NOTES

Data Operators

Mutability of Objects: Understanding Mutable vs Immutable Objects in Python

In Python, everything is an object, and objects are categorised as **mutable** or **immutable** based on whether their values can be changed after they are created. This concept is foundational in Python programming, especially for data structures and memory management.

1. What are Mutable and Immutable Objects?

- **Mutable objects**: These are objects whose values **can be changed** after their creation.
- **Immutable objects**: These are objects whose values **cannot be changed** after their creation.

2. Why is Mutability Important?

Understanding mutability helps in making better decisions about data structures and managing memory efficiently. If you need an object that changes over time (like a list of users), use a **mutable object**. If you need an object that remains constant (like a name or a key in a dictionary), use an **immutable object**.

Examples of Mutable Objects:

- **Lists:** Lists are mutable, which means you can modify their contents after creation.

Example:

```
Python
my_list = [1, 2, 3]
my_list[0] = 10  # Modifying the first element
print(my_list)  # Output: [10, 2, 3]
```

Output:

```
Python
[10, 2, 3]
```

- **Dictionaries:** Dictionaries allow changes to their key-value pairs.

Example:

```
Python
my_dict = {"name": "Alice", "age": 25}
my_dict["age"] = 26  # Changing the value of age
print(my_dict)  # Output: {'name': 'Alice', 'age': 26}
```

Output:

```
Python
{'name': 'Alice', 'age': 26}
```

Examples of Immutable Objects

- **Strings:** Strings are immutable, meaning once you create a string, you cannot change its content. Any modification results in a new string.

Example:

```
Python
my_string = "hello"
my_string[0] = "H"  # This will cause an error because strings are
immutable
```

- **Tuples:** Tuples are like lists, but they are immutable. Once created, you cannot modify them.

Example:

```
Python
my_tuple = (1, 2, 3)
my_tuple[0] = 10  # This will raise an error as tuples are immutable
```

3. Key Difference

Mutable Objects	Immutable Objects
Can be changed after creation.	Cannot be changed after creation.
Examples: Lists, Dictionaries.	Examples: Strings, Tuples.
Stored in memory as a reference, meaning changes affect the original object.	Each modification creates a new object in memory.

Example to Illustrate Mutability:

Let's look at an example to see how Python handles mutable and immutable objects differently:

```python
# Immutable object (String)
x = "hello"
y = x
x = "world"
print(x)  # Output: "world"
print(y)  # Output: "hello" (y is not affected)
```

Output:

```python
world
hello
```

```python
# Mutable object (List)
a = [1, 2, 3]
b = a
a[0] = 10
print(a)  # Output: [10, 2, 3]
print(b)  # Output: [10, 2, 3] (b is affected by the change to a)
```

Output:

```python
[10, 2, 3]
[10, 2, 3]
```

In this example:

- **String**: Changing `x` to `"world"` does not affect `y` because strings are immutable.
- **List**: Changing `a` also changes `b` because lists are mutable.

4. Why Use Immutable Objects?

- **Safety**: Immutable objects are safer to use, especially in concurrent programming, as they prevent accidental changes.
- **Efficiency**: Python can optimise memory use with immutable objects, as they do not need to be modified.

5. When to Use Mutable Objects?

- **Flexibility**: If you need to modify a collection of data over time (e.g., appending items to a list), mutable objects like lists and dictionaries are ideal.

6. Conclusion

Understanding the concept of **mutable** vs **immutable** objects in Python is critical for managing data effectively. Whether you're working with lists, dictionaries, strings, or tuples, knowing their mutability will guide you in choosing the right data structures and managing memory efficiently.

Understanding Operators in Python

Operators are essential in Python programming as they allow you to perform operations on variables and values. In this sub-module, we'll cover:

- Arithmetic operators
- Bitwise operators
- Comparison operators
- Assignment operators
- Operator precedence and associativity

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, and division.

Common Arithmetic Operators:

Operator	**Name**
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor Division
%	Modulus(return remainder)
**	Exponentiation(raises power)

Examples:

```
Python
x = 10
y = 3
print(x + y)  # 13
print(x - y)  # 7
print(x * y)  # 30
print(x / y)  # 3.333...
print(x // y) # 3 (floor division)
print(x % y)  # 1 (remainder)
print(x ** y) # 1000 (10 raised to power 3)
```

2. Bitwise Operators

Bitwise operators operate on binary representations of integers. They are used for operations such as shifting bits and comparing individual bits.

Common Bitwise Operators:

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR (exclusive OR)	Sets each bit to 1 if only one of two bits is 1
~	NOT (complement)	Inverts all the bits
<<	Left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off

>>	Right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Example:

```python
a = 5  # 101 in binary
b = 3  # 011 in binary

# Bitwise AND
print(f"a & b: {a & b}")  # Output: 1 (001 in binary)

# Bitwise OR
print(f"a | b: {a | b}")  # Output: 7 (111 in binary)

# Bitwise XOR
print(f"a ^ b: {a ^ b}")  # Output: 6 (110 in binary)

# Bitwise NOT
print(f"~a: {~a}")  # Output: -6 (In binary: NOT(101) -> -(110))

# Left Shift
print(f"a << 1: {a << 1}")  # Output: 10 (In binary: 101 -> 1010)

# Right Shift
print(f"a >> 1: {a >> 1}")  # Output: 2 (In binary: 101 -> 010)
```

Explanation:

Operator	Name	Description	Example	Result
&	AND	Compare each bit of two numbers. The result is 1 if both bits are 1, otherwise 0.	`5 & 3` → `101 & 011`	1(binary: `001`)
|	OR	Compare each bit of two numbers. The result is 1 if at least one of the bits is 1.	`5 | 3` → `101 | 011`	7 (111 in binary)
^	XOR	Compare each bit of two numbers. The result is 1 if the bits are different, otherwise 0.	`5 ^ 3` → `101 ^ 011`	6 (110 in binary)
~	NOT	Flips all the bits. For signed integers, it returns `-(n+1)` (using two's complement representation for negative	~5 → `NOT 101`	-6 (-(110) in binary:)

		numbers)		
<<	Left Shift	Shifts all the bits to the left by a specified number of positions, filling the empty spaces with `0`.	`5 << 1` → `101 << 1`	`10` (binary: `1010`)
>>	Right Shift	Shifts all the bits to the right by a specified number of positions, removing bits that are shifted out, and filling with `0` for unsigned or sign bit for signed integers.	`5 >> 1` → `101 >> 1`	`2` (binary: `010`)

Use Case:

- **AND, OR, XOR**: Often used in low-level programming, networking, and cryptography for tasks such as masking, setting bits, and comparing binary data.
- **NOT**: Used for toggling bits or performing bitwise complements.
- **Left/Right Shifts**: Efficient for multiplying or dividing integers by powers of two.

3. Comparison Operators

Comparison operators are used to compare two values and return a Boolean result: `True` or `False`.

Common Comparison Operators:

- `==` : Equal to
- `!=` : Not equal to
- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

Examples:

```python
x = 10
y = 5
print(x == y)  # False
print(x != y)  # True
print(x > y)   # True
print(x < y)   # False
print(x >= y)  # True
print(x <= y)  # False
```

4. Assignment Operators

Assignment operators are used to assign values to variables. They can also be combined with arithmetic operators to modify and assign a value in one step.

Common Assignment Operators:

- `=` : Simple assignment
- `+=` : Add and assign
- `-=` : Subtract and assign
- `*=` : Multiply and assign
- `/=` : Divide and assign
- `//=` : Floor divide and assign
- `%=` : Modulus and assign
- `**=` : Exponent and assign

Python
```python
# Simple Assignment
x = 10
print(f"Initial value: {x}")  # Output: 10

# Add and Assign (+=)
x += 5  # Equivalent to x = x + 5
print(f"After += 5: {x}")  # Output: 15

# Subtract and Assign (-=)
x -= 3  # Equivalent to x = x - 3
print(f"After -= 3: {x}")  # Output: 12

# Multiply and Assign (*=)
x *= 2  # Equivalent to x = x * 2
```

```
print(f"After *= 2: {x}")  # Output: 24

# Divide and Assign (/=)
x /= 4  # Equivalent to x = x / 4
print(f"After /= 4: {x}")  # Output: 6.0

# Floor Divide and Assign (//=)
x //= 2  # Equivalent to x = x // 2 (integer division)
print(f"After //= 2: {x}")  # Output: 3.0

# Modulus and Assign (%=)
x %= 2  # Equivalent to x = x % 2 (remainder of division)
print(f"After %= 2: {x}")  # Output: 1.0

# Exponent and Assign (**=)
x **= 3  # Equivalent to x = x ** 3 (raise to the power of 3)
print(f"After **= 3: {x}")  # Output: 1.0
```

5. Operator Precedence and Associativity

Operator precedence determines which operator is evaluated first in an expression with multiple operators. Associativity defines the direction in which an operation is performed when operators have the same precedence.

Precedence: Operators with higher precedence are evaluated first. **Associativity**: Determines how operators of the same precedence are grouped (left-to-right or right-to-left).

Operator Precedence (from high to low):

1. ** (Exponentiation)
2. *, /, //, % (Multiplication, division, floor division, modulus)
3. +, - (Addition, subtraction)
4. ==, !=, >, <, >=, <= (Comparison)
5. =, +=, -=, etc. (Assignment)

```python
result = 2 + 3 * 4 ** 2 / 2
# Step-by-step evaluation:
# 4 ** 2 = 16 (Exponentiation first)
# 3 * 16 = 48 (Multiplication)
# 48 / 2 = 24 (Division)
# 2 + 24 = 26 (Addition)
print(result)  # Output: 26.0
```

THANK YOU

AIMERZ.ai

Aim.Act.Achieve