Python Fundamentals: Operators

A Deep Dive into Python's Core and Advanced Operators

Bîcu Andrei Ovidiu

Table of Contents

Bîcu Andrei Ovidiu

What Are Operators in Python?

Operators are special symbols or keywords in Python used to perform operations on variables and values. They are the core of how Python evaluates expressions, compares values, manipulates data, and performs logic.

- Operators act on operands (e.g., numbers, strings, lists).
- Python includes arithmetic, logical, comparison, assignment, bitwise, and other operators.
- They make code expressive and concise

```
a = 10
b = 5

print(a + b) # Arithmetic: Addition
print(a > b) # Comparison: Greater than
print(a != b) # Comparison: Not equal
print(a and b) # Logical: AND
```

Categories of Operators in Python

Python operators are grouped into categories based on the type of operation they perform. Each category serves a distinct purpose – from basic math to comparing values, assigning data, or manipulating bits.

Category	Purpose	Examples
Arithmetic	Perform basic math operations	+, -, *, /, //, %, **
Comparison	Compare values	==, !=, <, >, <=, >=
Assignment	Assign and modify values	=, +=, -=, *=, /=, etc.
Logical	Combine boolean expressions	and, or, not
Bitwise	Operate on binary representations	&, `
Identity	Check object identity	is, is not
Membership	Test for presence in a sequence	in, not in

Bîcu Andrei Ovidiu

Arithmetic Operators

Arithmetic operators in Python perform basic mathematical operations like addition, subtraction, multiplication, and division. These are among the most frequently used operators in everyday programming. (see picture)

- Work with both integers and floats
- Follows standard mathematical precedence (PEMDAS)
- Can be combined in expressions for complex calculations

```
a = 10
b = 3


print(a + b) # Additions - Adds two values
print(a - b) # Substraction - Substracts second from first
print(a * b) # Multiplication - Multiplies two values
print(a / b) # Division - Divides and returns float
print(a // b) # Floor Division - Division rounded down
print(a % b) # Modulus - Remainder of division
print(a ** b) # Exponentiation - Raises to the power
```

Bîcu Andrei Ovidiu

Special Arithmetic Behavior

Python arithmetic follows some specific behaviors worth noting, especially regarding division and expression evaluation order. Misunderstanding these can lead to unexpected results.

- / returns a float even with integer operands, unlike some other languages
- // performs floor division, discarding the decimal part
- Operator **precedence** determines the order of operations in complex expressions

```
print(10 / 3) # 3.333.. (float division)
print(10 // 3) # 3 (float division)
print(2 + 3 * 4) # 14 (* has higher precedence)
print((2 + 3) * 4) # 20 (parentheses override precedence)
```

Operator Precedence Order (Top -> Bottom)

1. ** (Exponentiation)
2. *, /, //, %
3. +, -

Use parentheses () to make the order explicit and improve readability.

Comparison Operators

Comparison operators are used to compare two values. The result of a comparison is always a boolean value: either True or False. They are essential in conditions, loops, and logical decisions.

- Comparison operators return True or False
- Work with numbers, strings, and other comparable types
- Frequently used in if statements, filters, and loops

```
a = 5
b = 8

print(a == b) # False - Equal to
print(a != b) # True - NOT equal to
print(a < b) # True - Less than
print(a <= 5) # True - Less than or equal
print(b > a) # True - Greater than
print(b >= 10) # False - Greater than or equal
```

Bîcu Andrei Ovidiu

Comparison Chaining and Python Idioms

*Python allows **chaining comparisons**; a unique and elegant feature. You can write expressions like 3 < x < 10, which are evaluated as if connected by logical and. This keeps code cleaner and more readable compared to many other languages.*

- Chained comparisons evaluate all conditions as a single logical expression.
- Internally, 3 < x < 10 is interpreted as 3 < x and x < 10.
- Useful in filtering, bounds checking, and concise conditions.

```
x = 7

# Chained comparison
print(3 < x < 10) # True

# Equivalent expression without chaining
print(3 < x and x < 10 ) # True

# Another example
a = 5
print(1 < a <= 5) # True
```

Assignment Operators

*Assignment operators are used to assign values to variables. Python also supports **compound assignment**, where an operation and assignment happen in one step, like += or *=. These operators help reduce redundancy in your code.*

- = assigns a value to a variable.
- Compound operators like +=, -=, etc., perform an operation and update the variable in one step.
- Works with numbers, strings, lists, and more (depending on the operator).

```
x = 10 # '=' simple assignment

x += 5 # Add and assign / x = x + 5 -> 15
x *= 2 # Multiply and assign / x = x * 2 -> 30
x -= 10 # Substract and assign / x = x - 10 -> 20
x //= 3 # Floor divide and assign / x = x // 3 -> 6
x **= 2 # Power and assign / x = x ** 2 -> 36
x %= 3 # Modulo and assign / x = x // 3 -> 0

print(x)
```

Use Cases, Chaining, and Cautionary Tips

*Assignment in Python supports **chaining** multiple variables in a single statement. However, when dealing with **mutable objects** (like lists or dictionaries), be careful because assignment doesn't copy objects but binds names to the same reference.*

- Assignment chaining allows multiple variables to get the same value.
- For immutable types (e.g., ints, strings), this is straightforward.
- For mutable types (e.g., lists), modifying one reference affects all variables assigned to it.
- Use .copy() or deepcopy() to avoid unintended shared references.

```
# Assignment chaining
a = b = c = 5
print(a, b, c) # 5 5 5

# Mutable objects caution
list1 = [1, 2, 3]
list2 = list1 # Both names point to same list
list2.append(4)

print(list1, list2) # [1, 2, 3, 4] [1, 2, 3, 4]

# Correct way: copy the list
import copy

list3 = copy.copy(list1)
list3.append(5)

print(list1, list3) # [1, 2, 3, 4] [1, 2, 3, 4 ,5]
```

Logical Operators

Logical operators allow you to combine or invert boolean expressions. They are crucial for controlling flow and making decisions in Python programs.

- Operate on boolean values or expressions that evaluate True or False.
- Python's logical operators: and, or, not.
- Support **short-circuit evaluation** to optimize performance.

```python
a = True
b = False

print(a and b) #  AND - True if both operants are True -- False
print(a or b) # OR - True if at least one operand is True -- True
print( not a) # NOT - Inverts the truth value -- False

# Combining expressions
x = 5
print(x > 0 and x < 10) # True
print(not (x == 5)) # False
```

Bîcu Andrei Ovidiu

Short-Circuit Behavior

Short-circuiting *is an evaluation strategy used in logical expressions. Python stops evaluating as soon as the result is determined; which can improve performance and prevent errors in some cases.*

- and stops if the **first operand is False**
- or stops if the **first operand is True**
- This behavior is useful for conditions where the second part depends on the first.

```
def check():
    print("Function called")
    return True

# AND: stops early
print(False and check()) # Function not called -> False

# OR: stops early
print(True or check()) # Function not called -> True

# Full evaluation
print(True and check()) # Function called -> True
```

Bitwise Operators

*Bitwise operators perform operations on the **binary representation** of integers. They are useful in low-level programming, graphics, permissions, and optimization tasks.*

- Operate at the **bit level**, not the value level
- Only work with **integers**
- Can be used for **flags**, **masks**, and **efficient toggling**

```
a = 5 # 0b0101
b = 3 # 0b0011

print(a & b) # Bits that are 1 in both operands -> # Out: 1 -> 0b0001
print(a | b) # OR # Out: 7 -> 0b0111
print(a ^ b) # XOR, bits that are 1 in one but not both  # Out: 6 -> 0b0110
print(~a) # NOT, Inverts all bits (1's complemented) # Out: -6 (inverted bits)
print(a << 1) # Left shift, shifts bits left (multiply by 2^n) # Out: 10 -> 0b1010
print(a >> 1) # Right shift, shifts bits right (divide by 2^n) # Out: 2 -> 0b0010
```

Bîcu Andrei Ovidiu

Practical Bitwise Use Cases

Bitwise operations aren't just theoretical, they're used in real-world applications such as managing flags, performing fast math, and working with binary data like file headers or hardware interfaces.

- Useful in **feature toggles**, **access control**, **compression**, and **game development**
- Fast and memory-efficient compared to traditional alternatives
- Requires care when combining with user-facing logic

Common Use Cases:

- **Flags & Permissions**: Enable/disable features using individual bits
- **Masking**: Isolate or modify specific bits
- **Efficient Multiplication/Division**: Using << and >>

```
# Example: Feature flags using bits

READ = 0b0001 # 1
WRITE = 0b0010 # 2
EXEC = 0b0100 # 4

user_permissions = READ | WRITE # 0b0011

# Check if WRITE is enabled
print(user_permissions & WRITE != 0) # True

# Add EXEC permission
user_permissions |= EXEC

# Remove READ permission
user_permissions &= ~READ

print(bin(user_permissions)) # 0b110 (WRITE + EXEC)
```

Membership and Identity Operators

*Python includes operators to test **membership** (whether a value is in a container) and **identity** (whether two references point to the same object in memory). These are especially useful when working with collection and object comparisons.*

- **Membership operators:** in, not in – check if a value exists in a sequence or collection.
- **Identity operators**: is, is not – check if two references point to the **same object**, not just equal values.
- is is not the same as == – the former checks identity, the latter checks equality.

```
# Membership
fruits = ['apple', 'banana', 'cherry']
print("banana" in fruits) # True
print("mango" not in fruits) # True

# Identity
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y) # True (same object)
print(x == z) # True (equal values)
print(x is z) # False (different objects)
```

Operator Precedence and Associativity

*When multiple operators appear in a single expression, Python follows a set of **precedence rules** to determine the evaluation order. Operators also have **associativity**, which tells Python whether to evaluate from left to right or right to left when precedence is equal.*

Quick Tip:

When in doubt, **use parentheses** to make the evaluation order explicit and improve code clarity.

Precedence Level	Operators	Associativity
1 (Highest)	() (parentheses)	N/A
2	**	Right-to-left
3	+, - (unary), ~	Right-to-left
4	*, /, //, %	Left-to-right
5	+, - (binary)	Left-to-right
6	<<, >>	Left-to-right
7	&	Left-to-right
8	^	Left-to-right
9	`	`
10	==, !=, <, >, <=, >=, is, is not, in, not in	Left-to-right
11	not	Right-to-left
12	and	Left-to-right
13	or	Left-to-right
14 (Lowest)	=, +=, -=, etc.	Right-to-left

Custom Operators via Magic Methods

*Python allows you to **override operator behavior** for custom classes using special methods called **magic methods** or **dunder methods** (because they begin and end with double underscores). This lets you define how your objects respond to operators like +, ==, or <.*

- Magic methods are Python's way of supporting operator overloading.
- Common one include: __add__, __eq__, __lt__, __mul__, etc.
- Useful when building classes that behave like built-in types (e.g., vectors, money, units).

```python
class Point:
    def __init__(self, x, y):
        # Initialize a point with x and y coordinates
        self.x = x
        self.y = y

    def __add__(self, other):
        # Define how two Point objects are added using the '+' operator
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        # Define how the object is displayed when printed
        return f"({self.x}, {self.y})"

# Create two Point instances
p1 = Point(1, 2)
p2 = Point(3, 4)

# Use the overloaded '+' operator to add the two points
result = p1 + p2

# Print the res
print(result)  # Output: (4, 6)
```

Example: Custom Comparison for a Class

*Just like with arithmetic operators, Python lets you define **custom comparison logic** for your classes using methods such as __eq__, __lt__, __le__, and others. This is especially useful when sorting or comparing objects like people, products, or records.*

- __eq__ is used for ==
- __lt__ is used for <
- __gt__ is used for >
- Implementing __lt__ and __eq__ is often enough for sorting
- You can use the functools.total_ordering decorator to auto-generate the rest

```python
class Person:
    def __init__(self, name, age):
        # Initialize a person with name and age
        self.name = name
        self.age = age

    def __eq__(self, other):
        # Define equality: two people are equal if their age is equal
        return self.age == other.age

    def __lt__(self, other):
        # Define less-than: compare people based on age
        return self.age < other.age

    def __str__(self):
        # Define how the object is printed (human-readable string)
        return f"{self.name} ({self.age})"

# Create a list of people
people = [
    Person('Alice', 30),
    Person("Bob", 25),
    Person("Charlie", 35)
]

# Sort the list based on age using the overloaded < operator
people.sort()

# Print the sorted list
for person in people:
    print(person)

'''Expected output:
Bob (25)
Alice (30)
Charlie (35)
'''
```

Common Pitfalls with Operators

1. Using is vs == improperly

- == compares **values**
- is compares **identities** (memory address)
- a == b might be True **even if** a is b is False

2. Mutable default arguments in operator overloads

- Modifying mutable attributes (like lists) in magic methods can lead to side effects if not copied properly.

3. Forgetting parentheses in chained comparisons

- Python allows chained comparisons like 1 < x < 5 but parentheses can be necessary if you're mixing logical and comparison operators

```python
"""1. Using is vs == improperly"""
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b) # True - values are equal
print(a is b) # False - different objects in memory

"""2. Mutable default arguments in operator overloads"""
class Counter:
    def __init__(self, nums=[]):
        self.nums = nums

    def __add__(self, other):
        # Problem: modifies original list in-place!
        self.nums += other.nums
        return self
# Fix: Use .copy() or + to create a new list instead of modifying in-place

"""3. Forgetting parentheses in chained comparisons"""
x = 3
print(1 < x < 5) # True - chained comparison
print((1 < x) and (x < 5)) # Equivalent
```

Bîcu Andrei Ovidiu

Best Practices for Operator Usage

Key Guidelines:

- Use operators to write **clear**, **concise**, and **readable** code.
- Prefer == for **equality checks**, and use is **only for identity checks** (e.g., comparing to None)
- When overloading operators in custom classes, ensure **behavior is intuitive and consistent**.
- Avoid modifying mutable objects in-place inside operator methods; prefer creating new objects.
- Use **parentheses** to make complex expressions explicit and easier to read.
- Leverage **short-circuit evaluation** to write efficient conditional expressions.

```
# Clear equality vs identity check
if value == None:  # AVOID
    pass

if value is None: # PREFERRED
    pass

# Parentheses for clarity
if (x > 0 and x < 10) or y == 5:
    print("In range or y equals 5")
```

Python Operators Cheat Sheet

Operator Category	Operators	Description
Arithmetic	+, -, *, /, //, %, **	Basic math operations
Comparison	==, !=, <, >, <=, >=	Compare values, return boolean
Assignment	=, +=, -=, *=, /=, //=, %=	Assign and modify variables
Logical	and, or, not	Combine boolean expressions
Bitwise	&, `, ^, ~, <<, >>`	Operations to binary representations
Membership	in, not in	Check presence in sequences
Identity	is, is not	Check if two references point to the same object

Glossary

Term	Definition
Operator	A symbol or keyword that performs an operation on one or more operands.
Operand	The values or variables on which an operator acts.
Precedence	The order in which Python evaluates operators in an expression.
Associativity	The direction (left-to-right or right-to-left) in which operators of the same precedence are evaluated.
Short-circuiting	Evaluation strategy where Python stops evaluating a logical expression as soon as the result is determined.
Mutable	Objects that can be changed after creation (e.g., lists, dictionaries).
Immutable	Objects that cannot be changed after creation (e.g., integers, strings, tuples).
Magic Methods	Special methods with double underscores that allow customization of operator behavior (e.g., __add__, __eq__).
Chained Comparison	The ability to combine multiple comparison operators in a single expression (e.g., 1 < x < 10).
Bitwise Operator	Operator that acts on the binary representation of integers.
Membership Operator	Operator used to test if a value is present in a sequence or collection (in, not in).
Identity Operator	Operator used to test if two variables reference the same object (is, is not).

References & Further Reading

Books & Documentation

Python Official Documentation - https://docs.python.org/3/reference/expressions.html#operators

Online Resources

Real Python: https://realpython.com/python-operators-expressions

GeeksforGeeks: https://realpython.com/python-operators-expressions

Programiz: https://www.programiz.com/python-programming/operators

Bîcu Andrei Ovidiu