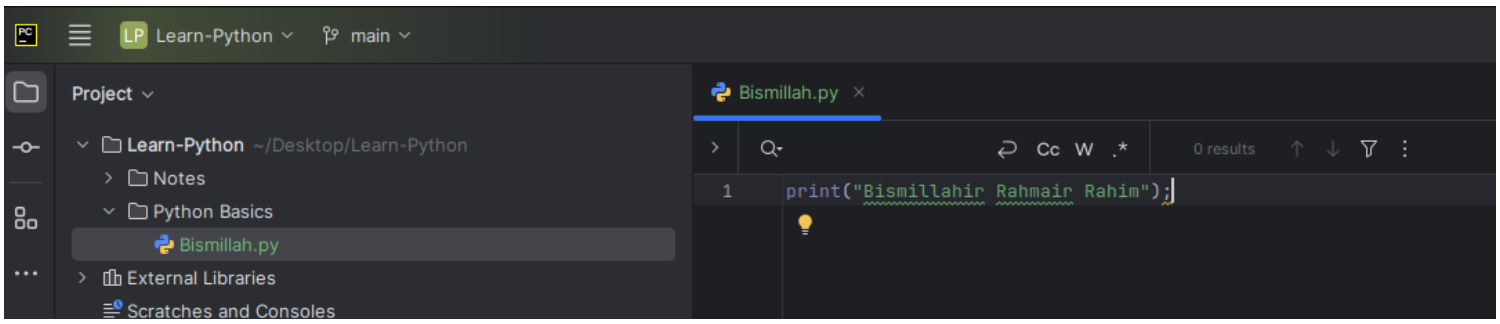
Pycharm Basics

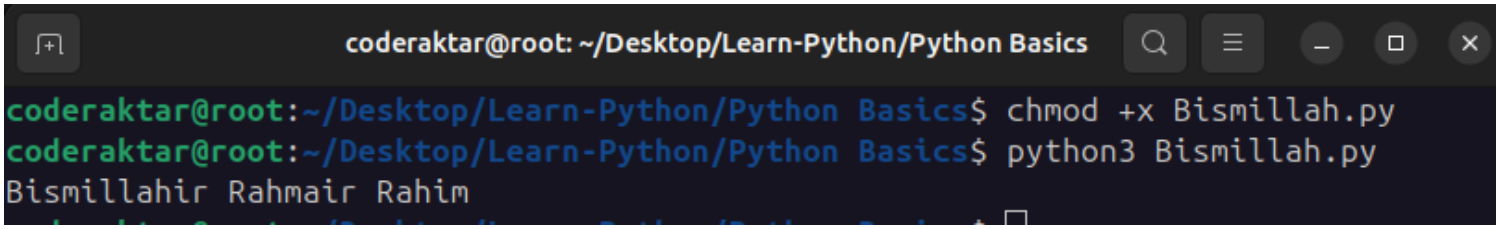
Duplicate Line: Ctrl + D
Search word in Current File: Ctrl + F
Search word in whole Project: Ctrl + shift + F
Search File in project: Shift double pressed

Run Python From Command Line

Create File name Bismillah.py



Go to this Folder where file is located
1. Giving Execution Permission for write: chmod +x Bismillah.py
2. Run Python From Terminal: python3 Bismillah.py



Python Basics

Variable In Python

A **variable** in Python is used to store data that can be accessed and manipulated later in the code. Variables are created by assigning a value to a name using the `=` operator.

Here’s a basic example:

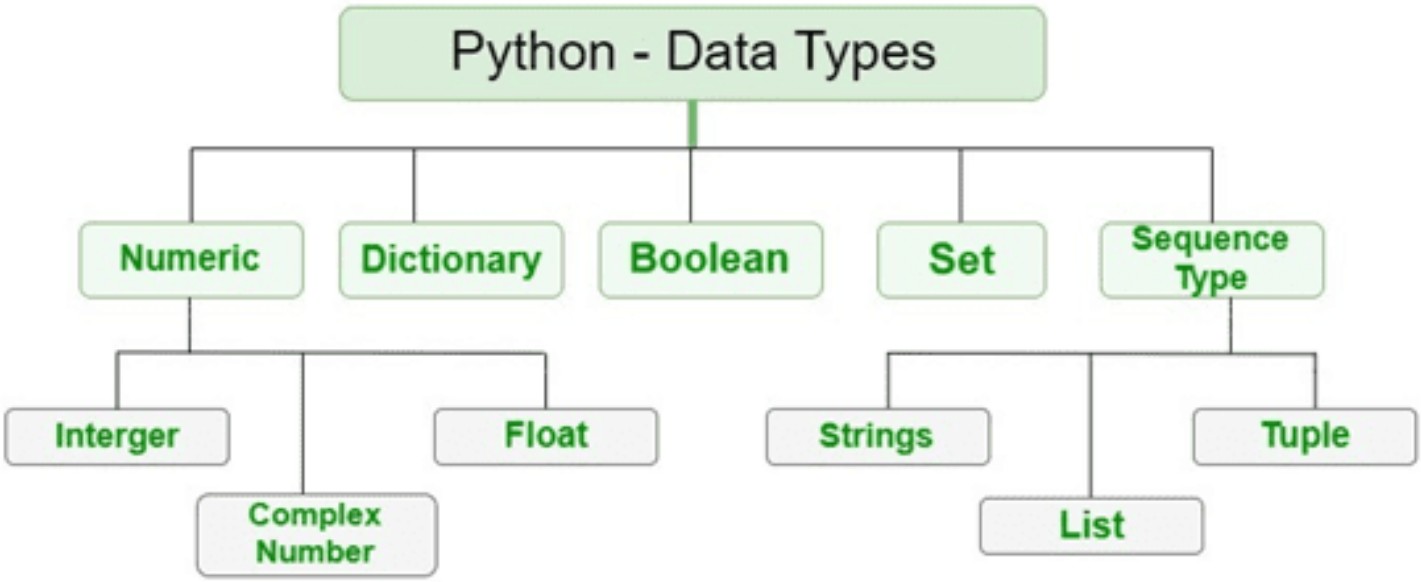
```
# Variable storing an integer
x = 5

# Variable storing a string
name = "Alice"

# Variable storing a floating-point number
pi = 3.14159

# Variable storing a list
```

numbers = [1, 2, 3, 4, 5]

Key Points:
- **Dynamic Typing**: You don't need to declare the type of a variable. Python determines the type based on the value you assign.
- **Case Sensitivity**: Variables are case-sensitive (`x` and `X` are different).
- **Naming Rules**: Variable names must start with a letter or underscore (`_`), and can contain letters, numbers, and underscores. They can't start with a number or be a Python reserved keyword (like `for`, `while`, etc.).

Would you like to see more specific examples of variable use in Python?

Built-in data type of python



Name	Type	Description
Integers	int	Whole numbers, such as: 3 300 200
Floating point	float	Numbers with a decimal point: 2.3 4.6 100.0
Strings	str	Ordered sequence of characters: "hello" 'Sammy' "2000" "楽しい"
Lists	list	Ordered sequence of objects: [10,"hello",200.3]
Dictionaries	dict	Unordered Key:Value pairs: {"mykey" : "value" , "name" : "Frankie"}
Tuples	tup	Ordered immutable sequence of objects: (10,"hello",200.3)
Sets	set	Unordered collection of unique objects: {"a","b"}
Booleans	bool	Logical value indicating True or False

In Python, **everything is an object**, and that includes primitive types like `int`, `float`, `bool`, `str`, and even more complex structures like `list`, `tuple`, `set`, `dict`, and others. Let's break this down in detail:

In Python, Everything is an Object

• Python is an object-oriented language, and everything you work with is an **instance (object)** of some **class**.
• Even basic types like integers, floats, booleans, and strings are instances of their respective classes.

For example:
◇ `int` is a class, and any integer value you use is an instance (object) of the `int` class.
◇ `float` is a class, and any floating-point number you use is an instance (object) of the `float` class.
◇ `bool` is a class, and `True` and `False` are objects of the `bool` class.

typecasting

Typecasting in Python, also known as **type conversion**, refers to the process of converting one data type to another. Python provides both **implicit** and **explicit** typecasting mechanisms, allowing you to seamlessly work with different data types as needed.

Understanding typecasting is essential for:

• **Data Manipulation:** Converting data types to perform specific operations.
• **Data Validation:** Ensuring data is in the correct format before processing.
• **Interoperability:** Facilitating operations between different data types.

Typecasting in Python, also known as **type conversion**, refers to the process of converting one data type to another. Python provides both **implicit** and **explicit** typecasting mechanisms, allowing you to seamlessly work with different data types as needed.
Understanding typecasting is essential for:
• **Data Manipulation:** Converting data types to perform specific operations.
• **Data Validation:** Ensuring data is in the correct format before processing.
• **Interoperability:** Facilitating operations between different data types.

Let's delve into the details of typecasting in Python.

1. Implicit Typecasting

Implicit typecasting occurs when Python automatically converts one data type to another without any explicit instruction from the programmer. This typically happens in

operations involving mixed data types to prevent data loss.

2. Explicit Typecasting

Explicit typecasting requires the programmer to manually convert data types using built-in functions. This is useful when automatic type conversion isn't possible or desired.

Common Typecasting Functions:

Function	Purpose
`int()`	Converts to integer
`float()`	Converts to float
`str()`	Converts to string
`list()`	Converts to list
`tuple()`	Converts to tuple
`set()`	Converts to set
`dict()`	Converts to dictionary (from key-value pairs)

arithmetic operators

Operator	Name	Description	Example	Result
`+`	Addition	Adds two operands	`5 + 3`	`8`
`-`	Subtraction	Subtracts the second operand from the first	`5 - 3`	`2`
`*`	Multiplication	Multiplies two operands	`5 * 3`	`15`
`/`	Division	Divides the first operand by the second (always returns float)	`5 / 2`	`2.5`
`//`	Floor Division	Divides and returns the integer part of the quotient	`5 // 2`	`2`
`%`	Modulus (Remainder)	Returns the remainder after division	`5 % 2`	`1`
`**`	Exponentiation	Raises the first operand to the power of the second	`5 ** 2`	`25`
`()`	Parentheses	Used to change the precedence of operations	`(5 + 3) * 2`	`16`

Operator Precedence

When multiple operators are used in an expression, **operator precedence** determines the order in which the operations are performed. Here's the hierarchy (from highest to lowest):

1. **Parentheses** ( )
2. **Exponentiation** **
3. **Unary plus and minus** +x, -x
4. **Multiplication**, **Division**, **Floor Division**, **Modulus** *, /, //, %
5. **Addition** and **Subtraction** +, -

Example:

```
result = 3 + 4 * 2 ** 2 / (1 - 5) % 2
print(result)  # Output: 3.0
```

Step-by-Step Evaluation:

1. **Parentheses:** `(1 - 5)` → `-4`
2. **Exponentiation:** `2 ** 2` → `4`
3. **Multiplication:** `4 * 4` → `16`
4. **Division:** `16 / -4` → `-4.0`
5. **Modulus:** `-4.0 % 2` → `0.0` (since `-4.0` is evenly divisible by `2`)
6. **Addition:** `3 + 0.0` → `3.0`

assignment Operator

Basic Assignment Operator (=)

The **basic assignment operator** = is used to assign a value to a variable. It's the most fundamental way to store data in Python.

Syntax:

variable_name = value

Examples:

```
# Assigning an integer
x = 10
print(x)  # Output: 10

# Assigning a string
name = "Alice"
print(name)  # Output: Alice

# Assigning a float
pi = 3.1415
print(pi)  # Output: 3.1415

# Assigning a boolean
is_active = True
print(is_active)  # Output: True
```

Key Points:

• **Right-Associative:** The assignment operator is right-associative, meaning the expression on the right is evaluated first, then assigned to the variable on the left.

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

Chained Assignment

Chained assignment allows you to assign the same value to multiple variables in a single line. This can make your code more concise.

```
a = b = c = 100
print(a)  # Output: 100
print(b)  # Output: 100
print(c)  # Output: 100

# Assigning the same list to multiple variables
list1 = list2 = []
list1.append(1)
print(list2)  # Output: [1]
```

Table of Augmented Assignment Operators:

Operator	Description	Equivalent To	Example
`+=`	Add and assign	`x = x + y`	`x += y`
`-=`	Subtract and assign	`x = x - y`	`x -= y`
`*=`	Multiply and assign	`x = x * y`	`x *= y`
`/=`	Divide and assign	`x = x / y`	`x /= y`
`//=`	Floor divide and assign	`x = x // y`	`x //= y`
`%=`	Modulus and assign	`x = x % y`	`x %= y`
`**=`	Exponentiate and assign	`x = x ** y`	`x **= y`
`&=`	Bitwise AND and assign	`x = x & y`	`x &= y`
`` ` ``	=`	Bitwise OR and assign	`x = x
`^=`	Bitwise XOR and assign	`x = x ^ y`	`x ^= y`
`<<=`	Left shift and assign	`x = x << y`	`x <<= y`
`>>=`	Right shift and assign	`x = x >> y`	`x >>= y`

Bitwise Assignment Operators (&=, |=, ^=, <<=, >>=)

```
x = 5      # Binary: 0101
y = 3      # Binary: 0011

x &= y     # Binary AND: 0001
print(x)   # Output: 1

x = 5
x |= y     # Binary OR: 0111
print(x)   # Output: 7
```

```
x = 5
x ^= y     # Binary XOR: 0110
print(x)   # Output: 6

x = 5
x <<= 1    # Left shift: 1010 (10 in decimal)
print(x)   # Output: 10

x = 5
x >>= 1    # Right shift: 0010 (2 in decimal)
print(x)   # Output: 2
```

Unpacking Assignments
Unpacking assignments allow you to assign values from iterable objects (like tuples, lists, or dictionaries) to multiple variables simultaneously. This enhances code readability and efficiency.

a. Tuple Unpacking

Assigning elements of a tuple to variables

```
# Example
point = (10, 20)
x, y = point
print(x)  # Output: 10
print(y)  # Output: 20
```

b. List Unpacking

Assigning elements of a list to variables.

```
# Example
colors = ["red", "green", "blue"]
first, second, third = colors
print(first)   # Output: red
print(second)  # Output: green
print(third)   # Output: blue
```

c. Dictionary Unpacking

Assigning keys or values of a dictionary to variables.

```
# Assigning keys
person = {"name": "Alice", "age": 30}
key1, key2 = person
print(key1)  # Output: name
print(key2)  # Output: age

# Assigning values using .values()
value1, value2 = person.values()
print(value1)  # Output: Alice
print(value2)  # Output: 30

# Assigning key-value pairs using .items()
(k1, v1), (k2, v2) = person.items()
```

```
print(k1, v1)  # Output: name Alice
print(k2, v2)  # Output: age 30
```

Key Points:

• **Matching Structure:** The number of variables on the left must match the number of elements in the iterable.

```
# This will raise a ValueError
a, b = [1, 2, 3]
```

• Using Asterisk (*) for Variable-Length Unpacking:

```
numbers = [1, 2, 3, 4, 5]
first, *middle, last = numbers
print(first)   # Output: 1
print(middle)  # Output: [2, 3, 4]
print(last)    # Output: 5
```

Immutable and Mutable

Object(Importand)

1. Immutable Objects

• **Definition:** Immutable objects cannot be changed once they are created.
• **Example:◇ Integers (`int`):** If you want to change the value of an integer, you must reassign it to a new integer. For example:

```
a = 10  # a points to the integer object 10
a = 20  # a now points to a new integer object 20
```

Behavior: If you "change" an immutable object, you are actually creating a new object instead of modifying the existing one.

• **Reassignment and Memory Address Change:**• When you reassign an immutable object, such as an integer, string, or tuple, you create a new object. The variable then points to this new object, which is stored at a different memory address.
• **Example:**

```
a = 10  # a points to memory address for the integer 10
print(id(a))  # Example output: 140634989756128 (some memory address)
```

```
a = 20  # Now a points to a new memory address for the integer 20
print(id(a))  # Example output: 140634989756144 (different memory address)
```

Mutable Objects

• **Definition:** Mutable objects can be changed after they are created.

Lists (`list`): You can change the contents of a list without needing to reassign the list variable. For example:

```
my_list = [1, 2, 3]  # my_list contains [1, 2, 3]
my_list.append(4)    # my_list is now [1, 2, 3, 4]
```

Behavior: Mutating a mutable object (like appending to a list) modifies the object in place, and all references to that object will reflect the changes.

• **Modification and Same Memory Address:**
  • When you modify a mutable object, such as a list, dictionary, or set, you are changing the object in place. The memory address remains the same because the object itself has not been replaced; its content has just been altered.

• **Example:**

```
my_list = [1, 2, 3]  # my_list points to some memory address
```

```
print(id(my_list))  # Example output: 140634989756224

my_list.append(4)    # Modify the list in place
print(my_list)       # Output: [1, 2, 3, 4]
print(id(my_list))  # Memory address remains the same: 140634989756224
```

Key Takeaway

• For **immutable objects**

■ you must reassign to change their value (like integers or strings).

■ Reassignment creates a new object.

■ The variable points to a different memory address after reassignment.

• For **mutable objects**

■ you can modify their contents directly (like lists or dictionaries) without reassignment.

■ Modifications occur in place, and the object retains the same memory address.

■ No reassignment is necessary to change their contents.

Comparison operators

1. Overview of Comparison Operators

Comparison operators in Python are used to compare two values. The result of a comparison is always a Boolean value: `True` or `False`.

2. List of Comparison Operators

Here's a comprehensive list of comparison operators available in Python:

Operator	Name	Description
`==`	Equal to	Checks if the values of two operands are equal
`!=`	Not Equal to	Checks if the values of two operands are not equal
`>`	Greater Than	Checks if the value on the left is greater than the right
`<`	Less Than	Checks if the value on the left is less than the right
`>=`	Greater Than or Equal	Checks if the value on the left is greater than or equal to the right
`<=`	Less Than or Equal	Checks if the value on the left is less than or equal to the right
`is`	Identity Operator	Checks if two operands refer to the same object in memory
`is not`	Negative Identity	Checks if two operands do not refer to the same object in memory
`in`	Membership Operator	Checks if a value exists within an iterable
`not in`	Negative Membership	Checks if a value does not exist within an iterable

3. Detailed Explanation and Examples

1. Equal to (==)

Description:

Checks if the values of two operands are equal. Returns `True` if they are equal, `False` otherwise.

```
a = 5
b = 5
c = 10

print(a == b)  # Output: True
print(a == c)  # Output: False
```

2. Not Equal to (!=)

Description:

Checks if the values of two operands are not equal. Returns `True` if they are not equal, `False` otherwise.

```
x = "Python"
y = "Java"
z = "Python"

print(x != y)  # Output: True
print(x != z)  # Output: False
```

Greater Than (>)

Description:
Checks if the value on the left is greater than the value on the right. Returns `True` if it is, `False` otherwise.

```
num1 = 15
num2 = 10

print(num1 > num2)  # Output: True
print(num2 > num1)  # Output: False
```

4. Less Than (`<`)

Description:
Checks if the value on the left is less than the value on the right. Returns `True` if it is, `False` otherwise.

```
a = 3
b = 7

print(a < b)  # Output: True
print(b < a)  # Output: False
```

5. Greater Than or Equal to (`>=`)

Description:
Checks if the value on the left is greater than or equal to the

value on the right. Returns `True` if it is, `False` otherwise.

```
x = 20
y = 20
z = 15

print(x >= y)  # Output: True
print(z >= y)  # Output: False
```

6. Less Than or Equal to (`<=`)
Description:
Checks if the value on the left is less than or equal to the value on the right. Returns `True` if it is, `False` otherwise.

```
m = 8
n = 12
p = 8

print(m <= n)  # Output: True
print(m <= p)  # Output: True
print(n <= m)  # Output: False
```

7. Identity Operators (`is`, `is not`)

Description:
• `is` checks if two operands refer to the **same object** in memory.
• `is not` checks if two operands do **not** refer to the same

object in memory.

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

print(a is b)      # Output: True
print(a is c)      # Output: False
print(a is not c)  # Output: True
```

8. Membership Operators (`in`, `not in`)

Description:

• `in` checks if a value exists within an **iterable** (like a list, tuple, string, etc.). Returns `True` if it does, `False` otherwise.
• `not in` checks if a value does **not** exist within an iterable. Returns `True` if it does not, `False` otherwise.

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)     # Output: True
print("grape" in fruits)      # Output: False
print("grape" not in fruits)  # Output: True

Additional Example with Strings:
text = "Hello, World!"
print("World" in text)      # Output: True
print("Python" in text)     # Output: False
```

```
print("Python" not in text) # Output: True
```

4. Operator Precedence

Understanding **operator precedence** helps predict the order in which operations are evaluated in complex expressions. In Python, comparison operators have a specific precedence level.

Precedence Hierarchy (Relevant to Comparison Operators):

1. **Parentheses** `()`
2. **Exponentiation** `**`
3. **Unary plus and minus** `+x`, `-x`
4. **Multiplication**, **Division**, **Floor Division**, **Modulus** `*`, `/`, `//`, `%`
5. **Addition** and **Subtraction** `+`, `-`
6. **Comparison Operators** `==`, `!=`, `>`, `<`, `>=`, `<=`, `is`, `is not`, `in`, `not in`
7. **Logical NOT** `not`
8. **Logical AND** `and`
9. **Logical OR** `or`

Example:

```
result = 3 + 4 > 5  # Evaluated as (3 + 4) > 5
print(result)       # Output: True
```

```
result = 3 + (4 > 5)  # Evaluated as 3 + False => 3 + 0 => 3
print(result)        # Output: 3
```

5. Chaining Comparison Operators

Example:

```
x = 5
print(1 < x < 10)  # Output: True
print(5 < x < 10)  # Output: False
```

Explanation:

- `1 < x < 10` checks if `x` is greater than `1` **and** less than `10`.
- `5 < x < 10` checks if `x` is greater than `5` **and** less than `10`.

Since `x` is `5`, this evaluates to `False`.

Logical operators

Overview of Logical Operators

Logical operators allow you to combine multiple conditional statements and make complex decisions in your code. Python provides three primary logical operators:

- `and`

- `or`
- `not`

These operators work with Boolean values (`True` and `False`) and expressions that evaluate to Boolean values.

List of Logical Operators

Operator	Name	Description
`and`	Logical AND	Returns `True` if **both** operands are `True`.
`or`	Logical OR	Returns `True` if **at least one** operand is `True`.
`not`	Logical NOT	Returns the **opposite** Boolean value of the operand.

Detailed Explanation and Examples

1. `and` Operator

Description:

The `and` operator returns `True` only if **both** operands are `True`. If either operand is `False`, the result is `False`.

Truth Table:

operand1	operand2	operand1 and operand2
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

Example:

```
# Example 1: Both conditions True
age = 25
has_license = True

can_drive = age >= 18 and has_license
print(can_drive)  # Output: True

# Example 2: One condition False
age = 16
has_license = False

can_drive = age >= 18 and has_license
print(can_drive)  # Output: False

# Example 3: Mixed conditions
age = 20
has_license = False

can_drive = age >= 18 and has_license
print(can_drive)  # Output: False
```

2. `or` Operator

Description:

The `or` operator returns `True` if **at least one** of the operands is `True`. If both operands are `False`, the result is

`False`.

Truth Table:

operand1	operand2	operand1 or operand2
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

Example:

```
# Example 1: Both conditions True
is_weekend = True
is_holiday = True

can_sleep_in = is_weekend or is_holiday
print(can_sleep_in)  # Output: True

# Example 2: One condition True
is_weekend = True
is_holiday = False

can_sleep_in = is_weekend or is_holiday
print(can_sleep_in)  # Output: True

# Example 3: Both conditions False
is_weekend = False
is_holiday = False
```

```
can_sleep_in = is_weekend or is_holiday
print(can_sleep_in)  # Output: False
```

3. `not` Operator

Description:

The `not` operator negates the Boolean value of its operand. If the operand is `True`, `not` returns `False`, and vice versa.

Truth Table:

operand	not operand
`True`	`False`
`False`	`True`

```
# Example 1: Negating True
is_sunny = True
is_not_sunny = not is_sunny
print(is_not_sunny)  # Output: False

# Example 2: Negating False
is_raining = False
is_not_raining = not is_raining
print(is_not_raining)  # Output: True

# Example 3: Using with conditions
age = 20
```

```
has_license = False

can_drive = age >= 18 and has_license
print(can_drive)  # Output: False

can_not_drive = not can_drive
print(can_not_drive)  # Output: True
```

4. Operator Precedence
Precedence Hierarchy (Relevant to Logical Operators):
1. **Parentheses** `()`
2. **Unary** `not` not
3. **Logical** `and` and
4. **Logical** `or` or

Example:
```
# Without parentheses
result = True or False and False
print(result)  # Output: True

# With parentheses to change precedence
result = (True or False) and False
print(result)  # Output: False
```

5. Short-Circuit Evaluation

a. `and` Operator:

• **Behavior:**

If the first operand is `False`, Python **does not evaluate** the second operand because the overall result cannot be `True`.

Example:

```
def first():
    print("First function called")
    return False

def second():
    print("Second function called")
    return True

result = first() and second()
print(result)

# Output:
# First function called
# False
```

b. `or` Operator:

• **Behavior:**

If the first operand is `True`, Python **does not evaluate** the second operand because the overall result is already `True`.

Example:

```
def first():
    print("First function called")
    return True

def second():
    print("Second function called")
    return False

result = first() or second()
print(result)

# Output:
# First function called
# True
```

6. Truthy and Falsy Values

In Python, values are inherently classified as **truthy** or **falsy** based on their Boolean value. Understanding this classification is essential when using logical operators.

a. Falsy Values:

These values are considered `False` in Boolean contexts:

- `False`
- `None`
- Zero of any numeric type: `0`, `0.0`, `0j`, etc.
- Empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`,

`range(0)`

- Objects of classes that implement `__bool__()` or `__len__()` returning `False` or `0`

b. Truthy Values:

Any value that is **not** falsy is considered `True`:

◇ `True`

◇ Non-zero numbers: `1`, `-1`, `3.14`, etc.

◇ Non-empty sequences and collections: `'a'`, `(1,)`, `[1, 2]`, `{'key': 'value'}`, `{1}`, `range(1)`

◇ Objects of classes that implement `__bool__()` or `__len__()` returning `True` or a positive number

7. Using Logical Operators in Control Flow

Logical operators are extensively used in control flow statements like `if`, `elif`, `while`, and more to make decisions based on multiple conditions.

a. Using `and` in `if` Statements

Example:

```
age = 22
has_license = True

if age >= 18 and has_license:
    print("You are eligible to drive.")
else:
```

```
    print("You are not eligible to drive.")
```

b. Using `or` in `if` Statements

Example:

```
is_weekend = False
is_holiday = True

if is_weekend or is_holiday:
    print("You can relax today!")
else:
    print("Time to work!")
```

c. Using `not` in `if` Statements

Example:

```
is_raining = False

if not is_raining:
    print("You don't need an umbrella today.")
else:
    print("Don't forget your umbrella!")
```

d. Combining Multiple Logical Operators

```
age = 30
has_license = True
is_insured = False

if (age >= 18 and has_license) or is_insured:
```

```
    print("Eligible for the driving test.")
else:
    print("Not eligible for the driving test.")
```

8. Advanced Topics

a. Combining Logical Operators with Other Operators
Logical operators can be combined with comparison operators, membership operators, and more to form complex conditions.

Example:

```
username = "admin"
password = "password123"

if username == "admin" and password == "password123":
    print("Access granted.")
else:
    print("Access denied.")
```

b. Logical Operators in List Comprehensions

Example:

```
numbers = range(1, 21)
filtered = [num for num in numbers if num % 2 == 0 and
num % 3 == 0]
print(filtered)  # Output: [6, 12, 18]
```

c. Using Logical Operators with Functions

You can use logical operators to combine multiple function calls that return Boolean values.

Example:

```
def is_even(num):
    return num % 2 == 0

def is_positive(num):
    return num > 0

number = 4

if is_even(number) and is_positive(number):
    print("Number is positive and even.")
else:
    print("Number does not meet the criteria.")
```

d. Ternary Conditional Operator with Logical Operators

Combine logical operators with Python's ternary conditional operator for concise conditional expressions.

Example:

```
age = 20
```

```
has_permission = True

status = "Allowed" if age >= 18 and has_permission else "Not Allowed"
print(status)  # Output: Allowed
```

Comments

Types of Comments in Python
Python supports several types of comments, each serving different purposes. Understanding these types will help you effectively annotate your code.

1. Single-Line Comments

Syntax:

```
# This is a single-line comment
```

Example:

```
# Calculate the area of a circle
radius = 5
area = 3.14159 * radius ** 2  # Area formula: πr²
print(area)
```

2. Multi-Line Comments

Approach 1: Multiple Single-Line Comments

```
# This is a multi-line comment.
# It spans several lines.
# Each line starts with a hash symbol.
```

Approach 2: Multi-Line Strings (Not Recommended for Comments)

```
"""
This is a multi-line string,
not a true comment.
It can be used as a comment,
but it's intended for docstrings.
"""
```

Note:
While multi-line strings can be used as comments, it's generally recommended to use multiple single-line comments for clarity and to adhere to best practices.

3. Docstrings

Definition:
Docstrings (documentation strings) are multi-line comments used to document modules, classes, functions, and methods. Unlike regular comments, docstrings are accessible at runtime via the `__doc__` attribute and are used by documentation tools.

Syntax:
• Enclosed in triple quotes (`"""` or `'''`).
• Placed immediately after the definition of a function, class, or module.

```
def greet(name):
    """
    Greet the user by name.

    Parameters:
    name (str): The name of the user.

    Returns:
    str: A greeting message.
    """
    return f"Hello, {name}!"
```

Explanation:
• The docstring provides a description of what the function does, its parameters, and its return value.
• Tools like `help()` and documentation generators (e.g., Sphinx) utilize docstrings to create documentatio

Accessing Docstrings:

```
print(greet.__doc__)
```

Output:
Greet the user by name.

Parameters:
name (str): The name of the user.

Returns:
str: A greeting message.

List

Lists are one of the most versatile and commonly used data structures in Python. They allow you to store, organize, and manipulate collections of items efficiently. Whether you're dealing with numbers, strings, or even other lists, Python lists provide a flexible way to handle data.

Declaring Lists

A **list** in Python is an ordered, mutable (changeable) collection of items. Lists are defined by enclosing elements within square brackets `[]`, separated by commas.

Examples:

```
# An empty list
empty_list = []

# List of integers
numbers = [1, 2, 3, 4, 5]

# List of strings
fruits = ["apple", "banana", "cherry"]

# Mixed data types
mixed = [1, "hello", 3.14, True]

# List of lists (nested list)
nested_list = [[1, 2], [3, 4], [5, 6]]
```

Creating Lists Using the `list()` Constructor:

```
# From a tuple
tuple_data = (1, 2, 3)
list_from_tuple = list(tuple_data)
```

```
print(list_from_tuple)  # Output: [1, 2, 3]

# From a string
string_data = "hello"
list_from_string = list(string_data)
print(list_from_string)  # Output: ['h', 'e', 'l', 'l', 'o']
```

Accessing Items

You can access items in a list by referring to their **index**. Python uses **zero-based indexing**, meaning the first item has an index of `0`, the second item has an index of `1`, and so on.

Examples:

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]

# Access the first item
print(fruits[0])  # Output: apple

# Access the third item
```

```
print(fruits[2])  # Output: cherry

# Access the last item using negative indexing
print(fruits[-1])  # Output: elderberry

# Access the second-to-last item
print(fruits[-2])  # Output: date

Handling Index Errors:
print(fruits[10])  # Raises IndexError: list index out of range
```
Tip: To avoid errors, ensure that the index is within the valid range using `len()` or try-except blocks.

Changing Items

Lists are **mutable**, meaning you can change their content after creation by assigning a new value to a specific index.

Examples:

```
numbers = [10, 20, 30, 40, 50]

# Change the first item
```

```
numbers[0] = 15
print(numbers)  # Output: [15, 20, 30, 40, 50]

# Change the last item using negative indexing
numbers[-1] = 55
print(numbers)  # Output: [15, 20, 30, 40, 55]

# Change a middle item
numbers[2] = 35
print(numbers)  # Output: [15, 20, 35, 40, 55]
```

Removing List Items

Removing List Items
Python provides several methods to remove items from a list. The choice of method depends on whether you know the item's index, the item's value, or need to remove items based on a condition

a. `remove()` Method

Syntax:
my_list.remove(value)

Removes the **first occurrence** of a specified value.
fruits = ["apple", "banana", "cherry", "banana", "date"]

Example:
# Remove the first 'banana'
fruits.remove("banana")
print(fruits) # Output: ['apple', 'cherry', 'banana', 'date']

Note: If the value is not found, a `ValueError` is raised.

b. `pop()` Method

Removes an item at a specified index and **returns** it. If no index is specified, it removes and returns the **last item**.

Syntax:
item = my_list.pop(index)

Examples:
numbers = [10, 20, 30, 40, 50]

# Remove and return the item at index 2
removed_item = numbers.pop(2)

```
print(removed_item)  # Output: 30
print(numbers)       # Output: [10, 20, 40, 50]

# Remove and return the last item
last_item = numbers.pop()
print(last_item)  # Output: 50
print(numbers)    # Output: [10, 20, 40]
```

c. `del` Statement

Removes an item or a **slice** from the list. It does not return the removed item.

Syntax:

```
del my_list[index]
del my_list[start:end]
```

Examples:

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]

# Remove the item at index 1
del fruits[1]
print(fruits)  # Output: ['apple', 'cherry', 'date', 'elderberry']

# Remove a slice (first two items)
del fruits[0:2]
print(fruits)  # Output: ['date', 'elderberry']
```

d. `clear()` Method

Removes **all items** from the list, resulting in an empty list.

Syntax:

my_list.clear()

Example:

numbers = [1, 2, 3, 4, 5]
numbers.clear()
print(numbers) # Output: []

Indexing

Indexing allows you to access elements in a list based on their position. Python supports both **positive** and **negative** indexing.

a. Positive Indexing: Starts from `0` (first element) to `n-1` (last element).

Example:

```
colors = ["red", "green", "blue", "yellow", "purple"]

print(colors[0])  # Output: red
print(colors[2])  # Output: blue
print(colors[4])  # Output: purple
```

b. Negative Indexing: Starts from `-1` (last element) to `-n` (first element).

Example:

```
colors = ["red", "green", "blue", "yellow", "purple"]

print(colors[-1])  # Output: purple
print(colors[-3])  # Output: blue
print(colors[-5])  # Output: red
```

c. Accessing Elements in Nested Lists

Example:

```
nested_list = [
    [1, 2, 3],
    ["a", "b", "c"],
    [True, False]
]

print(nested_list[0][1])  # Output: 2
print(nested_list[1][2])  # Output: c
```

```
print(nested_list[2][0])  # Output: True
```

Slicing

Slicing allows you to access a **subset** of a list by specifying a **range** of indices. The syntax uses the colon `:` operator.

Syntax:
```
subset = my_list[start:stop:step]
```

• **start:** Starting index (inclusive). Defaults to `0` if omitted.
• **stop:** Ending index (exclusive). Defaults to the end of the list if omitted.
• **step:** Step size. Defaults to `1` if omitted.

Examples:
```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get items from index 2 to 5
subset = numbers[2:6]
print(subset)  # Output: [2, 3, 4, 5]
```

```
# Get items from the beginning to index 4
subset = numbers[:5]
print(subset)  # Output: [0, 1, 2, 3, 4]

# Get items from index 5 to the end
subset = numbers[5:]
print(subset)  # Output: [5, 6, 7, 8, 9]

# Get every second item from index 1 to 7
subset = numbers[1:8:2]
print(subset)  # Output: [1, 3, 5, 7]

# Reverse the list using slicing
reversed_list = numbers[::-1]
print(reversed_list)  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Slicing with Negative Indices:

```
colors = ["red", "green", "blue", "yellow", "purple"]

# Get last two items
last_two = colors[-2:]
print(last_two)  # Output: ['yellow', 'purple']

# Get items from index -4 to -1
subset = colors[-4:-1]
print(subset)  # Output: ['green', 'blue', 'yellow']
```

Note:

• The `stop` index is **exclusive**, meaning the element at that index is **not** included in the result.
• If `start` is greater than `stop` with a positive `step`, the result is an empty list.

5. Summary of Default Values in Slicing

Slicing Syntax	Step	Default Start	Default Stop
`list[start:stop:step]`	Positive	`0`	`len(list)`
`list[start:stop:step]`	Negative	`len(list) - 1`	`-1` (exclusive)

5. Summary of Default Values in Slicing

Slicing Syntax	Step	Default Start	Default Stop
`list[start:stop:step]`	Positive (`> 0`)	`0`	`len(list)`
`list[start:stop:step]`	Negative (`< 0`)	`-1` (or `len(list) - 1`)	`-(len(list) + 1)` (exclusive)

```
# Important Note
# //print(numbers[-1 or 9 is same ::-1])
#
# in stop: -1 (If I put it, Python Thinking the last element =-1)
# So It print: []
#
#  start: -1[inclusive]
```

```
#  end: -1 [execlusive]
#
#  There is nothing; that's why it print []
```

Example:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[-1::-1])
print(numbers[len(numbers)-1::-1])
print(numbers[::1])
```

Sorting Lists

Sorting arranges the elements of a list in a specified order, either **ascending** or **descending**. Python provides built-in methods to sort lists.

a. `sort()` Method

Sorts the list **in place**, modifying the original list.

Syntax:

```
my_list.sort(reverse=False, key=None)
```

• **reverse (bool):** If `True`, sort the list in **descending** order.

• **key (function):** A function that serves as a **key** for the sort comparison.

Examples:

```
# Sorting a list of numbers in ascending order
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 5, 6, 9]

# Sorting in descending order
numbers.sort(reverse=True)
print(numbers)  # Output: [9, 6, 5, 5, 2, 1]

# Sorting a list of strings
fruits = ["banana", "apple", "cherry", "date"]
fruits.sort()
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']

# Sorting based on the length of strings
fruits.sort(key=len)
print(fruits)  # Output: ['date', 'apple', 'banana', 'cherry']
```

Examples:

```
# Sorting a list of numbers in ascending order
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 5, 6, 9]
```

```
# Sorting in descending order
numbers.sort(reverse=True)
print(numbers)  # Output: [9, 6, 5, 5, 2, 1]

# Sorting a list of strings
fruits = ["banana", "apple", "cherry", "date"]
fruits.sort()
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']

# Sorting based on the length of strings
fruits.sort(key=len)
print(fruits)  # Output: ['date', 'apple', 'banana', 'cherry']
```

b. `sorted()` Function

Returns a **new sorted list** without modifying the original list.

Syntax:

```
sorted_list = sorted(my_list, reverse=False, key=None)
```

Examples:

```
numbers = [3, 1, 4, 1, 5, 9]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 1, 3, 4, 5, 9]
print(numbers)         # Output: [3, 1, 4, 1, 5, 9] (original list
```

remains unchanged)

# Sorting strings in descending order
fruits = ["banana", "apple", "cherry", "date"]

sorted_fruits = sorted(fruits, reverse=True)
print(sorted_fruits) # Output: ['date', 'cherry', 'banana', 'apple']

Sorting with Keys

To sort based on multiple criteria, the key function can return a tuple.

Example:

# Sorting a list of tuples based on the second element
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]
students.sort(key=lambda student: student[1])
print(students) # Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]

`lambda student: student[1]`:

• This is a small anonymous function defined using `lambda`.
• It takes a single argument `student`, which represents each tuple in the `students` list.
• `student[1]` returns the second element of the tuple

(which is the age).

Sorting Order

• The `sort()` method arranges the tuples in ascending order based on the values returned by the key function. So the sorting will work as follows:

◇ Compare ages: `20`, `23`, `25`

◇ Sorted order will be:

■ `("Bob", 20)`

■ `("Charlie", 23)`

■ `("Alice", 25)`

d. Sorting with Multiple Keys

To sort based on multiple criteria, the key function can return a tuple.

Example:

```
# Sorting by age, then by name
people = [
```

```
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 20},
    {"name": "Charlie", "age": 25},
    {"name": "Dave", "age": 20}
]



# Sort first by age, then by name
people_sorted = sorted(people, key=lambda person:
(person["age"], person["name"]))
print(people_sorted)
# Output:
# [
#     {'name': 'Bob', 'age': 20},
#     {'name': 'Dave', 'age': 20},
#     {'name': 'Alice', 'age': 25},
#     {'name': 'Charlie', 'age': 25}
# ]
```

Joining Lists

Joining (or concatenating) lists combines two or more lists into a single list. Python provides several ways to achieve this.

a. Using the `+` Operator

The `+` operator concatenates two lists, returning a new list.

Syntax:

```
combined_list = list1 + list2
```

Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 + list2
print(combined)  # Output: [1, 2, 3, 4, 5, 6]
```

b. Using the `extend()` Method

The `extend()` method appends all elements from another list to the **end** of the current list. This modifies the original list in place.

Syntax:

```
list1.extend(list2)
```

Example:

```
list1 = ["a", "b", "c"]
list2 = ["d", "e", "f"]
list1.extend(list2)
print(list1)  # Output: ['a', 'b', 'c', 'd', 'e', 'f']
```

c. Using the * Operator (Replicating Lists)

The * operator can be used to **repeat** a list multiple times.

Syntax:

```
repeated_list = my_list * n
```

Example:

```
letters = ["x", "y"]
repeated = letters * 3
print(repeated)  # Output: ['x', 'y', 'x', 'y', 'x', 'y']
```

d. Using List Comprehensions

List comprehensions can be used to combine lists in more complex ways.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Combine lists with list comprehension
combined = [item for sublist in [list1, list2] for item in sublist]
print(combined)  # Output: [1, 2, 3, 4, 5, 6]
```

e. Using the `itertools` Module

For more advanced list joining, the `itertools` module provides tools like `chain()`.

Example:

```
import itertools

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

combined = list(itertools.chain(list1, list2, list3))
print(combined)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Appending to Lists

Appending adds a new element to the **end** of a list. Python provides the `append()` and `insert()` methods for adding elements.

a. `append()` Method

Adds a single element to the end of the list.

Syntax:

```
my_list.append(element)
```

Example:
fruits = ["apple", "banana", "cherry"]
fruits.append("date")
print(fruits) # Output: ['apple', 'banana', 'cherry', 'date']

b. `insert()` Method

Inserts an element at a specified index.

Syntax:
my_list.insert(index, element)

Example:
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "blueberry") # Insert at index 1
print(fruits) # Output: ['apple', 'blueberry', 'banana', 'cherry']

`extend()` Method

As discussed earlier, `extend()` adds multiple elements from another list.

Example:
numbers = [1, 2, 3]
numbers.extend([4, 5, 6])

```
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]
```

Counting Item

Counting the number of occurrences of a specific element in a list can be done using the `count()` method.

Syntax:

```
count = my_list.count(element)
```

Example:

```
numbers = [1, 2, 3, 2, 4, 2, 5]

# Count occurrences of 2
count_twos = numbers.count(2)
print(count_twos)  # Output: 3

# Count occurrences of 6 (which is not in the list)
count_sixes = numbers.count(6)
print(count_sixes)  # Output: 0
```

Use Case:

Counting items is useful for frequency analysis, statistical calculations, and conditional operations based on the number of occurrences.

Additional List Operations

a. `index()` Method

Finds the **first index** of a specified value. Raises a `ValueError` if the value is not found.

Syntax:

index = my_list.index(value, start, end)

- **start (optional):** Start searching from this index.
- **end (optional):** Stop searching before this index.

```
fruits = ["apple", "banana", "cherry", "banana", "date"]

# Find the index of 'banana'
index_banana = fruits.index("banana")
print(index_banana)  # Output: 1
```

```
# Find the index of 'banana' starting from index 2
index_banana_from_2 = fruits.index("banana", 2)
print(index_banana_from_2)  # Output: 3
```

b. `reverse()` Method

Reverses the elements of the list **in place**.

Syntax:

```
my_list.reverse()
```

Example:

```
numbers = [1, 2, 3, 4, 5]
numbers.reverse()
print(numbers)  # Output: [5, 4, 3, 2, 1]
```

c. `copy()` Method

Creates a **shallow copy** of the list.

Syntax

```
new_list = my_list.copy()
```

Example:

```
original = [1, 2, 3]
copied = original.copy()
copied.append(4)
```

```
print(original)  # Output: [1, 2, 3]
print(copied)    # Output: [1, 2, 3, 4]
```

d. `clear()` **Method**
Removes **all items** from the list, resulting in an empty list.

Syntax:
```
my_list.clear()
```

Example:
```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits)  # Output: []
```

e. `extend()` vs `append()`

• `append()`**:** Adds its argument as a single element to the end of the list.
• `extend()`**:** Iterates over its argument and adds each element to the list, extending it.

Example:
```
list1 = [1, 2, 3]

# Using append()
list1.append([4, 5])
```

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

# Reset list1
list1 = [1, 2, 3]

# Using extend()
list1.extend([4, 5])
print(list1)  # Output: [1, 2, 3, 4, 5]
```

List Comprehensions

List comprehensions provide a concise way to create lists based on existing lists. They can include conditional logic and transformations.

Basic Syntax:

```
new_list = [expression for item in iterable if condition]
```

Examples:

```
# Create a list of squares
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Create a list of even numbers
```

```
evens = [x for x in range(20) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# Create a list of uppercase fruits
fruits = ["apple", "banana", "cherry"]
uppercase_fruits = [fruit.upper() for fruit in fruits]
print(uppercase_fruits)  # Output: ['APPLE', 'BANANA',
'CHERRY']

# Nested list comprehension
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Nested Lists

Nested lists are lists within lists, allowing you to create multi-dimensional data structures.

Examples:

```
# Creating a nested list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
```

```
    [7, 8, 9]
]

print(matrix)
# Output:
# [
#     [1, 2, 3],
#     [4, 5, 6],
#     [7, 8, 9]
# ]

# Accessing elements in a nested list
print(matrix[0][1])  # Output: 2 (first row, second column)
print(matrix[2][0])  # Output: 7 (third row, first column)

# Modifying an element
matrix[1][2] = 60
print(matrix)
# Output:
# [
#     [1, 2, 3],
#     [4, 5, 60],
#     [7, 8, 9]
# ]
```

Copying Lists

Copying Lists

Copying lists involves creating a new list that contains the same elements as the original. Python provides different ways to copy lists, each with its own implications

a. Shallow Copy

A **shallow copy** creates a new list object but **does not** create copies of nested objects. Instead, it references the same nested objects.

Methods to Create a Shallow Copy:

1. Using the `copy()` method
2. Using slicing `[:]`
3. Using the `list()` constructor

Examples:

```
import copy

original = [1, 2, [3, 4]]

# Using the copy() method
shallow_copy1 = original.copy()

# Using slicing
shallow_copy2 = original[:]
```

```
# Using the list() constructor
shallow_copy3 = list(original)

print(shallow_copy1)  # Output: [1, 2, [3, 4]]
print(shallow_copy2)  # Output: [1, 2, [3, 4]]
print(shallow_copy3)  # Output: [1, 2, [3, 4]]

# Modifying a nested list in the shallow copy affects the original
shallow_copy1[2].append(5)
print(original)       # Output: [1, 2, [3, 4, 5]]
print(shallow_copy1)  # Output: [1, 2, [3, 4, 5]]
```

b. Deep Copy

A **deep copy** creates a new list and **recursively copies** all nested objects, resulting in complete independence from the original list.

Method to Create a Deep Copy:

• Using the `deepcopy()` function from the `copy` module

Example:

```
import copy

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

```
deep_copy = copy.deepcopy(original)

# Modifying the deep copy does not affect the original
deep_copy[2].append(5)
print(original)   # Output: [1, 2, [3, 4]]
print(deep_copy)  # Output: [1, 2, [3, 4, 5]]
```

Differences Between Shallow and Deep Copies

Feature	Shallow Copy	Deep Copy
Copies nested objects?	No (references same nested objects)	Yes (creates independent copies)
Affects original when nested objects change?	Yes	No

Use Cases:

• **Shallow Copy:** When you need a copy of the list structure but are okay with shared nested objects.
• **Deep Copy:** When you require complete independence, especially with complex, nested data structures.

Control Flow

Conditional Statements

1. Introduction to Conditional Statements

Conditional statements allow your program to execute certain pieces of code based on whether a condition is `True` or `False`. They are essential for creating dynamic programs that can handle different scenarios.

Why Use Conditional Statements?

• **Decision Making**: Choose different paths of execution based on input or other factors.
• **Control Flow**: Direct the flow of your program to perform tasks only when specific conditions are met.
• **Flexibility**: Make your code adaptable to varying situations.

The if Statement

The `if` Statement

The `if` statement is the most basic form of a conditional statement. It allows you to execute a block of code only if a specified condition is `True`.

Syntax

```
if condition:
    # Code block to execute if condition is True
```

Example

```
age = 18

if age >= 18:
    print("You are an adult.")
```

Output:

```
You are an adult.
```

The if-else Statement

An `if-else` statement allows you to execute one block of code if a condition is `True` and another block if the condition is `False`.

Syntax

```
if condition:
    # Code block if condition is True
else:
    # Code block if condition is False
```

Example

```
age = 16

if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

Output:

```
You are not an adult.
```

The if-elif-else Statement

The `if-elif-else` statement allows you to check multiple conditions sequentially. Once a condition is `True`, the corresponding block is executed, and the rest are skipped.

Syntax

```
if condition1:
    # Code block if condition1 is True
elif condition2:
    # Code block if condition2 is True
elif condition3:
    # Code block if condition3 is True
...
else:
    # Code block if none of the above conditions are True
```

Example

```
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

Nested if Statements

Nested `if` statements are `if` statements placed inside another `if` statement. They allow for more granular decision-making based on multiple layers of conditions.

Syntax

```
if condition1:
    # Code block if condition1 is True
    if condition2:
        # Code block if condition2 is True
    else:
        # Code block if condition2 is False
else:
    # Code block if condition1 is False
```

Example

```
age = 20
has_license = True

if age >= 18:
    print("You are eligible to drive.")
    if has_license:
```

```
        print("You can drive a car.")
    else:
        print("You need a driver's license to drive.")
else:
    print("You are not eligible to drive.")
```

Output:
You are eligible to drive.
You can drive a car.

Using Logical Operators in Conditions

Logical operators allow you to combine multiple conditions within a single `if`, `elif`, or `else` statement.

Logical Operators

- `and`: Both conditions must be `True`.
- `or`: At least one condition must be `True`.
- `not`: Inverts the truth value of the condition.

Examples

a. Using `and`

```
age = 25
has_license = True

if age >= 18 and has_license:
    print("You can drive a car.")
```

Output:

```
You can drive a car.
```

b. Using `or`

```
is_weekend = True
has_free_time = False

if is_weekend or has_free_time:
    print("You can go out.")
```

Output:

```
You can go out.
```

c. Using `not`

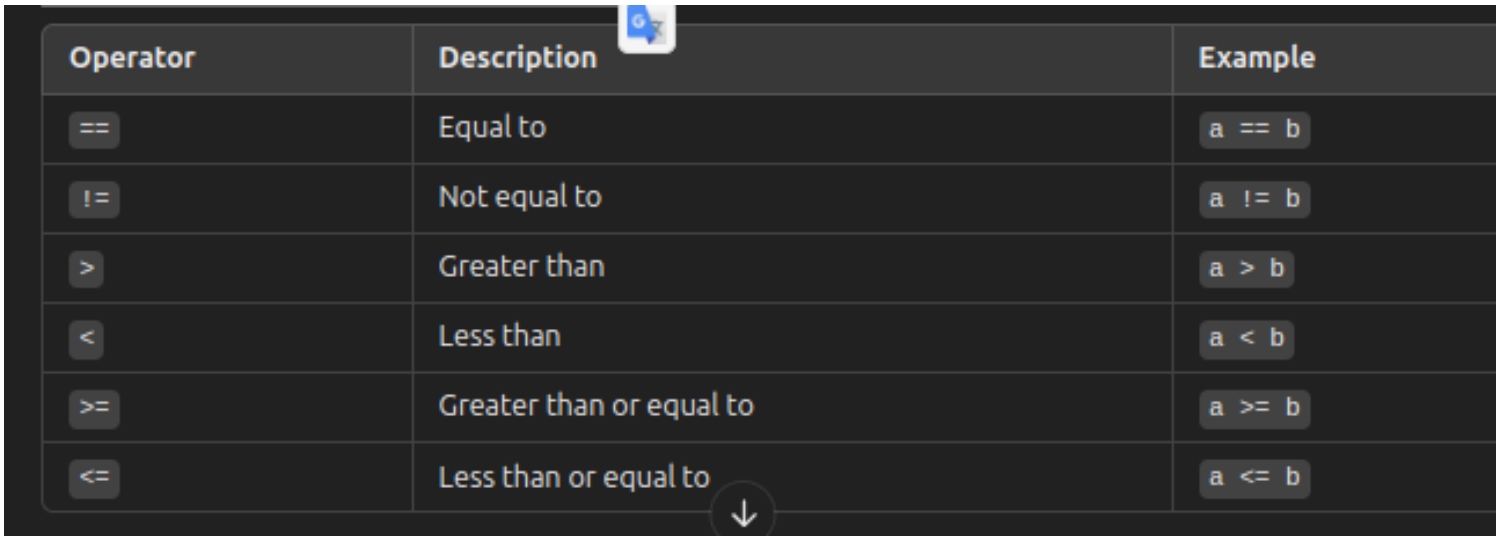
```
is_raining = False

if not is_raining:
    print("You don't need an umbrella.")
```

Output:
You don't need an umbrella.

Comparison Operators

Comparison operators are used to compare two values. They return `True` or `False` based on the comparison.

List of Comparison Operators



Operator	Description	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater than or equal to	`a >= b`
`<=`	Less than or equal to	`a <= b`

Examples
x = 10
y = 20

Equal to
print(x == y) # Output: False

```
# Not equal to
print(x != y)  # Output: True

# Greater than
print(x > y)   # Output: False

# Less than
print(x < y)   # Output: True

# Greater than or equal to
print(x >= 10) # Output: True

# Less than or equal to
print(y <= 20) # Output: True
```

Conditional Expressions (Ternary Operator)

Python offers a concise way to write conditional statements using **conditional expressions**, also known as the **ternary operator**. This allows you to assign a value to a variable based on a condition in a single line.

Syntax
variable = value_if_true if condition else value_if_false

Example
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status) # Output: Adult

Best Practices

Avoid Deep Nesting:
• Excessive nesting can make code hard to read. Consider refactoring using functions or logical operators

```
# Deeply nested
if condition1:
    if condition2:
        if condition3:
            do_something()

# Refactored
if condition1 and condition2 and condition3:
    do_something()
```

Use `elif` Instead of Multiple `if` Statements:
• Using `elif` ensures that only one block is executed, improving efficiency.

```
# Using multiple ifs
if x > 0:
    print("Positive")
if x == 0:
    print("Zero")
if x < 0:
    print("Negative")

# Using elif
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

Common Mistakes

Using Assignment Instead of Comparison `==`:

```
age = 18

# Incorrect
```

```
if age = 18:
    print("You are 18 years old.")

# Correct
if age == 18:
    print("You are 18 years old.")
```

Logical Errors in Conditions:

```
# Intended: Age between 18 and 25
if age >= 18 and age <= 25:
    print("Young adult")

# Common mistake: Using or
if age >= 18 or age <= 25:
    print("Young adult")  # This condition is always True for age >= 18
```

Advanced Topics

a. **Chained Comparisons**

Python allows you to chain multiple comparisons in a single

statement, making the code more concise and readable.

Example:

```
age = 25

if 18 <= age < 30:
    print("Young adult")
```

Output:

Young adult

b. **Using Functions in Conditions**

Encapsulating conditions within functions can make code more modular and reusable.

Example:

```
def is_adult(age):
    return age >= 18

age = 20

if is_adult(age):
    print("You are an adult.")
```

Output:

You are an adult.

c. **Short-Circuit Evaluation**

Logical operators in Python use short-circuit evaluation, meaning the second condition is evaluated only if necessary.

Example:

```
def check_condition():
    print("Checking condition...")
    return True

if False and check_condition():
    print("This won't print.")
```

Output:

Exercise

Examples and Exercises

Example 1: Determine Pass or Fail

Example:

```
score = 75

if score >= 60:
    print("Pass")
else:
    print("Fail")
```

Output:

```
Pass
```

Example 2: Grading System

```
score = 88

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
```

```
    grade = "D"
else:
    grade = "F"

print(f"Grade: {grade}")
```

Output:

```
Grade: B
```

Example 3: Nested `if` Statements

```
age = 22
has_license = True

if age >= 18:
    print("Eligible to drive.")
    if has_license:
        print("You can drive a car.")
    else:
        print("You need a driver's license to
drive.")
else:
    print("Not eligible to drive.")
```

Output:

```
Eligible to drive.
You can drive a car.
```

Exercise 4: Even or Odd

Task: Write a program that checks if a number is even or odd.

```
number = 7

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

Expected Output:

```
Odd
```

Exercise 5: Leap Year Checker

Task: Determine if a given year is a leap year.

Rules:

• A year is a leap year if it is divisible by 4 **and not** divisible by 100, **unless** it is also divisible by 400.

```
year = 2000

if (year % 4 == 0 and year % 100 != 0) or
(year % 400 == 0):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

Exercise 6: Maximum of Three Numbers

Task: Find the maximum of three numbers using `if-elif-else`.

```
a = 10
b = 25
c = 20

if a >= b and a >= c:
    max_num = a
elif b >= a and b >= c:
    max_num = b
else:
    max_num = c
```

```
print(f"The maximum number is {max_num}.")
```

Expected Output:

```
The maximum number is 25.
```

Loop Statements

Loops are a fundamental concept in Python (and programming in general) that allow you to execute a block of code multiple times. They are essential for tasks that require repetition, such as iterating over items in a list, processing data, or performing repetitive calculations.

In Python, there are two primary types of loops:
1. `for` **Loops**
2. `while` **Loops**

Additionally, Python provides **loop control statements** that modify the behavior of loops:
• `break`
• `continue`

• `pass`

The for Loop

The `for` loop in Python is used for iterating over a sequence (such as a list, tuple, dictionary, set, or string). It allows you to execute a block of code once for each item in the sequence.

Syntax

```
for variable in sequence:
    # Code block to execute for each item
```

• `variable`: A name you choose to refer to the current item in the sequence.
• `sequence`: The collection of items you want to iterate over.

Example

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Output:
apple
banana
cherry

Iterating Over Different Data Types

a. **List**

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    print(num)
```

Output:
1
2
3
4
5

b. **Tuple**

```
colors = ("red", "green", "blue")

for color in colors:
    print(color)
```

Output:
red
green
blue

c. **Dictionary**
When iterating over a dictionary, the default behavior is to iterate over its keys.

```
student_ages = {"Alice": 25, "Bob": 22, "Charlie": 23}

for student in student_ages:
    print(student)
```

Output:
Alice
Bob
Charlie

To iterate over both keys and values:

```
for student, age in student_ages.items():
    print(f"{student} is {age} years old.")
```

Output:
Alice is 25 years old.

Bob is 22 years old.
Charlie is 23 years old.

d. **String**

Iterating over a string allows you to access each character individually.

```
word = "Python"

for letter in word:
    print(letter)
```

Output

```
P
y
t
h
o
n
```

Using `range()` with `for` Loops

The `range()` function generates a sequence of numbers, which is often used with `for` loops for iteration a specific

number of times.

a. **Basic Usage**

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

b. **Specifying Start and End**

```
for i in range(2, 6):
    print(i)
```

Output:

```
for i in range(2, 6):
    print(i)
```

c. **Specifying Step**

```
for i in range(0, 10, 2):
    print(i)
```

Output:

```
0
2
4
6
8
```

Nested `for` Loops

You can place one `for` loop inside another to handle multi-dimensional data structures.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for num in row:
        print(num, end=' ')
    print()  # Newline after each row
```

Output:

```
1 2 3
4 5 6
7 8 9
```

The while Loop

The `while` loop in Python repeatedly executes a block of code as long as a given condition is `True`.

Syntax

```
while condition:
    # Code block to execute repeatedly
```

`condition`: A boolean expression that is evaluated before each iteration. If `True`, the loop continues; if `False`, the loop stops

Example

```
count = 0

while count < 5:
    print(count)
    count += 1  # Increment count to eventually break the loop
```

Output:

```
0
1
```

```
2
3
4
```

Preventing Infinite Loops
An **infinite loop** occurs when the loop's condition never becomes `False`. To prevent this, ensure that the condition will eventually be met.

Example of an Infinite Loop (Avoid!)

```
count = 0

while count < 5:
    print(count)
    count += 1
```

Nested `while` Loops
Similar to nested `for` loops, you can nest `while` loops to handle more complex scenarios.

```
i = 1

while i <= 3:
    j = 1
    while j <= 2:
        print(f"i={i}, j={j}")
        j += 1
```

```
    i += 1
```

Output:
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
i=3, j=1
i=3, j=2

Loop Control Statements

Loop control statements alter the behavior of loops. Python provides three primary loop control statements:
• `break`
• `continue`
• `pass`

`break`

The `break` statement terminates the nearest enclosing loop, causing the program to exit the loop immediately.

Example
```
for num in range(10):
```

```
    if num == 5:
        break  # Exit the loop when num is 5
    print(num)
```

Output:

```
0
1
2
3
4
```

`continue`

The `continue` statement skips the rest of the code inside the loop for the current iteration and moves to the next iteratio

Example

```
for num in range(5):
    if num == 2:
        continue  # Skip the rest when num is 2
    print(num)
```

Output:

```
0
1
3
```

4

`pass`

The `pass` statement is a **null operation**; it does nothing. It is useful as a placeholder when a statement is syntactically required but no action is needed.

Loop else Clause

Loop `else` Clause

In Python, loops (`for` and `while`) can have an `else` clause. The code inside the `else` block is executed **only if the loop completes normally** (i.e., not terminated by a `break` statement).

Using `else` with `for` Loops

Example Without `break`

```
for num in range(3):
    print(num)
else:
    print("Loop completed without break.")
```

Output:

```
0
1
2
Loop completed without break.
```

Example With `break`

```
for num in range(5):
    print(num)
    if num == 2:
        break
else:
    print("Loop completed without break.")
```

Output:

```
0
1
2
```

Explanation:

• Since the loop is terminated by `break`, the `else` block is **not** executed.

Using `else` with `while` Loops

Example Without `break`

```
count = 0

while count < 3:
    print(count)
    count += 1
else:
    print("While loop completed without break.")
```

Output:

```
0
1
2
While loop completed without break.
```

Example With `break`

```
count = 0

while count < 5:
    print(count)
    if count == 3:
        break
    count += 1
else:
```

```
    print("While loop completed without break.")
```

Output:

```
0
1
2
3
```

Explanation:

• The loop is terminated by `break`, so the `else` block is **not** executed.

Best Practices

• **Choose the Right Loop Type**:

    • Use `for` loops when iterating over a sequence or when the number of iterations is known.

    • Use `while` loops when the number of iterations is not known in advance and depends on a condition.

• **Avoid Infinite Loops**:

    ◇ Ensure that the loops condition will eventually

become `False`.

◇ If using `while True`, make sure there is a `break` statement within the loop.

Use `enumerate()` and `zip()` When Appropriate:

• `enumerate()` provides both the index and the value when iterating.

• `zip()` allows parallel iteration over multiple sequences.

Example Using `enumerate()`:

```
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits):
    print(f"Fruit {index}: {fruit}")
```

Output:

```
Fruit 0: apple
Fruit 1: banana
Fruit 2: cherry
```

Example Using `zip()`:

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 22, 23]
```

```
for name, age in zip(names, ages):
    print(f"{name} is {age} years old.")
```

Output:

```
Alice is 25 years old.
Bob is 22 years old.
Charlie is 23 years old.
```

Examples and Exercises

Example 1: Printing Numbers from 1 to 5

Using `for` Loop

```
for num in range(1, 6):
    print(num)
```

Using `while` Loop

```
count = 1

while count <= 5:
```

```
    print(count)
    count += 1
```

Output:

```
1
2
3
4
5
```

Example 2: Summing Numbers in a List

Using `for` Loop

```
numbers = [1, 2, 3, 4, 5]
total = 0

for num in numbers:
    total += num

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

Using `while` Loop

```
numbers = [1, 2, 3, 4, 5]
total = 0
```

```
index = 0

while index < len(numbers):
    total += numbers[index]
    index += 1

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

Output:
Total: 15

Example 3: **Finding a Specific Element**

Using `for` Loop with `break`

```
fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:
    if fruit == "cherry":
        print("Cherry found!")
        break
else:
    print("Cherry not found.")
```

Output:
Cherry found!

Exercise 4: Multiplication Table

Task: Write a program that prints the multiplication table for numbers 1 through 5.

Solution Using `for` Loop

```
for i in range(1, 6):
    for j in range(1, 11):
        product = i * j
        print(f"{i} x {j} = {product}")
    print("-" * 15)  # Separator after each table
```

Output:

```
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
---------------
2 x 1 = 2
...
5 x 10 = 50
---------------
```

Exercise 5: Prime Number Checker

```
num = 29
is_prime = True
```

```
if num <= 1:
    is_prime = False
else:
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break

if is_prime:
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

Output:
29 is a prime number.

Exercise 6: FizzBuzz

Task: For numbers from 1 to 15, print "Fizz" for multiples of 3, "Buzz" for multiples of 5, and "FizzBuzz" for multiples of both.

Solution Using `for` Loop and `if-elif-else`

```
for num in range(1, 16):
    if num % 3 == 0 and num % 5 == 0:
```

```
        print("FizzBuzz")
    elif num % 3 == 0:
        print("Fizz")
    elif num % 5 == 0:
        print("Buzz")
    else:
        print(num)
```

Output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

Exercise 7: Factorial Calculator
Task: Calculate the factorial of a given number.

Solution Using `while` Loop

```
num = 5
factorial = 1

if num < 0:
    print("Factorial does not exist for negative numbers.")
elif num == 0:
    print("Factorial of 0 is 1.")
else:
    while num > 0:
        factorial *= num
        num -= 1
    print(f"Factorial is {factorial}")
```

Output:
Factorial is 120

List Comprehensions and

Generator Expressions

In Python, **List Comprehensions** and **Generator Expressions** provide a concise way to create and manipulate lists and generators. They are not only syntactically elegant but also often more efficient than traditional loops.

• **List Comprehensions**: Generate new lists by applying an expression to each item in an existing iterable, optionally filtering items based on a condition.

• **Generator Expressions**: Similar to list comprehensions but generate items one at a time and are more memory-efficient, especially for large datasets.

List Comprehensions

List comprehensions offer a concise way to create lists. They consist of brackets containing an expression followed by a `for` clause, and optionally, one or more `if` clauses.

Basic Syntax
[expression for item in iterable if condition]

• **expression**: The value or transformation to apply to each item.
• **item**: The variable representing each element in the iterable.
• **iterable**: The collection of items to iterate over (e.g., list, tuple, range).
• **condition** (optional): A filter that determines whether the `expression` is applied to the `item`.

Examples

a. Creating a List of Squares

```
squares = [x**2 for x in range(10)]
print(squares)
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

b. Filtering Even Numbers

```
even_numbers = [x for x in range(20) if x % 2 == 0]
print(even_numbers)
```

Output:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

c. Applying a Function to Each Item

```
def square(x):
    return x * x

squared = [square(x) for x in range(5)]
print(squared)
```

Output:

```
[0, 1, 4, 9, 16]
```

d. Flattening a Nested List

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

flattened = [num for row in matrix for num in row]
print(flattened)

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

flattened = [num for row in matrix for num in row]
print(flattened)
```

Output:
[1, 2, 3, 4, 5, 6, 7, 8, 9]

Nested List Comprehensions:
List comprehensions can be nested to handle multi-dimensional data structures.

Example: Transposing a Matrix

List comprehensions can be nested to handle multi-dimensional data structures.

Example: Transposing a Matrix

Order of `for` Clauses:
• In **single list comprehensions** with multiple `for` clauses, the `for` clauses are ordered from **outer to inner**, just like nested `for` loops.

Syntax

```
[expr for x in outer for y in inner]  # x is outer, y is inner
```

• In **nested list comprehensions**, each list comprehension can have its own `for` clauses, with the outer list comprehension controlling the outer loop and the inner list

comprehension controlling the inner loop.

Syntax
[[expr for inner_var in inner_iterable] for outer_var in outer_iterable]

Generator Expressions

Generator expressions are similar to list comprehensions but use parentheses `()` instead of brackets `[]`. They generate items one at a time and are more memory-efficient, making them suitable for large datasets.

Basic Syntax
(expression for item in iterable if condition)

Examples

a. Creating a Generator for Squares

```
squares_gen = (x**2 for x in range(10))
print(squares_gen)
```

Output

```
<generator object <genexpr> at 0x...>
```

To retrieve the items, you can iterate over the generator:

```
for square in squares_gen:
    print(square)
```

Output:

```
0
1
4
9
16
25
36
49
64
81
```

b. Filtering with a Generator Expression

```
even_gen = (x for x in range(20) if x % 2 == 0)
for even in even_gen:
    print(even)
```

Output:
0
2
4
6
8
10
12
14
16
18

c. Using `next()` with Generators
You can manually retrieve items using the `next()` function.
```
squares_gen = (x**2 for x in range(5))
print(next(squares_gen))  # Output: 0
print(next(squares_gen))  # Output: 1
```

Output:
0
1

Note: Once a generator is exhausted, subsequent calls to `next()` will raise a `StopIteration` exception.

d. Memory Efficiency Example

Creating a large list vs. a generator:

```
# List comprehension
large_list = [x for x in range(1000000)]

# Generator expression
large_gen = (x for x in range(1000000))
```

• **List Comprehension**: Allocates memory for the entire list.
• **Generator Expression**: Generates items on-the-fly, using much less memory.

Generator Memory Efficiency

Example: Reading a Large File
Suppose you want to read a large file and process each line

Using List Comprehension:

```
with open('large_file.txt') as file:
    lines = [line.strip() for line in file]
# All lines are stored in memory
```

Using Generator Expression:

```
with open('large_file.txt') as file:
    lines_gen = (line.strip() for line in file)
    for line in lines_gen:
        process(line)  # Process each line one at a time
# Only one line is in memory at a time
```

Exercise

Example 1: List Comprehension with Condition

Task: Create a list of squares for even numbers between 1 and 10.

```
squares_even = [x**2 for x in range(1, 11) if x % 2 == 0]
print(squares_even)
```

Output:

```
[4, 16, 36, 64, 100]
```

Example 2: Generator Expression for Large Dataset

Task: Create a generator that yields squares of numbers from 1 to 1,000,000.

```
squares_gen = (x**2 for x in range(1, 1000001))
# To demonstrate, we'll print the first 5 squares
for _ in range(5):
    print(next(squares_gen))
```

Output:

```
1
4
9
16
25
```

Note: Using a generator here avoids storing a large list in memory.

Example 3: Nested List Comprehension

Task: Create a 3x3 matrix initialized with zeros using a nested list comprehension.

```
rows, cols = 3, 3
matrix = [[0 for _ in range(cols)] for _ in range(rows)]
print(matrix)
```

Output:

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Exercise 4: Filter and Transform

Task: Given a list of numbers, create a new list containing the cubes of numbers that are divisible by 3.
List: `[1, 3, 4, 6, 7, 9, 12]`

Expected Output: `[27, 216, 729, 1728]`

Solution Using List Comprehension
```
numbers = [1, 3, 4, 6, 7, 9, 12]
cubes_div3 = [x**3 for x in numbers if x % 3 == 0]
print(cubes_div3)
```

Exercise 5: Create a Dictionary from Two Lists
Task: Given two lists, one containing names and the other containing ages, create a dictionary that maps each name to its corresponding age.

Names: `["Alice", "Bob", "Charlie"]`
Ages: `[25, 22, 23]`

Expected Output: `{"Alice": 25, "Bob": 22, "Charlie": 23}`

Solution Using Dictionary Comprehension
```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 22, 23]
age_dict = {name: age for name, age in zip(names, ages)}
print(age_dict)
```

Output:
{'Alice': 25, 'Bob': 22, 'Charlie': 23}

Exercise 6: Prime Number Generator

Exercise 7: Prime Number Generator

Task: Create a generator expression that yields prime numbers up to 50.

Solution

Creating a generator expression for prime numbers is more complex because it involves checking each number for primality. Instead, we'll use a generator function with `yield`

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

primes_gen = (x for x in range(2, 51) if is_prime(x))

for prime in primes_gen:
```

```
    print(prime, end=' ')
```

Output:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

Note: While list comprehensions can handle this, using a generator expression is more memory-efficient.

Exercise 8: List Comprehension vs Loop

Task: Compare the performance of list comprehensions and `for` loops by creating a list of squares from 1 to 1,000,000.

Solution

We'll use the `time` module to measure execution time.

```
import time

# Using List Comprehension
start_time = time.time()
squares_list = [x**2 for x in range(1, 1000001)]
end_time = time.time()
print(f"List Comprehension took {end_time - start_time:.4f} seconds.")

# Using For Loop
start_time = time.time()
```

```
squares_loop = []
for x in range(1, 1000001):
    squares_loop.append(x**2)
end_time = time.time()
print(f"For Loop took {end_time - start_time:.4f} seconds.")
```

Sample Output:
List Comprehension took 0.0543 seconds.
For Loop took 0.1521 seconds.

Strings

Strings are one of the most fundamental data types in Python, used extensively for handling and manipulating textual data. Understanding strings and their associated methods is crucial for effective Python programming.

Table of Contents

Introduction to Strings

In Python, a **string** is a sequence of characters enclosed within quotes. Strings are used to represent and manipulate text data.

• **Single Quotes:** `'Hello, World!'`
• **Double Quotes:** `"Hello, World!"`
• **Triple Quotes:** `'''Hello, World!'''` or `"""Hello, World!"""` (used for multi-line strings)

Key Characteristics:
◇ **Immutable:** Once created, the contents of a string cannot be changed.
◇ **Ordered:** Characters in a string have a defined order and can be accessed via indices.
◇ **Iterable:** You can iterate over each character in a string using loops.

String Basics

String Creation

Strings can be created in various ways:

```
# Using single quotes
string1 = 'Hello, World!'
```

```
# Using double quotes
string2 = "Hello, World!"

# Using triple quotes for multi-line strings
string3 = """Hello,
World!"""

# Using the str() constructor
string4 = str(123)  # Converts integer to string '123'

print(string1)
print(string2)
print(string3)
print(string4)
```

Output:

```
Hello, World!
Hello, World!
Hello,
World!
123
```

String Immutability

Strings in Python are **immutable**, meaning that once a string is created, its content cannot be changed. Any

operation that modifies a string actually creates a new string.

```
s = "Hello"
# Attempting to change the first character
# s[0] = 'h'  # This will raise a TypeError

# Correct way: Create a new string
s_new = 'h' + s[1:]
print(s_new)  # Output: "hello"
```

String Indexing and Slicing

Slicing Syntax:
string[start:stop:step]
• **start**: Starting index (inclusive)
• **stop**: Ending index (exclusive)
• **step**: Step size (optional)

You can access individual characters in a string using **indices**. Python uses **zero-based indexing**.

```
s = "Python"

# Positive indices
```

```
print(s[0])  # Output: 'P'
print(s[2])  # Output: 't'

# Negative indices
print(s[-1])  # Output: 'n'
print(s[-3])  # Output: 'h'

# Slicing
print(s[1:4])   # Output: 'yth'
print(s[:3])    # Output: 'Pyt'
print(s[3:])    # Output: 'hon'
print(s[-4:-1]) # Output: 'tho'
```

String Concatenation and Repetition

Concatenation combines two or more strings, while **repetition** repeats a string multiple times.

```
# Concatenation
s1 = "Hello"
s2 = "World"
s3 = s1 + ", " + s2 + "!"
print(s3)  # Output: "Hello, World!"

# Repetition
s4 = "Echo! " * 3
print(s4)  # Output: "Echo! Echo! Echo! "
```

Common String Methods

Python provides a rich set of **string methods** to perform various operations on strings. Below are some of the most useful and commonly used string methods.

and

Convert a string to uppercase or lowercase.

```
s = "Hello, World!"

print(s.upper())  # Output: "HELLO, WORLD!"
print(s.lower())  # Output: "hello, world!"
```

, , and

Remove whitespace or specified characters from the beginning and/or end of a string.

```
s = "   Hello, World!   "

# Remove leading and trailing whitespace
```

```
print(s.strip())   # Output: "Hello, World!"

# Remove leading whitespace
print(s.lstrip())  # Output: "Hello, World!   "

# Remove trailing whitespace
print(s.rstrip())  # Output: "   Hello, World!"

Removing Specific Characters:
s = "###Hello, World!###"

# Remove leading and trailing '#'
print(s.strip('#'))   # Output: "Hello, World!"
```

and

`.split()` divides a string into a list based on a specified separator. `.join()` combines elements of an iterable into a single string with a specified separator.

```
# Using split()
s = "apple,banana,cherry"
fruits = s.split(',')  # Split by comma
print(fruits)  # Output: ['apple', 'banana', 'cherry']

# Using join()
```

```
separator = "-_-"
joined = separator.join(fruits)
print(joined)  # Output: "apple-_-banana-_-cherry"
```

Default Separator:
• If no separator is specified, `.split()` uses any whitespace as the default separator.

```
s = "Hello World! Welcome to Python."
words = s.split()  # Split by whitespace
print(words)  # Output: ['Hello', 'World!', 'Welcome', 'to', 'Python.']
```

.replace()

Replace occurrences of a substring with another substring.

```
s = "Hello, World!"
s_new = s.replace("World", "Python")
print(s_new)  # Output: "Hello, Python!"
```

Replacing Multiple Occurrences:

```
s = "banana"
s_new = s.replace("a", "o")
print(s_new)  # Output: "bonono"
```

Limiting Replacements:

The `.replace()` method can take an optional third argument specifying the maximum number of replacements.

```
s = "apple, apple, apple"
s_new = s.replace("apple", "orange", 2)
print(s_new)  # Output: "orange, orange, apple"
```

and

`.find()` and `.index()` are used to locate the position of a substring within a string. Both return the lowest index where the substring is found.

- `.find()` returns `-1` if the substring is not found.
- `.index()` raises a `ValueError` if the substring is not found.

```
s = "Hello, World!"

# Using find()
position = s.find("World")
print(position)  # Output: 7

# Using index()
position = s.index("World")
print(position)  # Output: 7
```

```
# Substring not found
print(s.find("Python"))  # Output: -1
# print(s.index("Python"))  # Raises ValueError
```

and

Check if a string starts or ends with a specified substring.

```
s = "Hello, World!"
print(s.startswith("Hello"))  # Output: True
print(s.endswith("World!"))   # Output: True
print(s.startswith("World"))  # Output: False
```

Case Sensitivity: These methods are case-sensitive.

```
print(s.startswith("hello"))  # Output: False
```

Count the number of occurrences of a substring within a string.

```
s = "banana"
count = s.count("a")
print(count)  # Output: 3

# Counting overlapping substrings
s = "aaaaa"
count = s.count("aa")
```

```
print(count)  # Output: 2  # 'aa' overlaps, counted separately
```

, , and

`.capitalize()` capitalizes the first character of the string and lowers the rest.

```
s = "hello, world!"
print(s.capitalize())  # Output: "Hello, world!"
```

`.title()` capitalizes the first character of each word

```
s = "hello, world!"
print(s.title())  # Output: "Hello, World!"
```

`.swapcase()` swaps the case of each character.

```
s = "Hello, World!"
print(s.swapcase())  # Output: "hELLO, wORLD!"
```

`.()` and f-strings

`.format()`

Use placeholders `{}` within the string and provide values via `.format()`.

```
name = "Alice"
age = 25
s = "My name is {} and I am {} years old.".format(name, age)
print(s)  # Output: "My name is Alice and I am 25 years old."
```

Specifying Order and Keys:

Order
s = "My name is {0} and I am {1} years old. {0} loves Python.".format(name, age)
print(s) # Output: "My name is Alice and I am 25 years old. Alice loves Python."

Keys:
s = "My name is {name} and I am {age} years old.".format(name="Bob", age=30)print(s) # Output: "My name is Bob and I am 30 years old

f-strings (Formatted String Literals)
Introduced in Python 3.6, f-strings offer a more concise and readable way to embed expressions inside string literals.
name = "Charlie"
age = 23
s = f"My name is {name} and I am {age} years old."
print(s) # Output: "My name is Charlie and I am 23 years old."

Including Expressions:
s = f"Next year, I will be {age + 1} years old."
print(s) # Output: "Next year, I will be 24 years old."

Formatting Numbers:

```
pi = 3.141592653589793
s = f"Pi rounded to 2 decimal places is {pi:.2f}."
print(s)  # Output: "Pi rounded to 2 decimal places is 3.14."
```

Using Methods Inside f-strings:

```
name = "david"
s = f"My name is {name.capitalize()}."
print(s)  # Output: "My name is David."
```

, , , and More

```
s1 = "HelloWorld"
print(s1.isalpha())  # Output: True  # All characters are alphabetic

s2 = "12345"
print(s2.isdigit())  # Output: True  # All characters are digits

s3 = "Hello123"
print(s3.isalnum())  # Output: True  # All characters are alphanumeric

s4 = "Hello World!"
print(s4.isalnum())  # Output: False  # Contains space and punctuation
```

```
# Other similar methods:
# .isspace(), .islower(), .isupper(), .istitle(), etc.

s5 = "   "
print(s5.isspace())  # Output: True

s6 = "hello"
print(s6.islower())  # Output: True

s7 = "HELLO"
print(s7.isupper())  # Output: True

s8 = "Hello"
print(s8.istitle())  # Output: True
```

Exercise

Example 1: Basic String Manipulation

```
s = "  Python Programming  "

# Remove leading and trailing whitespace
s_clean = s.strip()
print(s_clean)  # Output: "Python Programming"
```

```
# Convert to uppercase
s_upper = s_clean.upper()
print(s_upper)  # Output: "PYTHON PROGRAMMING"

# Replace "Programming" with "Language"
s_replaced = s_clean.replace("Programming", "Language")
print(s_replaced)  # Output: "Python Language"

# Split into words
words = s_clean.split()
print(words)  # Output: ['Python', 'Programming']

# Join words with a hyphen
s_joined = '-'.join(words)
print(s_joined)  # Output: "Python-Programming"
```

Example 2: Using and f-strings

```
name = "Diana"
age = 27

# Using .format()
s1 = "My name is {} and I am {} years old.".format(name,
age)
print(s1)  # Output: "My name is Diana and I am 27 years
old."
```

```
# Using f-strings
s2 = f"My name is {name} and I am {age} years old."
print(s2)  # Output: "My name is Diana and I am 27 years old."

# Including expressions in f-strings
s3 = f"In five years, I will be {age + 5} years old."
print(s3)  # Output: "In five years, I will be 32 years old."
```

Example 2: Using and f-strings

```
name = "Diana"
age = 27

# Using .format()
s1 = "My name is {} and I am {} years old.".format(name, age)
print(s1)  # Output: "My name is Diana and I am 27 years old."

# Using f-strings
s2 = f"My name is {name} and I am {age} years old."
print(s2)  # Output: "My name is Diana and I am 27 years old."

# Including expressions in f-strings
s3 = f"In five years, I will be {age + 5} years old."
```

```
print(s3)  # Output: "In five years, I will be 32 years old."
```

Example 3: Checking String Content

```
s = "Hello123"

# Check if all characters are alphanumeric
print(s.isalnum())  # Output: True

# Check if all characters are alphabetic
print(s.isalpha())  # Output: False

# Check if all characters are digits
print(s.isdigit())  # Output: False

# Check if string starts with "He"
print(s.startswith("He"))  # Output: True

# Check if string ends with "123"
print(s.endswith("123"))  # Output: True
```

Exercise 1: Palindrome Checker

Task: Write a program that checks if a given string is a palindrome (reads the same backward as forward).

```
def is_palindrome(s):
    # Remove spaces and convert to lowercase
```

```
    s_clean = ''.join(s.split()).lower()
    return s_clean == s_clean[::-1]

# Test cases
test_strings = ["Racecar", "Hello", "A man a plan a canal
Panama", "Python"]

for string in test_strings:
    if is_palindrome(string):
        print(f"'{string}' is a palindrome.")
    else:
        print(f"'{string}' is not a palindrome.")
```

Expected Output:

```
Expected Output:
'Racecar' is a palindrome.
'Hello' is not a palindrome.
'A man a plan a canal Panama' is a palindrome.
'Python' is not a palindrome.
```

Exercise 2: Sentence Capitalization

Task: Given a sentence, capitalize the first letter of each word.

```
sentence = "python programming is fun!"
```

```
# Using .title()
capitalized = sentence.title()
print(capitalized)  # Output: "Python Programming Is Fun!"

# Using .split() and .join()
words = sentence.split()
capitalized_words = [word.capitalize() for word in words]
capitalized = ' '.join(capitalized_words)
print(capitalized)  # Output: "Python Programming Is Fun!"
```

Exercise 3: Extracting Vowels

Task: Extract all vowels from a given string and return them as a list.

```
def extract_vowels(s):
    vowels = 'aeiouAEIOU'
    return [char for char in s if char in vowels]

s = "Hello, World!"
vowels_list = extract_vowels(s)
print(vowels_list)  # Output: ['e', 'o', 'o']
```

Exercise 4: Reversing Words in a Sentence

Task: Reverse the order of words in a given sentence.

```
sentence = "Python is a powerful programming language."

# Split the sentence into words
words = sentence.split()

# Reverse the list of words
reversed_words = words[::-1]

# Join the reversed words back into a string
reversed_sentence = ' '.join(reversed_words)
print(reversed_sentence)  # Output: "language.
programming powerful a is Python"
```

Exercise 5: Counting Specific Characters

```
def count_e(s):
    return s.lower().count('e')

s = "Eleven elephants entered the estate."
count = count_e(s)
print(f"The letter 'e' appears {count} times.")  # Output: "The
letter 'e' appears 7 times."
```

Exercise 5: Counting Specific Characters

Task: Count the number of times the letter 'e' appears in a string, case-insensitive

```
def count_e(s):
    return s.lower().count('e')

s = "Eleven elephants entered the estate."
count = count_e(s)
print(f"The letter 'e' appears {count} times.")  # Output: "The letter 'e' appears 7 times."
```

Tuples

Introduction to Tuples

A **tuple** is an ordered, immutable collection of items in Python. Unlike lists, tuples cannot be modified after their creation, making them suitable for storing data that should remain constant throughout the program.

Key Characteristics of Tuples:

- **Ordered:** Elements have a defined order, and that order will not change.
- **Immutable:** Once a tuple is created, its elements cannot be altered, added, or removed.
- **Heterogeneous:** Tuples can contain elements of different

data types.
• **Indexed:** Elements can be accessed using indices, starting from 0.
• **Hashable:** Tuples can be used as keys in dictionaries if all their elements are hashable.

Common Use Cases:
• Returning multiple values from a function.
• Storing related but different pieces of data.
• Using as keys in dictionaries.
• Ensuring data integrity by preventing modifications.

Tuple Basics

Tuples can be created in several ways:

Using Parentheses ( )

```
# Empty tuple
empty_tuple = ()

# Tuple with multiple elements
fruits = ("apple", "banana", "cherry")
print(fruits)  # Output: ('apple', 'banana', 'cherry')
```

Without Parentheses (Tuple Packing)

```
# Tuple packing
colors = "red", "green", "blue"
print(colors)  # Output: ('red', 'green', 'blue')
```

Using the `tuple()` Constructor

```
# From a list
numbers = tuple([1, 2, 3, 4])
print(numbers)  # Output: (1, 2, 3, 4)

# From a string
letters = tuple("Python")
print(letters)  # Output: ('P', 'y', 't', 'h', 'o', 'n')
```

Note: Parentheses are optional when defining a tuple; however, they enhance readability, especially in complex expressions.

Tuple Immutability

Tuples are **immutable**, meaning that once a tuple is created, its elements cannot be changed, added, or removed.

```
s = (1, 2, 3)
# Attempting to modify an element
```

```
# s[0] = 10  # This will raise a TypeError

# Attempting to add an element
# s.append(4)  # AttributeError: 'tuple' object has no
attribute 'append'

# Attempting to remove an element
# del s[1]  # TypeError: 'tuple' object doesn't support item
deletion
```

Why Use Immutable Tuples?

- **Data Integrity:** Ensures that the data remains constant throughout the program.
- **Hashable:** Allows tuples to be used as keys in dictionaries and elements in sets.
- **Performance:** Slightly faster than lists due to immutability.

Tuple Indexing and Slicing

Like lists, tuples support **indexing** and **slicing** to access their elements.

Slicing Syntax:

- **start:** Starting index (inclusive).
- **stop:** Ending index (exclusive).
- **step:** Step size (optional).

```
s = ("a", "b", "c", "d", "e")

# Indexing
print(s[0])   # Output: 'a'
print(s[-1])  # Output: 'e'

# Slicing
print(s[1:4])   # Output: ('b', 'c', 'd')
print(s[:3])    # Output: ('a', 'b', 'c')
print(s[2:])    # Output: ('c', 'd', 'e')
print(s[::2])   # Output: ('a', 'c', 'e')
```

Tuple Concatenation and Repetition

Tuples support **concatenation** using the + operator and **repetition** using the * operator.

```
t1 = (1, 2, 3)
t2 = (4, 5)

# Concatenation
t3 = t1 + t2
print(t3)  # Output: (1, 2, 3, 4, 5)

# Repetition
```

```
t4 = ("repeat",) * 3
print(t4)  # Output: ('repeat', 'repeat', 'repeat')
```

Important Note on Repetition:

• To create a single-element tuple, include a comma:

```
single = ("single",)
print(single)  # Output: ('single',)
```

Without the comma, it's just a string within parentheses:

```
not_a_tuple = ("single")
print(not_a_tuple)  # Output: 'single'
```

Single-Element Tuples

Creating a tuple with a single element requires a trailing comma to differentiate it from a regular parenthesis-enclosed expression.

```
# Single-element tuple
singleton = (42,)
print(singleton)  # Output: (42,)

# Not a tuple
not_tuple = (42)
print(not_tuple)  # Output: 42
```

Why the Comma?

• The comma is the defining characteristic of a tuple, not the parentheses. Parentheses are used for grouping, while commas indicate tuple elements

Tuple Operations

Tuple Unpacking

Tuple unpacking allows you to assign the elements of a tuple to individual variables in a single statement.

```
# Unpacking a tuple
person = ("Alice", 30, "Engineer")

name, age, profession = person
print(name)       # Output: Alice
print(age)        # Output: 30
print(profession) # Output: Engineer
```

Error Example (Mismatch in Elements):

```
data = (1, 2, 3)
# a, b = data  # ValueError: too many values to unpack (expected 2)
```

Handling Variable Number of Elements:

```
# Using * to capture remaining elements
data = (1, 2, 3, 4, 5)
a, b, *c = data
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: [3, 4, 5]
```

Packing Tuples

Tuple packing is the process of combining multiple values into a single tuple without explicitly using parentheses.

```
# Tuple packing
packed = "x", "y", "z"
print(packed)  # Output: ('x', 'y', 'z')

# Equivalent to
packed = ("x", "y", "z")
print(packed)  # Output: ('x', 'y', 'z')
```

Use Cases:

- Returning multiple values from a function.
- Passing multiple values as a single entity.

Swapping Variables

Tuples make it easy to swap the values of two variables

without needing a temporary variable.

```
a = 10
b = 20

# Swapping using tuple unpacking
a, b = b, a

print(a)  # Output: 20
print(b)  # Output: 10
```

Explanation:
• The right-hand side `(b, a)` creates a tuple `(20, 10)`.
• The left-hand side `a, b` unpacks the tuple, assigning `20` to `a` and `10` to `b`.

Common Tuple Methods

Tuples have a minimal set of built-in methods due to their immutability. Here are the most commonly used ones:

`.count()`

Counts the number of occurrences of a specified value in

the tuple.

```
t = (1, 2, 3, 2, 4, 2, 5)

count_2 = t.count(2)
print(count_2)  # Output: 3

count_6 = t.count(6)
print(count_6)  # Output: 0
```

.index()

Returns the index of the first occurrence of a specified value. Raises a `ValueError` if the value is not found.

```
t = ("apple", "banana", "cherry", "banana")

index_banana = t.index("banana")
print(index_banana)  # Output: 1

# Finding index with start and end parameters
index_banana_second = t.index("banana", 2)
print(index_banana_second)  # Output: 3

# Value not in tuple
# t.index("orange")  # Raises ValueError: tuple.index(x): x not in tuple
```

Handling ValueError:

```
t = (1, 2, 3)
```

```
try:
    t.index(4)
except ValueError:
    print("Value not found in tuple.")  # Output: Value not found in tuple.
```

Nested Tuples

Tuples can contain other tuples (or other data structures) as elements, allowing the creation of multi-dimensional tuples.

Advanced Tuple Concepts

Nested Tuples

Tuples can contain other tuples (or other data structures) as elements, allowing the creation of multi-dimensional tuples.

```
nested = ((1, 2), (3, 4), (5, 6))
```

```
print(nested)         # Output: ((1, 2), (3, 4), (5, 6))
print(nested[0])      # Output: (1, 2)
print(nested[0][1])   # Output: 2
```

Named Tuples

Named Tuples provide a way to define simple classes with named fields, making tuples more readable and accessible.

Advantages:

- Access elements by name instead of index.
- Improves code readability.
- Immutable like regular tuples.

Using the `collections` module:

```
from collections import namedtuple

# Define a named tuple type
Person = namedtuple('Person', ['name', 'age', 'profession'])

# Create instances
alice = Person(name="Alice", age=30, profession="Engineer")
bob = Person("Bob", 25, "Designer")

print(alice)     # Output: Person(name='Alice', age=30,
profession='Engineer')
print(bob.name)   # Output: Bob
```

```
print(bob.age)    # Output: 25
```

Accessing Elements:

```
print(alice.name)      # Output: Alice
print(alice[1])        # Output: 30 (index-based access is also allowed)
```

Extending Named Tuples:

Named tuples can be extended by defining new named tuple types or by using inheritance.

```
# Define a new named tuple type that extends Person
Employee = namedtuple('Employee', Person._fields + ('employee_id',))

# Create an instance
charlie = Employee(name="Charlie", age=28, profession="Developer", employee_id=1001)
print(charlie)          # Output: Employee(name='Charlie', age=28, profession='Developer', employee_id=1001)
print(charlie.employee_id)  # Output: 1001
```

Tuple Comprehensions

Python does **not** support tuple comprehensions in the same way it does list comprehensions. Instead, generator expressions are used for similar purposes, as tuples require all elements to be generated upfront.

However, you can create a tuple from a generator expression using the `tuple()` constructor.

```
# Attempting a tuple comprehension-like syntax (Not valid)
# t = (x for x in range(5))  # This creates a generator, not a tuple

# Correct way to create a tuple from a generator expression
t = tuple(x for x in range(5))
print(t)  # Output: (0, 1, 2, 3, 4)
```

Tuples vs Lists

6. Tuples vs. Lists

Understanding the differences between tuples and lists is crucial for choosing the appropriate data structure based on the use case.

Feature	Tuples	Lists
Syntax	Parentheses `()` or no parentheses	Square brackets `[]`
Mutability	Immutable	Mutable
Use Cases	Fixed collections, dictionary keys	Dynamic collections, collections needing modification
Performance	Slightly faster due to immutability	Slower for certain operations
Methods Available	`.count()`, `.index()`	Numerous methods like `.append()`, `.remove()`, `.pop()`, etc.
Memory Consumption	Generally uses less memory	Generally uses more memory
Hashability	Hashable if all elements are hashable	Not hashable

When to Use Tuples

- **Fixed Data:** When the collection of data should not change.
- **Dictionary Keys:** Tuples can be used as keys in dictionaries, while lists cannot.
- **Data Integrity:** Ensuring that data remains constant throughout the program.
- **Performance:** Slightly faster access and iteration.

When to Use Lists

◇ **Dynamic Data:** When the collection needs to be modified (add, remove, change elements).
◇ **Homogeneous Data:** Often used for collections of similar items.
◇ **Advanced Operations:** When you need to perform operations like sorting, reversing, etc.

Examples and Exercises

Example 1: Returning Multiple Values from a Function

Tuples are commonly used to return multiple values from a function.

```
def get_user_info():
    name = "Eve"
    age = 28
    profession = "Data Scientist"
    return name, age, profession  # Returns a tuple

user_info = get_user_info()
```

```
print(user_info)  # Output: ('Eve', 28, 'Data Scientist')

# Unpacking the tuple
name, age, profession = get_user_info()
print(name)       # Output: Eve
print(age)        # Output: 28
print(profession)  # Output: Data Scientist
```

Example 2: Using Named Tuples for Structured Data

Named tuples enhance the readability and usability of tuples by allowing access via named fields.

```
from collections import namedtuple

# Define the named tuple
Point = namedtuple('Point', ['x', 'y'])

# Create instances
p1 = Point(10, 20)
p2 = Point(x=15, y=25)

print(p1)         # Output: Point(x=10, y=20)
print(p2.x, p2.y)  # Output: 15 25
```

Exercise 1: Finding Unique Elements

Task: Given a tuple of numbers, return a new tuple containing only the unique elements.

List: `(1, 2, 2, 3, 4, 4, 5)`

Expected Output: `(1, 2, 3, 4, 5)`

```
def unique_elements(t):
    return tuple(set(t))

numbers = (1, 2, 2, 3, 4, 4, 5)
unique = unique_elements(numbers)
print(unique)  # Output: (1, 2, 3, 4, 5)
```

Exercise 2: Merging Two Tuples

Task: Merge two tuples into a single tuple.

Tuples:

- `t1 = (1, 2, 3)`
- `t2 = (4, 5, 6)`

Expected Output: `(1, 2, 3, 4, 5, 6)`

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)
```

```
merged = t1 + t2
print(merged)  # Output: (1, 2, 3, 4, 5, 6)
```

Exercise 3: Reversing a Tuple

Task: Reverse the elements of a tuple.
Tuple: `(1, 2, 3, 4, 5)`
Expected Output: `(5, 4, 3, 2, 1)`

```
t = (1, 2, 3, 4, 5)
reversed_t = t[::-1]
print(reversed_t)  # Output: (5, 4, 3, 2, 1)
```

Exercise 4: Checking for an Element in a Tuple

Task: Write a function that checks if a given element exists in a tuple.
Tuple: `("apple", "banana", "cherry")`
Function: `def contains(t, element):`

```
t = ("apple", "banana", "cherry")

def contains(t, element):
    return element in t

print(contains(t, "banana"))  # Output: True
print(contains(t, "grape"))   # Output: False
```

Exercise 5: Counting Vowels in a Tuple of Strings

Task: Given a tuple of strings, count the total number of vowels present.

Tuple: `("hello", "world", "python", "programming")`

Expected Output: `7`

Solution:

```
def count_vowels(t):
    vowels = 'aeiouAEIOU'
    return sum(1 for word in t for char in word if char in vowels)

words = ("hello", "world", "python", "programming")
total_vowels = count_vowels(words)
print(total_vowels)  # Output: 7
```

Sets

Introduction to Sets

A **set** is an unordered collection of unique elements in Python. Sets are mutable, meaning you can add or remove

elements after creation. However, the elements contained within a set must be **hashable** (immutable types like numbers, strings, and tuples).

Key Characteristics of Sets:

• **Unordered:** Sets do not maintain any order. The elements have no index positions.

• **Unique:** All elements in a set are unique; duplicates are automatically removed.

• **Mutable:** You can add or remove elements from a set.

• **Iterable:** You can loop through the elements in a set.

Common Use Cases:

◇ **Membership Testing:** Quickly check if an element exists in a collection.

◇ **Removing Duplicates:** Eliminate duplicate entries from a list.

◇ **Set Operations:** Perform mathematical set operations like union, intersection, and difference.

Set Basics

Creating Sets

There are several ways to create sets in Python:

Using Curly Braces `{}`:

```
# Creating a set with multiple elements
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output: {'banana', 'cherry', 'apple'}

# Creating an empty set (Note: `{}` creates an empty dictionary)
empty_set = set()
print(empty_set)  # Output: set()
```

Using the `set()` Constructor:

```
# From a list
numbers = set([1, 2, 3, 4, 5])
print(numbers)  # Output: {1, 2, 3, 4, 5}

# From a string
letters = set("Python")
print(letters)  # Output: {'n', 'h', 'y', 'P', 't', 'o'}

# From a tuple
t = set((1, 2, 3))
print(t)  # Output: {1, 2, 3}
```

Using Set Comprehensions:

```
# Creating a set of squares from 0 to 9
squares = {x**2 for x in range(10)}
```

```
print(squares)  # Output: {0, 1, 64, 4, 9, 16, 25, 36, 49, 81}
```

Important Notes:

Uniqueness: When creating a set, duplicate elements are automatically removed

```
duplicates = {1, 2, 2, 3, 4, 4, 5}
print(duplicates)  # Output: {1, 2, 3, 4, 5}
```

Immutable Elements: Sets cannot contain mutable elements like lists or dictionaries.

```
# This will raise a TypeError
invalid_set = {1, 2, [3, 4]}
# TypeError: unhashable type: 'list'
```

Set Immutability

While **sets themselves are mutable** (you can add or remove elements), the **elements within a set must be immutable** (hashable).

Mutable Elements (Not Allowed):

```
# Attempting to add a list to a set
s = {1, 2, 3}
s.add([4, 5])  # Raises TypeError: unhashable type: 'list'
```

Immutable Elements (Allowed):

```
# Adding a tuple to a set
s = {1, 2, 3}
s.add((4, 5))
print(s)  # Output: {1, 2, 3, (4, 5)}
```

Set Indexing and Slicing

Sets are unordered, which means they do not support indexing, slicing, or other sequence-like behavior

```
s = {"apple", "banana", "cherry"}

# Attempting to access an element by index
# print(s[0])  # Raises TypeError: 'set' object is not subscriptable

# Iterating through a set
for fruit in s:
    print(fruit)
```

Output:

```
banana
cherry
apple
```

Note: The order of elements when iterating over a set is not

guaranteed and may vary.

Set Concatenation and Repetition
Sets do not support concatenation (+) or repetition (*) operations like lists and strings because they are unordered and contain unique element

```
s1 = {"a", "b", "c"}
s2 = {"d", "e", "f"}

# Concatenation (Not Supported)
# combined = s1 + s2  # Raises TypeError: unsupported operand type(s) for +: 'set' and 'set'

# Repetition (Not Supported)
# repeated = s1 * 2  # Raises TypeError: unsupported operand type(s) for *: 'set' and 'int'
```

Alternative for Union:
Use the `union()` method or the `|` operator to combine sets.

```
# Using union()
combined = s1.union(s2)
print(combined)  # Output: {'a', 'b', 'c', 'd', 'e', 'f'}

# Using | operator
combined = s1 | s2
```

```
print(combined)  # Output: {'a', 'b', 'c', 'd', 'e', 'f'}
```

Single-Element Sets
To create a set with a single element, include a trailing comma inside the curly braces.

```
# Single-element set
singleton = {"apple"}
print(singleton)  # Output: {'apple'}

# Without trailing comma (Not a set)
not_set = {"apple"}
print(not_set)  # Output: {'apple'}

# However, {} is an empty dictionary, not a set
empty = {}
print(type(empty))  # Output: <class 'dict'>

# Correct way for empty set
empty_set = set()
print(type(empty_set))  # Output: <class 'set'>
```

Note: `{}` creates an empty dictionary, not a set. Use `set()` to create an empty set.