Python Operators and Expressions

Make Python Calculate, Compare and Decide

Learning Objectives

By the end of this guide, you will understand:

- What operators and expressions are
- Arithmetic operators in Python
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators
- Operator precedence and associativity
- How expressions are evaluated
- How to combine operators safely
- Common operator mistakes
- How operators appear in real applications
- How to use operators in problem solving
- How to write readable expressions

---

1. Why Operators Matter

Imagine building an expense tracker.

A user enters:

```
Income = 100000
Rent = 30000
Food = 15000
Transport = 8000
```

Your program needs to calculate:

```
Remaining money = Income - Rent - Food - Transport
```

Python needs a way to perform that calculation.

That is where **operators** come in.

Operators allow Python programs to:

- calculate values
- compare values
- assign values
- make decisions
- combine conditions
- check membership
- test object identity
- manipulate binary data

Without operators, most Python programs would only be able to store and display information.

With operators, programs can **process information**.

---

2. What Is an Operator?

An **operator** is a symbol or keyword that tells Python to perform an operation.

Example:

```
10 + 5
```

Here:

```
+  → operator
10 → operand
5  → operand
```

The operator tells Python:

> Add these two values.

Result:

```
15
```

Common operators

```
+
-
*
/
%
**
//
```

```
==
!=
>
<
>=
<=
and
or
not
in
is
```

Each operator has a specific purpose.

---

3. What Is an Operand?

The values that an operator works with are called **operands**.

Example:

```
20 - 7
```

There are three parts:

```
20 → operand
-  → operator
7  → operand
```

Another example:

```
price * quantity
```

Here:

```
price     → operand
*         → operator
quantity  → operand
```

Operands can be:

- numbers
- strings
- variables
- function results
- expressions

- objects

---

4. What Is an Expression?

An **expression** is a piece of Python code that produces a value.

Example:

```
10 + 5
```

The result is:

```
15
```

Therefore, it is an expression.

Another example:

```
price * quantity
```

If:

```
price = 500
quantity = 3
```

then:

```
price * quantity
```

produces:

```
1500
```

That makes it an expression.

---

5. Operator vs Expression

These concepts are related but different.

Operator

An operator performs an operation.

```
+
```

Expression

An expression combines values and operators to produce a result.

```
10 + 5
```

Think of it like this:

```
Operand + Operator + Operand
        ↓
     Expression
        ↓
       Value
```

---

6. Arithmetic Operators

Arithmetic operators are used for mathematical calculations.

Python provides:

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus
`**`	Exponentiation
`//`	Floor division

---

7. Addition `+`

The `+` operator adds values.

```
a = 10
b = 5

result = a + b
```

```
print(result)
```

Output:

```
15
```

It also works with strings:

```
first_name = "Hafsa"
last_name = "Asim"

full_name = first_name + " " + last_name

print(full_name)
```

Output:

```
Hafsa Asim
```

So `+` can mean different things depending on the data.

With numbers:

```
10 + 5
```

means addition.

With strings:

```
"Hello" + " Python"
```

means string concatenation.

---

8. Subtraction `-`

The `-` operator subtracts one value from another.

```
balance = 10000
expense = 2500

remaining = balance - expense

print(remaining)
```

Output:

```
7500
```

Real-world example:

```
stock = 100
sold = 35

remaining_stock = stock - sold
```

Now:

```
remaining_stock = 65
```

---

9. Multiplication `*`

The `*` operator multiplies values.

```
price = 500
quantity = 4

total = price * quantity

print(total)
```

Output:

```
2000
```

A very common programming pattern is:

```
total = price * quantity
```

This appears in:

- shopping carts
- invoices
- billing systems
- salary calculations
- order systems
- subscriptions

---

10. Division `/`

The `/` operator performs division.

```
total = 100
people = 4
```

```
share = total / people

print(share)
```

Output:

```
25.0
```

Notice something important:

Even though:

```
100 / 4 = 25
```

Python returns:

```
25.0
```

The `/` operator produces a floating-point result.

---

11. Modulus `%`

The `%` operator returns the **remainder** after division.

Example:

```
result = 10 % 3

print(result)
```

Output:

```
1
```

Because:

```
10 ÷ 3 = 3 remainder 1
```

Why is modulus useful?

It is extremely useful for programming problems.

For example:

```
number = 10

if number % 2 == 0:
    print("Even")
```

The remainder of an even number divided by 2 is:

```
0
```

Therefore, modulus can help determine whether a number is even or odd.

---

12. Modulus in Real Applications

Suppose you are creating a system that gives every second customer a discount.

You could use:

```
customer_number = 6

if customer_number % 2 == 0:
    print("Discount available")
```

Output:

```
Discount available
```

Modulus is commonly useful for:

- even/odd checking
- alternating actions
- repeating patterns
- pagination
- cycles
- scheduling logic
- indexing patterns

---

13. Exponentiation `**`

The `**` operator calculates powers.

```
result = 2 ** 3

print(result)
```

Output:

```
8
```

Because:

```
2 × 2 × 2 = 8
```

Another example:

```
square = 5 ** 2
cube = 5 ** 3
```

Results:

```
square = 25
cube = 125
```

---

14. Floor Division `//`

Floor division divides two numbers and returns the floor of the result.

```
result = 10 // 3

print(result)
```

Output:

```
3
```

Normal division:

```
10 / 3
```

returns approximately:

```
3.3333333333333335
```

Floor division:

```
10 // 3
```

returns:

```
3
```

---

15. Floor Division with Negative Numbers

Be careful with negative numbers.

```
print(-10 // 3)
```

Output:

```
-4
```

Python's floor division rounds toward negative infinity, not simply toward zero.

That distinction matters when working with negative values.

---

16. Arithmetic Operator Summary

```
a = 10
b = 3

print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.333...
print(a % b)   # 1
print(a ** b)  # 1000
print(a // b)  # 3
```

---

17. Assignment Operator

The `=` operator assigns a value to a variable.

```
name = "Hafsa"
```

This does **not** mean:

> name equals Hafsa mathematically.

It means:

> Store the value `"Hafsa"` in the variable `name`.

Example:

```
age = 25
```

Python evaluates the right side and assigns the result to `age`.

---

18. Assignment Is Not Comparison

This is one of the most important distinctions in Python.

Assignment:

```
age = 25
```

Comparison:

```
age == 25
```

The first means:

```
Store 25 in age
```

The second means:

```
Is age equal to 25?
```

It produces:

```
True
```

---

19. Compound Assignment Operators

Python provides shorthand assignment operators.

Instead of:

```
score = score + 10
```

you can write:

```
score += 10
```

Both mean the same general operation.

Common compound assignment operators:

Operator	Example	Equivalent
`+=`	`x += 5`	`x = x + 5`
`-=`	`x -= 5`	`x = x - 5`
`*=`	`x *= 5`	`x = x * 5`
`/=`	`x /= 5`	`x = x / 5`
`//=`	`x //= 5`	`x = x // 5`

`%=`	`x %= 5`	`x = x % 5`
`**=`	`x **= 5`	`x = x ** 5`

---

20. Practical Assignment Example

Suppose a user's account starts with:

```
balance = 5000
```

The user receives:

```
balance += 2000
```

Now:

```
7000
```

The user spends:

```
balance -= 1500
```

Now:

```
5500
```

This makes code easier to read.

---

21. Comparison Operators

Comparison operators compare values.

They produce a Boolean result:

```
True
```

or:

```
False
```

Python comparison operators include:

```
==
!=
>
```

```
<
>=
<=
```

---

22. Equal To `==`

The `==` operator checks whether two values are equal.

```
age = 25

print(age == 25)
```

Output:

```
True
```

Another example:

```
print(10 == 5)
```

Output:

```
False
```

---

23. Not Equal `!=`

The `!=` operator checks whether values are different.

```
password = "python123"

print(password != "hello")
```

Output:

```
True
```

---

24. Greater Than `>`

```
age = 25

print(age > 18)
```

Output:

```
True
```

This can be used for eligibility logic:

```
age = 20

if age > 18:
    print("Eligible")
```

---

25. Less Than `<`

```
stock = 3

if stock < 5:
    print("Low stock")
```

Output:

```
Low stock
```

---

26. Greater Than or Equal `>=`

```
score = 70

if score >= 50:
    print("Pass")
```

The condition is true because:

```
70 >= 50
```

---

27. Less Than or Equal `<=`

```
age = 17

if age <= 18:
    print("Within age limit")
```

---

28. Comparison Operator Table

Operator	Meaning	Example
`==`	Equal	`5 == 5`

`!=`	Not equal	`5 != 3`
`>`	Greater than	`5 > 3`
`<`	Less than	`3 < 5`
`>=`	Greater than or equal	`5 >= 5`
`<=`	Less than or equal	`3 <= 5`

---

29. Logical Operators

Logical operators combine or modify conditions.

Python has:

```
and
or
not
```

These become especially important when you start writing decision-making programs.

---

30. `and`

`and` requires both conditions to be true.

```
age = 25
has_id = True

if age >= 18 and has_id:
    print("Access granted")
```

Both conditions must be true.

Think:

```
Condition A AND Condition B
```

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

---

31. `or`

`or` requires at least one condition to be true.

```
is_admin = False
is_manager = True

if is_admin or is_manager:
    print("Can access dashboard")
```

Because one condition is true, the complete expression is true.

Truth table:

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

---

32. `not`

`not` reverses a Boolean value.

```
logged_in = False

if not logged_in:
    print("Please log in")
```

`not False` becomes:

```
True
```

Another example:

```
is_blocked = False

if not is_blocked:
    print("User can continue")
```

---

33. Combining Logical Operators

You can combine several conditions.

```
age = 25
country = "Pakistan"
has_account = True

if age >= 18 and country == "Pakistan" and has_account:
    print("Eligible")
```

This expression contains:

- comparison operators
- logical operators

That is one reason expressions become powerful.

---

34. Membership Operators

Membership operators check whether a value exists inside a collection.

Python provides:

```
in
not in
```

Example:

```
skills = ["Python", "JavaScript", "Flutter"]

print("Python" in skills)
```

Output:

```
True
```

---

35. `in`

`in` checks membership.

```
languages = ["Python", "JavaScript"]

if "Python" in languages:
    print("Python is available")
```

Output:

```
Python is available
```

It also works with strings:

```
text = "Python programming"

print("Python" in text)
```

Output:

```
True
```

---

36. `not in`

`not in` checks that something does not exist in a collection.

```
skills = ["Python", "JavaScript"]

if "Java" not in skills:
    print("Java is not listed")
```

---

37. Identity Operators

Python provides:

```
is
is not
```

These check **object identity**, not simply value equality.

Example:

```
a = None

if a is None:
    print("No value")
```

This is a common and recommended use of `is`.

---

38. `==` vs `is`

This distinction is extremely important.

`==` asks:

> Do these objects have equal values?

`is` asks:

> Are these the exact same object?

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)
print(a is b)
```

The first can be:

```
True
```

The second:

```
False
```

because `a` and `b` are separate list objects containing the same values.

Rule

Use:

```
==
```

for value comparison.

Use:

```
is
```

for identity checks.

Especially:

```
value is None
```

---

39. Bitwise Operators

Bitwise operators work with the binary representation of integers.

They include:

```
&
|
^
~
<<
>>
```

These are more advanced than normal arithmetic and logical operators.

Example:

```
a = 5
b = 3

print(a & b)
```

Bitwise operators are useful in areas such as:

- low-level programming
- permissions
- flags
- embedded systems
- networking
- performance-sensitive code
- binary data processing

---

40. Understanding Binary Representation

The number:

```
5
```

in binary is:

```
101
```

The number:

```
3
```

is:

```
011
```

A bitwise AND compares corresponding bits.

```
101
011
---
001
```

So:

```
5 & 3
```

produces:

```
1
```

You do not need to memorize bitwise operations immediately. Understanding their purpose is more important at this stage.

---

41. Operator Precedence

What happens when an expression contains several operators?

Consider:

```
result = 2 + 3 * 4
```

Will Python calculate:

```
(2 + 3) * 4
```

or:

```
2 + (3 * 4)
```

Python follows **operator precedence**.

Multiplication has higher precedence than addition.

Therefore:

```
2 + 3 * 4
```

becomes:

```
2 + 12
```

Result:

```
14
```

---

42. Parentheses Control the Order

If you want addition first:

```
result = (2 + 3) * 4
```

Now:

```
5 * 4
```

Result:

```
20
```

Parentheses make your intention explicit.

---

43. Important Precedence Order

A simplified precedence order is:

```
()
**
+x, -x, ~x
*, /, //, %
+, -
<, <=, >, >=, ==, !=
not
and
or
```

Assignment operators such as:

```
=
+=
-=
```

have their own lower-precedence behavior.

When an expression becomes complicated, use parentheses instead of relying on memory.

---

44. Associativity

When operators have the same precedence, Python also follows associativity rules.

For many arithmetic operators, evaluation happens from left to right.

Example:

```
20 - 5 - 3
```

Python evaluates it as:

```
(20 - 5) - 3
```

Result:

```
12
```

Exponentiation behaves differently:

```
2 ** 3 ** 2
```

is interpreted as:

```
2 ** (3 ** 2)
```

which gives:

```
512
```

---

45. Expressions Can Be Nested

Expressions can contain other expressions.

```
total = (price * quantity) + shipping
```

Here:

```
price * quantity
```

is one expression.

The complete:

```
(price * quantity) + shipping
```

is another larger expression.

This allows complex calculations to be built from smaller pieces.

---

46. Expressions With Variables

Expressions do not need literal numbers.

```
price = 1500
discount = 200

final_price = price - discount
```

The expression:

```
price - discount
```

uses variable values.

---

47. Expressions With Function Results

A function call can also be part of an expression.

```
name = input("Enter your name: ")

message = "Hello, " + name
print(message)
```

The result of:

```
input(...)
```

becomes part of the program's data flow.

---

48. haas.dev Example: Resource Access

Imagine haas.dev has a system that counts how many free resources a learner can access.

```
total_resources = 100
completed_resources = 35

remaining_resources = total_resources - completed_resources

print(f"Resources remaining: {remaining_resources}")
```

The expression:

```
total_resources - completed_resources
```

transforms stored information into useful information.

This is the basic role of expressions in real software:

```
Input data
    ↓
```

```
Expression
    ↓
Processed result
    ↓
Useful output
```

---

49. Web Application Example

Imagine an e-commerce application.

```
price = 2500
quantity = 2
shipping = 300

subtotal = price * quantity
total = subtotal + shipping

print(total)
```

The calculations are:

```
2500 × 2 = 5000
5000 + 300 = 5300
```

Output:

```
5300
```

A real application could add:

```
if total >= 5000:
    shipping = 0
```

Now operators are being used for both:

- calculation
- decision making

---

50. Discount Calculation

Suppose a store gives a 10% discount.

```
price = 5000
discount_rate = 0.10

discount = price * discount_rate
```

```
final_price = price - discount

print(final_price)
```

Result:

```
4500.0
```

The expressions are:

```
price * discount_rate
```

and:

```
price - discount
```

---

51. Combining Comparison and Arithmetic

You can calculate something and compare the result.

```
price = 1200
quantity = 5

total = price * quantity

if total >= 5000:
    print("Discount available")
```

The expression:

```
price * quantity
```

is evaluated first.

Then:

```
total >= 5000
```

is evaluated.

---

52. Boolean Expressions

An expression does not always produce a number.

It can produce:

```
True
```

or:

```
False
```

Example:

```
age = 25

is_adult = age >= 18
```

Now:

```
is_adult
```

contains:

```
True
```

Boolean expressions are fundamental to:

- authentication
- authorization
- validation
- filtering
- search
- business rules
- application state

---

53. Chained Comparisons

Python allows chained comparisons.

Instead of:

```
age >= 18 and age <= 60
```

you can write:

```
18 <= age <= 60
```

Example:

```
age = 25

if 18 <= age <= 60:
    print("Age is within the range")
```

This is readable and idiomatic Python.

---

54. String Comparisons

Comparison operators can also work with strings.

```
print("apple" == "apple")
```

Output:

```
True
```

Python compares strings based on their values.

You can also compare ordering:

```
print("apple" < "banana")
```

This follows lexicographical ordering.

Do not assume string ordering represents natural-language meaning in every situation.

---

55. Comparing Different Data Types

Be careful when comparing unrelated types.

For example:

```
age = input("Enter your age: ")
```

The result is a string.

Therefore:

```
age >= 18
```

is not valid because `age` is text.

Convert it:

```
age = int(input("Enter your age: "))

if age >= 18:
    print("Adult")
```

This connects directly to the previous topic:

> Input → Conversion → Expression → Result

---

56. Operator Overloading

Python operators can behave differently depending on the type of object.

For example:

```
5 + 3
```

means numeric addition.

But:

```
"Hello " + "Python"
```

means string concatenation.

Lists can also use `+`:

```
[1, 2] + [3, 4]
```

Result:

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

Later, when learning OOP, you will see how custom classes can define operator behavior through special methods.

---

57. Common Mistake: `=` Instead of `==`

Incorrect:

```
if age = 18:
    print("Adult")
```

Correct:

```
if age == 18:
    print("Exactly 18")
```

Remember:

```
=   assignment
==  comparison
```

---

58. Common Mistake: Using `is` for Normal Value Comparison

Avoid:

```
if name is "Hafsa":
    ...
```

Use:

```
if name == "Hafsa":
    ...
```

Use `is` for identity checks, such as:

```
if value is None:
    ...
```

---

59. Common Mistake: Forgetting Operator Precedence

This:

```
result = 10 + 5 * 2
```

produces:

```
20
```

not:

```
30
```

If you mean:

```
(10 + 5) × 2
```

write:

```
result = (10 + 5) * 2
```

---

60. Common Mistake: Division by Zero

This causes an error:

```
result = 10 / 0
```

Python raises:

```
ZeroDivisionError
```

If a denominator comes from user input, validate it.

```
number = int(input("Enter denominator: "))

if number != 0:
    result = 100 / number
```

```
    print(result)
else:
    print("Cannot divide by zero")
```

---

61. Common Mistake: Mixing Strings and Numbers

This causes a problem:

```
age = input("Enter age: ")

print(age + 5)
```

Because `age` is a string.

Correct:

```
age = int(input("Enter age: "))

print(age + 5)
```

---

62. Common Mistake: Overly Complicated Expressions

This is difficult to understand:

```
result = ((price * quantity) - (price * quantity * discount)) + shipping -
tax
```

It may work, but readability can suffer.

Break it into meaningful steps:

```
subtotal = price * quantity
discount_amount = subtotal * discount
after_discount = subtotal - discount_amount
total = after_discount + shipping - tax
```

Readable code is easier to:

- debug
- test
- modify
- explain
- maintain

---

63. Expressions and Readability

Good:

```
total_price = price * quantity
```

Less readable:

```
x = p * q
```

The second may be technically correct, but meaningful names communicate intent.

Prefer:

```
remaining_balance = account_balance - withdrawal
```

over:

```
x = a - b
```

---

64. When Should You Use Parentheses?

Use parentheses when they:

- change evaluation order
- clarify intent
- make a business rule easier to understand

Example:

```
eligible = (age >= 18) and (has_account)
```

The parentheses are not always necessary, but they can make complex conditions easier to scan.

---

65. A Practical Calculation System

Let's build a small billing calculation.

```
price = 1200
quantity = 3
discount_rate = 0.10
shipping = 250

subtotal = price * quantity
discount = subtotal * discount_rate
after_discount = subtotal - discount
final_total = after_discount + shipping

print(f"Subtotal: {subtotal}")
```

```
print(f"Discount: {discount}")
print(f"Final total: {final_total}")
```

The program follows:

```
Price + Quantity
       ↓
   Subtotal
       ↓
   Discount
       ↓
After Discount
       ↓
   Shipping
       ↓
  Final Total
```

This is how real software often handles calculations: **small expressions connected into a meaningful data flow**.

---

66. Mini Project: Student Result Calculator

Create a program that calculates a student's total and average.

```
math = 80
english = 75
python = 90

total = math + english + python
average = total / 3

print(f"Total: {total}")
print(f"Average: {average}")
```

Now add a pass condition:

```
if average >= 50:
    print("Pass")
else:
    print("Fail")
```

You are now using:

- arithmetic operators
- assignment

- comparison
- expressions
- conditional logic

---

67. Mini Project: Shopping Bill

```
price = 1500
quantity = 2
shipping = 200

subtotal = price * quantity

if subtotal >= 3000:
    shipping = 0

total = subtotal + shipping

print(f"Subtotal: {subtotal}")
print(f"Shipping: {shipping}")
print(f"Total: {total}")
```

This small program already resembles a real business rule.

---

68. Mini Project: Login Eligibility

```
username = "Hafsa"
password_correct = True
account_active = True

if username == "Hafsa" and password_correct and account_active:
    print("Login successful")
else:
    print("Login failed")
```

This demonstrates how operators combine to represent application rules.

---

69. Five Minute Challenge

Create a Python program for an employee's salary.

Given:

```
salary = 80000
bonus = 10000
tax_rate = 0.05
```

Calculate:

1. Salary before tax
2. Tax amount
3. Final salary

Expected structure:

```
salary + bonus
       ↓
gross salary
       ↓
tax calculation
       ↓
final salary
```

Then print all three values.

---

70. Practice Exercises

Easy

Exercise 1

Calculate:

```
25 + 15
```

Exercise 2

Calculate:

```
100 - 45
```

Exercise 3

Calculate:

```
12 * 4
```

Exercise 4

Calculate:

```
50 / 5
```

Exercise 5

Find the remainder:

```
17 % 5
```

---

71. Medium Exercises

Exercise 6

Create variables:

```
price = 750
quantity = 4
```

Calculate the total price.

Exercise 7

Check whether a number is even.

Hint:

```
number % 2 == 0
```

Exercise 8

Check whether a student passed:

```
score >= 50
```

Exercise 9

Check whether a user is eligible when:

```
age >= 18
AND
has_account == True
```

Exercise 10

Check whether `"Python"` exists inside:

```
skills = ["HTML", "CSS", "Python", "JavaScript"]
```

---

72. Hard Exercises

Exercise 11: Discount System

Create a program where:

```
subtotal >= 5000
```

gives a 10% discount.

Otherwise, no discount is applied.

Exercise 12: Delivery System

Create rules:

```
Order >= 3000 → Free delivery
Order < 3000 → Delivery fee 250
```

Calculate the final amount.

Exercise 13: Login Rule

Allow login only when:

```
username is correct
AND
password is correct
AND
account is active
```

Exercise 14: Range Validation

Check whether a number is between 10 and 100.

Use a chained comparison.

Exercise 15: Expense Tracker

Given:

```
income = 100000
rent = 30000
food = 15000
transport = 8000
```

Calculate:

```
total expenses
remaining balance
```

Then check whether the remaining balance is greater than 40,000.

---

73. Interview Questions

What is an operator?

An operator is a symbol or keyword used to perform an operation on one or more operands.

What is an expression?

An expression is Python code that evaluates to a value.

Example:

```
price * quantity
```

What is the difference between `=` and `==`?

`=` assigns a value.

```
age = 25
```

`==` compares values.

```
age == 25
```

What does `%` do?

It returns the remainder of a division.

```
10 % 3
```

returns:

```
1
```

What is the difference between `/` and `//`?

`/` performs regular division and returns a floating-point result.

```
10 / 3
```

`//` performs floor division.

```
10 // 3
```

What does `**` do?

It performs exponentiation.

```
2 ** 3
```

returns:

```
8
```

What do comparison operators return?

They return Boolean values:

```
True
```

or:

```
False
```

What is the difference between `and` and `or`?

`and` requires all connected conditions to be true.

`or` requires at least one connected condition to be true.

What is the difference between `==` and `is`?

`==` compares values.

`is` checks whether two references point to the same object.

What are membership operators?

Python's membership operators are:

```
in
not in
```

They check whether a value exists within a collection.

Why is operator precedence important?

It determines the order in which Python evaluates operators in an expression.

For example:

```
2 + 3 * 4
```

evaluates multiplication before addition.

---

74. Operator Cheat Sheet

Arithmetic

```
+    # addition
-    # subtraction
*    # multiplication
/    # division
%    # remainder
**   # exponentiation
//   # floor division
```

Assignment

```
=
+=
-=
*=
/=
%=
**=
//=
```

Comparison

```
==
!=
>
<
>=
<=
```

Logical

```
and
or
not
```

Membership

```
in
not in
```

Identity

```
is
is not
```

Bitwise

```
&
|
^
~
<<
>>
```

---

75. The Mental Model

When you see a Python expression, think:

```
VALUES
  ↓
OPERATORS
  ↓
EXPRESSION
  ↓
EVALUATION
  ↓
RESULT
```

For example:

```
price * quantity + shipping
```

Python processes the expression according to operator rules and produces a value.

That value can then be:

- stored in a variable
- printed
- compared
- passed to a function
- used in another expression
- used to make a decision

---

76. The Bigger Programming Picture

Operators are not isolated syntax.

They are the building blocks behind programming logic.

A real application often follows this pattern:

```
User Input
    ↓
Data
    ↓
Expressions
    ↓
Calculations / Comparisons
    ↓
Business Rules
    ↓
Decision
    ↓
Output
```

For example, an online store might do:

```
subtotal = price * quantity
```

then:

```
if subtotal >= 5000:
```

then:

```
discount = subtotal * 0.10
```

then:

```
final_total = subtotal - discount
```

A few operators can therefore represent real business logic.

---

77. Key Takeaways

- Operators tell Python what operation to perform.
- Operands are the values operators work with.
- Expressions produce values.
- Arithmetic operators perform calculations.
- Assignment operators store and update values.
- Comparison operators produce Boolean results.

- Logical operators combine conditions.
- Membership operators check whether values exist in collections.
- Identity operators check object identity.
- Bitwise operators work with binary representations.
- Operator precedence determines evaluation order.
- Parentheses can make expressions clearer.
- `=` and `==` have completely different purposes.
- `==` compares values while `is` checks identity.
- `%` is especially useful for remainder and even/odd logic.
- Expressions are the foundation of calculations and business rules.
- Readable expressions are easier to maintain than unnecessarily complex ones.

Final Mental Model

Do not think of operators as symbols you simply need to memorize.

Think of them as **tools for transforming information**.

```
Input
  ↓
Values
  ↓
Operators
  ↓
Expressions
  ↓
Calculated / Compared Result
  ↓
Program Decision
  ↓
Output
```

Once you understand this flow, Python code becomes much easier to read because you can look at an expression and ask:

> What values are being used, what operation is happening, and what result will Python produce?

That question is one of the foundations of programming problem solving.

---

haas.dev

https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.