Arithmetic Operators - Python

Premanand S

Assistant Professor,
School of Electronics Engineering
Vellore Institute of Technology
Chennai Campus

premanand.s@vit.ac.in

December 2, 2024

Arithmetic Operators in Python

- Arithmetic operators are used to perform mathematical operations.
- Python supports the following arithmetic operators:

Operator	**Description**
+	Addition
−	Subtraction
*	Multiplication
/	Division
//	Floor Division
%	Modulus
**	Exponentiation

Addition Operator (+)

- Adds two operands.

Example (Python Code)

```
x = 10
y = 5
print(x + y)  # Output: 15
```

Subtraction Operator (–)

- Subtracts the second operand from the first.

Example (Python Code)

```
x = 10
y = 5
print(x - y)  # Output: 5
```

Multiplication Operator (*)

- Multiplies two operands.

Example (Python Code)

```
x = 10
y = 5
print(x * y)  # Output: 50
```

Division Operator (/)

- Divides the first operand by the second, returning a float.

Example (Python Code)

```
x = 10
y = 4
print(x / y)  # Output: 2.5
```

Floor Division Operator (//)

- Divides the first operand by the second and returns the largest integer less than or equal to the result.

Example (Python Code)

```
x = 10
y = 4
print(x // y)  # Output: 2
```

Modulus Operator (%)

- Returns the remainder when the first operand is divided by the second.

Example (Python Code)

```
x = 10
y = 3
print(x % y)  # Output: 1
```

Exponentiation Operator (**)

- Raises the first operand to the power of the second.

Example (Python Code)

```
x = 2
y = 3
print(x ** y)  # Output: 8
```

Common Errors and Notes

- **Division by Zero:** Dividing by zero raises a ZeroDivisionError.
- **Type Errors:** Mixing incompatible types (e.g., str + int) raises a TypeError.
- **Floating-Point Precision:** Floating-point operations may result in small precision errors.

Example (Python Code)

```
print(10 / 0)  # ZeroDivisionError

print("10" + 5)  # TypeError
```

Operator Precedence in Python

- **Operator Precedence** determines the order in which operators are evaluated in an expression.
- Operators with higher precedence are evaluated first.
- Arithmetic operators in Python follow a specific precedence order.

Arithmetic Operators Precedence

Operator	Description
$**$	Exponentiation (Power)
$+, -$	Unary plus and minus (for positive/negative numbers)
$*, /, //, \%$	Multiplication, Division, Floor Division, Modulus
$+, -$	Addition and Subtraction

Order of Precedence

- Exponentiation ($**$) has the highest precedence.
- Unary operators ($+$ and -) come next.
- Multiplication, Division, Floor Division, and Modulus have the same precedence and are evaluated from left to right.
- Addition and Subtraction have the lowest precedence and are evaluated last.

Example 1: Exponentiation before Other Operations

- The operator `**` (exponentiation) is evaluated first.
- Result: `2 + 9 = 11`.

Example (Python Code)

```
print(2 + 3 ** 2)  # Output: 11
```

Example 2: Multiplication and Division Before Addition/Subtraction

- The operator * (multiplication) is evaluated first.
- Result: `3 + 8 = 11`.

Example (Python Code)

```
print(3 + 2 * 4)  # Output: 11
```

Example 3: Floor Division and Modulus

- Floor division (//) and modulus (%) are evaluated before addition or subtraction.

Example (Python Code)

```
print(7 // 3)  # Output: 2 (Floor division)
print(7 % 3)   # Output: 1 (Modulus)
```

Example 4: Parentheses Overriding Precedence

- Parentheses have the highest precedence, so `(2 + 3)` is evaluated first.
- Result: `5 * 4 = 20`.

Example (Python Code)

```
print((2 + 3) * 4)  # Output: 20
```

Example 5: Left-to-Right Evaluation

- * and / have the same precedence and are evaluated from left to right.
- Result: 6 / 4 = 1.5.

Example (Python Code)

```
print(2 * 3 / 4)  # Output: 1.5
```

Associativity in Exponentiation

- Exponentiation (**) has right-to-left associativity.
- Result: `2 ** (3 ** 2)` → `2 ** 9` = 512.

Example (Python Code)

```
print(2 ** 3 ** 2)  # Output: 512
```

Combining Arithmetic with Logical Conditions

- Arithmetic operators can be combined with logical operators to form complex expressions.

Example (Python Code)

```
x, y = 10, 20
if x + y > 25 and y - x < 15:
    print("Condition met!")
```

Arithmetic with Strings

- **Concatenation (+)**: Combines two strings.
- **Repetition (∗)**: Repeats a string multiple times.

Example (Python Code)

```
print("Python" + "Programming")  # Output: PythonProgramming
print("Learn! " * 3)             # Output: Learn! Learn! Lear
```

Augmented Assignment Operators

- Combine arithmetic operations with assignment for concise code.
- Examples: +=, −=, ∗=, /=, etc.

Example (Python Code)

```
x = 10
x += 5  # Equivalent to x = x + 5
print(x)  # Output: 15
```

Arithmetic with Complex Numbers

- Python's `complex` type supports arithmetic operations.
- Complex numbers are represented as `a + bj`.

Example (Python Code)

```
a = 2 + 3j
b = 1 - 4j
print(a + b)  # Output: (3-1j)
print(a * b)  # Output: (14-5j)
```

Division Behavior in Python 2 vs Python 3

- In Python 2, / performs integer division if operands are integers.
- In Python 3, / always performs floating-point division.

Example (Python Code)

```
print(10 / 3)   # Output: 3.3333333333333335
print(10 // 3)  # Output: 3 (Floor division)
```

Using `decimal.Decimal` for High-Precision Arithmetic

- Use the `decimal` module for precise floating-point arithmetic.

Example (Python Code)

```
from decimal import Decimal
a = Decimal(’0.1’)
b = Decimal(’0.2’)
print(a + b)  # Output: 0.3
```

Arithmetic Operations on NumPy Arrays

- Arithmetic operators work element-wise on NumPy arrays.

Example (Python Code)

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b)  # Output: [5 7 9]
print(a * b)  # Output: [4 10 18]
```

Using math Module for Advanced Arithmetic

- The `math` module provides advanced arithmetic functions like:
  - `math.sqrt(x)`: Square root.
  - `math.pow(x, y)`: Power.
  - `math.fmod(x, y)`: Floating-point modulus.

Example (Python Code)

```
import math
print(math.sqrt(16))  # Output: 4.0
print(math.pow(2, 3)) # Output: 8.0
```

Arithmetic with Booleans

- In Python, True is treated as 1, and False as 0.
- Arithmetic operations can be performed on boolean values.

Example (Python Code)

```
print(True + True)  # Output: 2
print(True * False) # Output: 0
```

Handling Division Errors

- Dividing by zero raises a `ZeroDivisionError`.
- Use `try-except` to handle such cases gracefully.

Example (Python Code)

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```

Special Cases and Fun Facts

- **Integer Overflow**: Python handles large integers seamlessly.
- **Floating-Point Precision Issues**: Precision limitations can lead to unexpected results.

Example (Python Code)

```
print(10**100)  # Output: 1 followed by 100 zeros

print(0.1 + 0.2)  # Output: 0.30000000000000004
```

Brush it up!

What are Arithmetic Operators in Python?

Definition: Arithmetic operators are symbols used to perform mathematical operations on numbers or variables. They include:

- +: Addition
- −: Subtraction
- ∗: Multiplication
- /: Division
- //: Floor Division
- %: Modulus (Remainder)
- ∗∗: Exponentiation (Power)

Example (Python Code)

```python
x, y = 10, 5
print(x + y)  # Output: 15 (Arithmetic)
print(x > y)  # Output: True (Comparison)
```

Difference Between / and //

- **/ (Division)**: Performs floating-point division. Always returns a float.
- **// (Floor Division)**: Performs integer division and truncates the decimal part. Always returns an integer for integers and a float for floats.

Use Case:

Use ‘/‘ when precise results are required (e.g., in scientific calculations).

Use ‘//‘ when you only need the whole number part of the division.

Example (Python Code)

```
# Using /
print(10 / 3)   # Output: 3.3333333333333335
print(10.0 / 3) # Output: 3.3333333333333335
# Using //
print(10 // 3)  # Output: 3
print(10.0 // 3) # Output: 3.0
```

Why is ** Evaluated Before *?

Explanation: - Python follows a strict operator precedence hierarchy. - ** (Exponentiation) has higher precedence than * (Multiplication), so it is evaluated first. **Example:**

- Expression: `2 + 3 ** 2`
- Steps:
  1. `3 ** 2` → `9` (Exponentiation)
  2. `2 + 9` → `11` (Addition)

Operator Precedence: - Higher precedence operators are evaluated before lower precedence ones. - Use parentheses to override precedence and control evaluation order.

Example (Python Code)

```
result = 2 + 3 ** 2
print(result)  # Output: 11
print((2 + 3) ** 2)  # Output: 25
```

Calculate Compound Interest

Problem: Write a program to calculate compound interest for a given principal, rate, and time.

Example:

- Use the formula: `A = P * (1 + r) ** t`, where:
  - `P` is the principal amount.
  - `r` is the annual interest rate (in decimal).
  - `t` is the time in years.

Example (Python Code)

```
principal = float(input("Enter principal: "))
rate = float(input("Enter annual rate (%): ")) / 100
time = int(input("Enter time in years: "))
amount = principal * (1 + rate) ** time
interest = amount - principal
print(f"Compound Interest: ${interest:.2f}")
```

Determine Divisibility by Both 3 and 5

Problem: Create a script to check if a number is divisible by both 3 and 5 without using logical operators.

Example:

- Use multiplication of boolean results to simulate and.

Example (Python Code)

```
num = int(input("Enter a number: "))
# Using multiplication for logical AND
if (num % 3 == 0) * (num % 5 == 0):
    print(f"{num} is divisible by both 3 and 5.")
else:
    print(f"{num} is not divisible by both 3 and 5.")
```

Calculate Area of a Triangle

Problem: Calculate the area of a triangle given its base and height using arithmetic operators.

Example:

- Use the formula: `Area = 0.5 * base * height`.

Example (Python Code)

```
base = float(input("Enter base of the triangle: "))
height = float(input("Enter height of the triangle: "))
area = 0.5 * base * height
print(f"Area of the triangle: {area:.2f}")
```

Combining Arithmetic and Logical Operators

Explanation: Arithmetic operators can be used in conjunction with logical operators (`and`, `or`, `not`) to form complex conditions for decision-making.
Example: Check if the sum of two numbers is greater than 10 and their product is even.

Example (Python Code)

```
x, y = 4, 6
if (x + y > 10) and ((x * y) % 2 == 0):
    print("Conditions met!")
else:
    print("Conditions not met!")
```

Explanation of Logic:

- Arithmetic operators calculate the sum and product.
- Logical `and` combines the two conditions into one.

Check if a Number is a Perfect Square

Problem: Write a program to check if a number is a perfect square using `**` and `math.sqrt()`.

Example (Python Code)

```python
import math

num = int(input("Enter a number: "))
if math.sqrt(num) == int(math.sqrt(num)):
    print(f"{num} is a perfect square.")
else:
    print(f"{num} is not a perfect square.")
```

Explanation:

- `math.sqrt()` computes the square root of the number.
- `int()` checks if the square root is an integer.

Handling Floating-Point Precision Errors

Problem: Manage precision issues when performing operations like `0.1 + 0.2`. Floating-point numbers in Python may have small precision errors due to how they are stored in memory. Use the `decimal` module to handle such cases accurately.

Example (Python Code)

```
from decimal import Decimal
a = Decimal(’0.1’)
b = Decimal(’0.2’)
result = a + b
print(result)  # Output: 0.3
```

Why Use `Decimal`?

- It provides precise decimal arithmetic for financial or scientific applications.
- Avoids common floating-point errors.

Swap Two Numbers Without a Third Variable

Problem: Swap two numbers without using a third variable. **Solution:** Use arithmetic operators such as addition and subtraction or XOR.

Example (Python Code)

```python
# Using addition and subtraction
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
a = a + b
b = a - b
a = a - b
print(f"Swapped values: a = {a}, b = {b}")
```

Why Arithmetic Operators?

- Avoids using extra memory for a third variable.
- Efficient for simple operations like swapping values.

Efficiency of Augmented Assignment Operators

Problem: How can augmented assignment operators improve loop efficiency?

- Augmented assignment operators like +=, −=, etc., modify the variable in place.
- Reduces redundancy and enhances readability.

Example: Calculate the sum of numbers in a range.

Example (Python Code)

```
total = 0
for i in range(1, 6):  # Summing first 5 numbers
    total += i  # Equivalent to total = total + i
print(f"Sum of numbers: {total}")
```

Benefits:

- Faster and cleaner code.
- Reduces temporary assignments in memory.

High-Precision Financial Calculations

Problem: Use Python's `decimal.Decimal` to perform accurate calculations for financial data. Floating-point arithmetic can introduce errors due to binary representation of decimals.`decimal.Decimal` ensures precision by using a base-10 representation.

Example (Python Code)

```
from decimal import Decimal
price_per_item = Decimal(’19.99’)
quantity = Decimal(’3’)
total = price_per_item * quantity
print(f"Total: ${total}")  # Output: Total: $59.97
```

Why Use `Decimal`?

- Avoids common floating-point errors.
- Critical for financial and scientific applications.
- Supports high-precision arithmetic operations.