CSI-160 Python Programming
Python Operators

Vikas Thammanna Gowda

07/10/2025

Contents

1 Introduction

In Python (as in most programming languages), an operator is a special symbol or keyword that tells the interpreter to perform a specific computation or action on one or more values (called operands). You can think of operators as the "verbs" in the language—they take inputs, do something with them, and produce an output.

Key points about operators:

- Operands are the values or variables you apply the operator to.
- Unary operators work on a single operand (e.g. `-x` negates `x`).
- Binary operators work on two operands (e.g. `x + y` adds `x` and `y`).
- Operators always return a result, which you can assign to a variable or use directly in an expression.

Why operators matter:

- They let you compute numeric results (like adding `3 + 5`).
- They let you compare values (like checking if `age >= 18`).
- They let you combine logical conditions (like `is_member and has_paid`).
- They let you manipulate data structures (e.g. checking membership with `'a' in my_list`).

2 Arithmetic Operators

In programming, arithmetic operators let you perform mathematical calculations on numbers. Just as you use $+$, $-$, $\times$, and $\div$ on paper, Python provides a set of symbols that tell the computer how to combine or transform numeric values. Understanding these operators is essential for writing programs that handle everything from simple sums to complex formulas.

2.1 Overview

Operator	Name	Description
`+`	Addition	Adds two numbers
`-`	Subtraction	Subtracts the right number from the left
`*`	Multiplication	Multiplies two numbers
`/`	Division	Divides left by right, results in a floating-point number
`//`	Floor Division	Divides left by right, then rounds down to the nearest integer
`%`	Modulus (Remainder)	Returns the remainder after division
`**`	Exponentiation	Raises left number to the power of the right number
`()`	Parentheses (Grouping)	Changes the order of operations

Table 1: Arithmetic Operators

2.2 Types

1. **Addition (+)**

   What it does: Combines two numbers into their sum.

   Example:

```
value_1 = 7
value_2 = 5
result = value_1 + value_2
print("The result of", value_1, "+", value_2, "is", result)
```

   Output:

```
The result of 7 + 5 is 12
```

2. **Subtraction (-)**

   What it does: Takes one number away from another.

   Example:

```
value_1 = 7
value_2 = 5
result = value_1 - value_2
print("The result of", value_1, "-", value_2, "is", result)
```

   Output:

```
The result of 7 - 5 is 2
```

3. **Multiplication (*)**

   What it does: Finds the product of two numbers.

   Example:

```
value_1 = 7
value_2 = 5
result = value_1 * value_2
print("The result of", value_1, "*", value_2, "is", result)
```

   Output:

```
The result of 7 * 5 is 35
```

4. **Division (/)**

   What it does: Divides the first number by the second, always returning a float.

   Example:

```
value_1 = 8
value_2 = 2
result = value_1 / value_2
print("The result of", value_1, "-", value_2, "is", result)
```

   Output:

```
The result of 8 / 2 is 4.0
```

   Note: Even though 8 divided by 2 is 4, Python shows it as 4.0 because division yields a floating-point number.

5. **Floor Division (//)**

   What it does: Divides and then "floors" (rounds down) the result to the nearest integer.

   Example:

```
value_1 = 9
value_2 = 2
result = value_1 // value_2
print("The result of", value_1, "//", value_2, "is", result)
```

   Output:

```
The result of 9 // 2 is 4
```

   Note: 9 divided by 2 is 4.5, but floor division drops the .5 to give 4.

6. **Modulus (%)**

   What it does: Gives the remainder after division.

   Example:

```
value_1 = 9
value_2 = 2
result = value_1 % value_2
print("The result of", value_1, "%", value_2, "is", result)
```

   Output:

```
The result of 9 % 2 is 1
```

   Note: 9 divided by 2 leaves a remainder of 1.

7. **Exponentiation (`**`)**

   What it does: Raises one number to the power of another.

Example:

```
base = 3
exponent = 4
result = base ** exponent
print("The result of", base, "to the power of", exponent, "is",
    result)
```

Output:

```
The result of 3 to the power of 4 is 81
```

8. **Grouping with Parentheses (`()`)**

   What it does: Forces operations inside parentheses to be computed first.

Example:

```
result1 = 2 + 3 * 4
result2 = (2 + 3) * 4
print("The result of 2 + 3 * 4 is ",result1)
print("The result of (2 + 3) * 4 is ",result2)
```

Output:

```
The result of 2 + 3 * 4 is 14
The result of (2 + 3) * 4 is 20
```

2.3 Order of Operations

Python follows the same precedence rules as standard mathematics:

1. Parentheses
2. Exponentiation
3. Multiplication, Division, Floor Division, Modulus (left to right)
4. Addition and Subtraction (left to right)

Use parentheses to make your intentions clear and to override this default order.

Example:

```
value_1 = 5
value_2 = 2
value_3 = 3

result1 = value_1 + value_2 * value_3 ** 2 // 4 % 3

result2 = ((value_1 + value_2) * (value_3 ** 2)) // (4 % 3)

print("result1:", result1)
print("result2:", result2)
```

Output:

```
result1: 6
result2: 63
```

Breakdown for `result1`:

1. $value\_3 * *2 \rightarrow 3^2 = 9$
2. $value\_2 \times 9 \rightarrow 2 \times 9 = 18$
3. 18 `//` $4 \rightarrow 4$ (floor division since $18/4 = 4.5$)
4. $4 \,\%\, 3 \rightarrow 1$
5. $value\_1 + 1 \rightarrow 5 + 1 = 6$

Breakdown for `result2`:

1. Parentheses force $(value\_1 + value\_2) = 7$ and $value\_3 * *2 = 9$
2. Multiply: $7 \times 9 = 63$
3. Compute $4 \,\%\, 3 = 1$
4. Floor divide: 63 `//` $1 = 63$ (floor division by 1 leaves it unchanged)

2.4 Common Pitfalls

1. Division vs. floor division: Remember / always gives a float, while // gives an integer or float depending on the type of operands.

Example:

```
print("9/4 =", 9/4)
print("9//4 =", 9//4)
print("9.0/4 =", 9.0/4)
```

Output:

```
9/4 = 2.25
9//4 = 2
9.0//4 = 2.0
```

2. Negative numbers with floor division: Floor division rounds down, not toward zero.

Example:

```
value_1 = -3
value_2 = 2
result = value_1 // value_2
print("The result of", value_1, "//", value_2, "is", result)
```

Output:

```
The result of -3 // 2 is -2
```

3 Relational (Comparision) Operators

In programming, relational operators (also called comparison operators) let you compare two values. Instead of doing math, these operators ask questions like "Is one number bigger than the other?" or "Are these two values equal?" Each comparison returns a **Boolean** result: `True` if the statement is correct, or `False` if it's not. Relational operators are the **foundation of decision-making** in your programs, because they let you control what code runs next.

3.1 Overview

Operator	Name	What it Checks	Example
`==`	Equal to	Are the two values the same?	`5 == 5` → `True`
`!=`	Not equal to	Are the two values different?	`5 != 3` → `True`
`>`	Greater than	Is the left value larger than the right?	`7 > 10` → `False`
`<`	Less than	Is the left value smaller than the right?	`3 < 8` → `True`
`>=`	Greater than or equal to	Is the left value at least as big as the right?	`4 >= 4` → `True`
`<=`	Less than or equal to	Is the left value at most as big as the right?	`2 <= 1` → `False`

Table 2: Relational Operators

Example 1:

```
# Equal to (==)
actual_password = "SecurePass123"
entered_password = "securepass123"
is_password_correct = entered_password == actual_password

# Not equal to (!=)
required_subject = "Mathematics"
chosen_subject = "Physics"
is_different_subject = chosen_subject != required_subject

# Greater than (>)
student_score = 85
passing_score = 70
has_exceeded_passing = student_score > passing_score


# Print all results
print("Password correct?", is_password_correct)
print("Subject differs?", is_different_subject)
print("Score exceeds passing?", has_exceeded_passing)
```

Output:

```
Password correct? False
Subject differs? True
Score exceeds passing? True
```

Example 2:

```
# Less than (<)
current_temperature = 18.0  # in degree C
comfortable_temperature = 20.0
is_too_cold = current_temperature < comfortable_temperature

# Greater than or equal to (>=)
min_height_cm = 150
applicant_height_cm = 150
meets_height_requirement = applicant_height_cm >= min_height_cm

# Less than or equal to (<=)
max_team_members = 5
group_size = 4
within_team_limit = group_size <= max_team_members

# Chained comparison
age = 16
is_in_teen_range = 13 <= age <= 19

# Print all results
print("Too cold?", is_too_cold)
print("Meets height requirement?", meets_height_requirement)
print("Within team limit?", within_team_limit)
print("Is in teen age range?", is_in_teen_range)
```

Output:

```
Too cold? True
Meets height requirement? True
Within team limit? True
Is in teen age range? True
```

3.2 Common Pitfalls

1. **Accidental assignment vs. comparison:** In Python, `=` is assignment, `==` is comparison.
2. **Comparing different types and Floating-point precision issues:**

Example:

```
print(5 == "5")  # integer 5 is not the same as string "5"
print(0.1 + 0.2 == 0.3)  # Often False due to how floats are stored
```

Output:

```
False
False
```

Note: If you need to compare floats, consider checking if they're "close enough" using a small tolerance. (We will revisit when handling selection statements.)

4 Assignment Operator

In Python, assignment operators are used to store or update values in variables. The simplest assignment operator is `=`, which takes the value on the right and places it into the variable on the left. Beyond that, Python provides augmented assignment operators like `+=` and `*=` that combine an operation with assignment, letting you update a variable more concisely.

4.1 Simple Assignment Operator (=)

It assigns the right-hand value to the variable on the left hand, i.e., first the value on the RHS of `=` is computed and then assigned to the variable on the LHS of `=`.

Example:

```
length = 5                                  # line 1
width = 4                                   # line 2
area_rectangle = length * width             # line 3
print("The area is", area_rectangle)        # line 4
```

Output:

```
The area is 20
```

Breakdown for `area_rectangle`:

1. In **line 1**, **5** is assigned to the variable **`length`**.
2. In **line 2**, **4** is assigned to the variable **`length`**.
3. In **line 3**, the value of **`length * width`** (`5 * 4`)is first computed, i.e., **20**.
   Then, **20** is assigned to the variable **`area_rectangle`**.

4.2 Augmented Assignment Operator

Augmented assignment operators perform an operation and store the result back into the same variable.

Augmented Operator	Equivalent To	What It Does
`+=`	`x = x + y`	**Add** `y` to `x`, then assign back to `x`
`-=`	`x = x - y`	**Subtract** `y` from `x`, then assign back to `x`
`*=`	`x = x * y`	**Multiply** `x` by `y`, then assign back to `x`
`/=`	`x = x / y`	**Divide** `x` by `y`, then assign back to `x`
`//=`	`x = x // y`	**Floor-divide** `x` by `y`, then assign back to `x`
`%=`	`x = x % y`	**Modulo** (remainder) of `x` ÷ `y`, then assign
`**=`	`x = x ** y`	**Exponentiate** `x` to power `y`, then assign

Table 3: Augmented assignment operators

Syntax:

```
variable <op>= expression
```

The above syntax is equivalent to:

```
variable = variable <op> expression
```

Example (Equivalent code and Output are added as comments):

```
points = 10
print("Initial points:", points)      # Initial points: 10

# 1. +=  (add and assign)
points += 5 # points = points + 5
print("After +=5:", points)           # After +=5: 15

# 2. -=  (subtract and assign)
points -= 3 # points = points - 3
print("After -=3:", points)           # After -=3: 12

# 3. *=  (multiply and assign)
multiplier = 2
points *= multiplier
print("After *=2:", points)           # After *=2: 24

# 4. /=  (divide and assign)
points /= 4
print("After /=4:", points)           # After /=4: 6.0

# 5. //= (floor-divide and assign)
points //= 4
print("After //=4:", points)          # After //=4: 1.0

# 6. %=  (modulus and assign)
points = 10
points %= 3
print("After %=3:", points)           # After %=3: 1

# 7. **= (exponentiate and assign)
value = 3
value **= 3
print("After **=3:", value)           # After **=3: 27
```

4.3 Why Use Augmented Assignment?

- Conciseness: `x += 1` is shorter than `x = x + 1`.
- Clarity: It clearly shows "update this variable by doing ...".
- Performance: Under the hood, it can be slightly more efficient, especially with mutable types.

5 Logical Operators

In Python, logical operators let you combine or invert Boolean expressions—statements that are either `True` or `False`. They're fundamental for making decisions in your code (e.g., "if this and that are both true…"). Python provides three logical operators:

- `and`
- `or`
- `not`

Each one follows simple rules (shown below in truth tables) and can be used in conditions, loops, and anywhere you need to test multiple criteria.

5.1 Overview

Operator	Meaning	Result when applied
`and`	**Logical AND**	`True` only if **both** operands are `True`
`or`	**Logical OR**	`True` if **at least one** operand is `True` (or both)
`not`	**Logical NOT**	`True` becomes `False`, and `False` becomes `True`

Table 4: Logical operators

5.2 Logical AND (`and`)

What it does: Returns `True` only if **both** operands are `True`.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Table 5: Truth table for the logical `and` operator

Note:

- **A** and **B** can be the result of a logical operators.
- `and` stops evaluation as soon as it sees a `False` (because the whole expression can't be `True` anymore).

Example:

```
has_id_card = True
has_ticket  = False
can_enter = has_id_card and has_ticket
print("Can this person enter?", can_enter)

value_1 = 4 >= 4
value_2 = (10 % 3) == (3 // 2)
result = value_1 and value_2
print("The result is:", result)
```

Output:

```
Can this person enter? False
The result is: True
```

5.3 Logical OR (`or`)

What it does: Returns `True` if at least one operand is `True`.

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Table 6: Truth table for the logical `or` operator

Note:

- **A** and **B** can be the result of a logical operators.
- `or` stops as soon as it sees a `True` (because the whole expression is already `True`).

Example 1:

```
has_id_card = True
has_ticket  = False
can_enter = has_id_card or has_ticket
print("Can this person enter?", can_enter)

value_1 = 4 >= 4
value_2 = (10 % 3) == (4 // 2)
result = value_1 and value_2
print("The result is:", result)
```

Output:

```
Can this person enter? True
The result is: False
```

Example 2:

```
age = 17
has_parent_permission = True

can_watch_movie = (age >= 18) or (age >= 13 and has_parent_permission)
print("Enery into movies:", can_watch_movie)
```

Output:

```
Enery into movies: True
```

5.4 Logical NOT (`not`)

What it does: Inverts the boolean value.

A	**not A**
`True`	`False`
`False`	`True`

Table 7: Truth table for the logical `not` operator

Note: **A** can be the result of a logical operators.

Example:

```
is_raining = False

will_sunshine = not is_raining
print(will_sunshine)
```

Output:

```
True
```

6 Special Operators

6.1 Membership Operators

Test for membership in a sequence (string, list, tuple, etc.).

- `in` : True if value is found
- `not in` : True if value is not found

Example:

```
fruits = ['apple','banana','cherry']
check_1 = 'banana' in fruits
check_2 = 'pear' not in fruits
print(check_1)
print(check_2)
```

Output:

```
True
True
```

6.2 Identity Operators

Compare object identities (not values).

- `is` : True if both refer to the same object
- `is not` : True if they refer to different objects

Example:

```
value_a = [1,2,3]
value_b = value_a
value_c = [1,2,3]
print(value_a is value_b)
print(value_a is not value_c)
```

Output:

```
True
True
```