamental tools that enable various computations and logical manipulations on data. They form the basis of arithmetic operations, comparisons, logical evaluations, and more, allowing Python to process expressions efficiently and concisely.

Python provides a rich set of operators categorized into arithmetic, assignment, comparison, logical, identity, membership, bitwise, and ternary (conditional) operators. Each type serves a specific purpose, from performing basic mathematical calculations to making complex logical decisions. Operators follow precedence rules, which dictate the order of evaluation in expressions, ensuring that certain operations are executed before others.

Arithmetic operators handle standard mathematical computations, while assignment operators simplify updating variable values. Comparison operators allow evaluation of relationships between values, producing Boolean results. Logical operators enable combining multiple conditions, whereas identity and membership operators help in verifying object references and sequence containment. Bitwise operators operate at the binary level, making them useful for low-level programming and performance optimizations. The ternary operator provides a shorthand for conditional expressions, improving code readability.

Operator precedence ensures that expressions are evaluated correctly, with arithmetic operations taking priority over comparison, followed by logical and assignment operators. Parentheses can be used to override default precedence to enforce a specific evaluation order.

Additionally, Python's standard input (sys.stdin) and output (sys.stdout) facilitate interaction with users and external processes, providing controlled ways to handle data flow.

Understanding Python’s operators and their precedence is crucial for writing efficient, readable, and logically sound programs. Mastering these concepts empowers developers to construct complex expressions, implement decision-making logic, and perform advanced data manipulations effectively.

# Exercises

1. Write a Python program to calculate the sum, difference, product, and quotient of two numbers. Ensure that the output displays the operation and result clearly.
2. Create a program that uses comparison operators to compare three numbers. Print whether the numbers are equal, greater, or smaller relative to each other.

3. Implement a program that demonstrates the behavior of compound assignment operators by starting with an initial value and successively applying +=, -=, *=, and /= operators. Print the value at each step.
4. Create a program that evaluates a complex expression involving arithmetic, comparison, and logical operators. Show step-by-step how Python determines the result by using parentheses to clarify precedence.
5. Evaluate the following expression using pen and paper, following the correct order of operations. Then, verify your result by writing and running a Python program: result = (8 + 2 * 5) - (10 // 3) + (3 ** 2) % 4
6. Write a program that evaluates two expressions to show how operator precedence works. Evaluate: expr1 = 4 + 6 * 2 and expr2 = (4 + 6) * 2. Print both results, and include comments or print statements explaining why they differ.
7. * Write a program that declares two integers a = 12 and b = 5, then applies the bitwise AND (a & b), OR (a | b), and XOR (a ^ b). Print the results in both decimal and binary format.
8. * Write a program that takes the integer num = 7, applies the bitwise NOT (~num), and prints both the original and negated values in decimal and binary form.
9. * Write a program that takes the integer x = 8, shifts its bits left by 2 positions (x << 2) and right by 2 positions (x >> 2), and prints the results in both decimal and binary format.
10. * Write a program that evaluates the expression (a & b) | (c ^ d) << 1 using a = 6, b = 3, c = 5, and d = 2, then prints the final result.