Python Handbook

A Comprehensive Guide

Attila Asghari

2025

attilaasghari@gmail.com

python-handbook

Python Handbook: A Comprehensive Guide

By Attila Asghari

2025

Purpose of This Handbook

This handbook is designed to help learners understand Python efficiently, without overwhelming explanations. It serves as a fast reference guide that focuses on essential concepts and syntax. Unlike traditional programming books that dive deep into theory, this handbook aims to be a practical, no-nonsense guide that lets you get straight to coding.

Who Is This Handbook For?

This handbook is for:

- **Beginners** who are just starting with Python.
- **Intermediate learners** who want to reinforce their knowledge.
- **Students** looking for a quick reference while learning Python in school or university.
- **Self-learners** who prefer an efficient and structured way to grasp Python concepts.

How to Use This Handbook

One of the key purposes of this book is flexibility. You don't have to follow it in order—**you can jump between topics based on your needs**. Each section is designed to be self-contained so that you can quickly find what you're looking for without unnecessary distractions.

Acknowledgments

Writing this handbook has been an incredible journey, and I couldn't have done it without the support and encouragement of the people around me. I'd like to take a moment to express my gratitude to those who made this book possible.
First and foremost, I want to thank my amazing girlfriend, **Amanda (Asal) Karimi**, for her unwavering support, patience, and encouragement throughout this process. Her belief in me kept me motivated, even when the challenges seemed overwhelming. Thank you for being my inspiration and my rock.
Finally, thank you to the readers of this book. Your curiosity and eagerness to learn are what drive me to share my knowledge. I hope this book helps you on your Python journey and inspires you to create amazing things.

About the Author

My name is Attila Asghari, and I am a final-year Computer Engineering student. I have a deep interest in Artificial Intelligence (AI), Machine Learning (ML), Embedded Systems, and Data Science. While learning Data Science, I realized that I needed a resource to quickly grasp Python without unnecessary explanations—something like a structured cheat sheet. Since I couldn’t find exactly what I was looking for, I decided to create this handbook.

Website: https://attila.vitren.ir

Website: https://ata.vitren.ir

Email: attilaasghari@gmail.com

Prerequisites

This book assumes no prior programming experience. However, familiarity with basic computer operations (e.g., installing software, navigating files) will be helpful. To get started, you'll need:

- A computer with Python installed (Python 3.x is recommended).
- A text editor or IDE (Integrated Development Environment) for writing and running Python code. Popular options include:
  - **Jupyter Notebook** (used to write this book)
  - **VS Code**
  - **PyCharm**
  - **IDLE** (comes with Python)

If you're new to Python, don't worry! The first section of the book will guide you through the basics step by step.

Tools and Setup

To follow along with the examples in this book, you'll need to set up your Python environment. Here's how to get started:

1. **Install Python**:
   - Download the latest version of Python from the official website: https://www.python.org/downloads/
   - Follow the installation instructions for your operating system.
2. **Install Jupyter Notebook** (optional but recommended):
   - Open a terminal or command prompt and run:
     ```
     pip install notebook
     ```
   - Launch Jupyter Notebook by running:

```
jupyter-notebook
```

3. **Install a Text Editor or IDE**:
   - If you prefer a lightweight editor, install **VS Code** or **Sublime Text**.
   - If you want a full-featured IDE, install **PyCharm**.
4. **Verify Your Setup**:
   - Open a terminal or command prompt and type:

     ```
     python --version
     ```

   - You should see the installed Python version (e.g., Python 3.10.0).

Once your environment is set up, you're ready to start coding!

Links to IDE and Text Editors

Name	Link
Visual Studio Code	https://code.visualstudio.com/
Sublime Text	https://www.sublimetext.com/
PyCharm	https://www.jetbrains.com/pycharm/
Jupyter	https://jupyter.org/

Table of Contents

Beginner Python

1. Introduction to Python

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python is widely used in various fields, including web development, data science, artificial intelligence, automation, and more.

Why Learn Python?

- **Easy to Learn**: Python's syntax is straightforward and resembles English, making it beginner-friendly.
- **Versatile**: It can be used for web development, data analysis, machine learning, automation, and more.
- **Large Community**: Python has a vast community of developers, so finding help or libraries is easy.
- **Cross-Platform**: Python runs on Windows, macOS, Linux, and other platforms.

Installing Python

1. **Download Python**: Visit the official Python website (python.org) and download the latest version for your operating system.
2. **Install Python**: Follow the installation instructions. Make sure to check the box that says **"Add Python to PATH"** during installation.
3. **Verify Installation**: Open a terminal or command prompt and type `python --version` or `python3 --version`. If Python is installed correctly, it will display the version number.

Writing Your First Python Program

Python programs are written in `.py` files. You can use any text editor or an Integrated Development Environment (IDE) like **PyCharm**, **VS Code**, or **Jupyter Notebook**.

Let's write a simple program:

```python
# This is a comment. Comments are ignored by the Python interpreter.
print("Hello, World!")
```

```
Hello, World!
```

- **Explanation**:
  - `print()` is a built-in function that outputs text to the console.
  - `"Hello, World!"` is a string, which is a sequence of characters enclosed in quotes.

Running a Python Program

4. Save the file with a `.py` extension, e.g., `hello.py`.
5. Open a terminal or command prompt.
6. Navigate to the folder where the file is saved.

7. Run the program by typing `python hello.py` or `python3 hello.py`.

Python Syntax Basics

- **Indentation**: Python uses indentation (spaces or tabs) to define blocks of code. Unlike other languages, indentation is mandatory in Python.

```
if 5 > 2:
    print("Five is greater than two!")
```

```
Five is greater than two!
```

- **Case Sensitivity**: Python is case-sensitive, so `myVariable` and `myvariable` are treated as different names.
- **Comments**: Use `#` for single-line comments and `'''` or `"""` for multi-line comments.

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

```
'\nThis is a\nmulti-line comment\n'
```

Python Modes

8. **Interactive Mode**: You can run Python code directly in the terminal by typing `python` or `python3`. This is useful for testing small snippets of code.

   ```
   $ python
   >>> print("Hello, Python!")
   Hello, Python!
   ```

9. **Script Mode**: Save your code in a `.py` file and execute it as a script.

---

2. Variables & Data Types

What Are Variables?

A variable is a named location in memory used to store data. Think of it as a container that holds a value. In Python, you don't need to declare the type of a variable explicitly; Python automatically infers the type based on the value assigned.

Rules for Naming Variables

1. Variable names must start with a letter (a-z, A-Z) or an underscore (`_`).
2. The rest of the name can contain letters, numbers, or underscores.
3. Variable names are case-sensitive (`myVar` and `myvar` are different).
4. Avoid using Python keywords (e.g., `if` , `else` , `for` , `while` , etc.) as variable names.

Assigning Values to Variables

Use the `=` operator to assign a value to a variable.

```
x = 10           # x is an integer
name = "Alice"   # name is a string
is_student = True  # is_student is a boolean
```

Data Types in Python

Python has several built-in data types. The most common ones are:

1. **Integers (`int`)**: Whole numbers, positive or negative.

   ```
   age = 25
   ```

2. **Floats (`float`)**: Decimal numbers.

   ```
   height = 5.9
   ```

3. **Strings (`str`)**: Sequences of characters enclosed in single or double quotes.

   ```
   name = "Alice"
   greeting = 'Hello, World!'
   ```

4. **Booleans (`bool`)**: Represents `True` or `False` .

   ```
   is_student = True
   is_working = False
   ```

5. **Lists (`list`)**: Ordered, mutable collections of items.

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

6. **Tuples (`tuple`)**: Ordered, immutable collections of items.

```
coordinates = (10.0, 20.0)
```

7. **Dictionaries (`dict`)**: Unordered collections of key-value pairs.

```
person = {"name": "Alice", "age": 25}
```

8. **Sets (`set`)**: Unordered collections of unique items.

```
unique_numbers = {1, 2, 3, 4}
```

Dynamic Typing

Python is dynamically typed, meaning you don't need to declare the type of a variable. The type is determined at runtime based on the value assigned.

```
x = 10        # x is an integer
x = "Hello"   # Now x is a string
```

Checking Data Types

You can use the `type()` function to check the data type of a variable.

```
x = 10
print(type(x))  # Output: <class 'int'>

y = "Python"
print(type(y))  # Output: <class 'str'>
```

```
<class 'int'>
<class 'str'>
```

Type Conversion

You can convert one data type to another using built-in functions like `int()` , `float()` , `str()` , etc.

```
x = 10
y = float(x)  # Convert integer to float
print(y)      # Output: 10.0

z = str(x)    # Convert integer to string
print(z)      # Output: "10"
```

```
10.0
10
```

Variable Naming Conventions

- Use descriptive names (e.g., `user_age` instead of `x`).
- Use lowercase letters and underscores for variable names (snake_case).
- Avoid single-letter names unless they're used in a small scope (e.g., loop counters).

Example Program

```python
# Variables and Data Types
name = "Alice"
age = 25
height = 5.9
is_student = True

# Displaying values and types
print("Name:", name, type(name))
print("Age:", age, type(age))
print("Height:", height, type(height))
print("Is Student:", is_student, type(is_student))
```

```
Name: Alice <class 'str'>
Age: 25 <class 'int'>
Height: 5.9 <class 'float'>
Is Student: True <class 'bool'>
```

---

3. Working With Strings

A **string** is a sequence of characters enclosed in single (`'`) or double (`"`) quotes. Strings are one of the most commonly used data types in Python, and Python provides many built-in methods to work with them.

Creating Strings

You can create strings using single or double quotes:

```python
string1 = "Hello, World!"
string2 = 'Python is fun!'
```

If your string contains a single quote, use double quotes, and vice versa:

```
string3 = "It's a beautiful day."
string4 = 'He said, "Python is awesome!"'
```

For multi-line strings, use triple quotes (`'''` or `"""`):

```
multi_line_string = """This is a
multi-line
string."""
```

String Operations

1. **Concatenation**: Combine strings using the `+` operator.

```
    first_name = "Alice"
    last_name = "Smith"
    full_name = first_name + " " + last_name
    print(full_name)  # Output: Alice Smith
```

```
Alice Smith
```

2. **Repetition**: Repeat a string using the `*` operator.

```
    laugh = "Ha"
    print(laugh * 3)  # Output: HaHaHa
```

```
HaHaHa
```

3. **Length**: Use the `len()` function to get the length of a string.

```
    text = "Python"
    print(len(text))  # Output: 6
```

```
6
```

4. **Indexing**: Access individual characters in a string using their index. Python uses **zero-based indexing**.

```python
text = "Python"
print(text[0])  # Output: P
print(text[3])  # Output: h
```

```
P
h
```

Negative indexing starts from the end:

```python
print(text[-1])  # Output: n
print(text[-2])  # Output: o
```

```
n
o
```

5. **Slicing**: Extract a substring using slicing. The syntax is `[start:end:step]`.

```python
text = "Python Programming"
print(text[0:6])   # Output: Python
print(text[7:18])  # Output: Programming
print(text[:6])    # Output: Python (from start to index 5)
print(text[7:])    # Output: Programming (from index 7 to end)
print(text[::2])   # Output: Pto rgamn (every second character)
```

```
Python
Programming
Python
Programming
Pto rgamn
```

String Methods

Python provides many built-in methods to manipulate strings. Here are some commonly used ones:

6. `upper()`: Converts the string to uppercase.

```python
text = "Python"
print(text.upper())  # Output: PYTHON
```

```
PYTHON
```

7. `lower()` : Converts the string to lowercase.

```python
text = "Python"
print(text.lower())  # Output: python
```

```
python
```

8. `strip()` : Removes leading and trailing whitespace.

```python
text = "  Python  "
print(text.strip())  # Output: Python
```

```
Python
```

9. `replace()` : Replaces a substring with another substring.

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

```
Hello, Python!
```

10. `split()` : Splits the string into a list of substrings based on a delimiter.

```python
text = "Python is fun"
print(text.split(" "))  # Output: ['Python', 'is', 'fun']
```

```
['Python', 'is', 'fun']
```

11. `find()` : Returns the index of the first occurrence of a substring. Returns `-1` if not found.

```python
text = "Python is fun"
print(text.find("is"))  # Output: 7
```

```
7
```

12. `count()` : Counts the number of occurrences of a substring.

```python
text = "Python is fun and Python is easy"
print(text.count("Python"))  # Output: 2
```

```
2
```

13. `startswith()` and `endswith()`: Checks if a string starts or ends with a specific substring.

```python
text = "Python is fun"
print(text.startswith("Python"))  # Output: True
print(text.endswith("fun"))       # Output: True
```

```
True
True
```

String Formatting

14. **Using `f-strings` (Python 3.6+)**: Embed expressions inside string literals.

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

```
My name is Alice and I am 25 years old.
```

15. **Using `format()`**: Insert values into placeholders `{}`.

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

```
My name is Alice and I am 25 years old.
```

16. **Using `%` (older style)**:

```python
name = "Alice"
age = 25
```

```python
    print("My name is %s and I am %d years old." % (name, age))
    # Output: My name is Alice and I am 25 years old.
```

```
My name is Alice and I am 25 years old.
```

Escape Characters

Escape characters are used to include special characters in strings:

- `\n` : Newline
- `\t` : Tab
- `\\` : Backslash
- `\"` : Double quote
- `\'` : Single quote

Example:

```python
print("Hello,\nWorld!")  # Output: Hello,
                         #         World!
```

```
Hello,
World!
```

Example Program

```python
# Working with Strings
text = "Python is fun!"

# String operations
print("Length:", len(text))
print("First character:", text[0])
print("Last character:", text[-1])
print("Substring:", text[0:6])

# String methods
print("Uppercase:", text.upper())
print("Lowercase:", text.lower())
print("Replace 'fun' with 'awesome':", text.replace("fun", "awesome"))

# String formatting
name = "Alice"
```

```
age = 25
print(f"My name is {name} and I am {age} years old.")
```

```
Length: 14
First character: P
Last character: !
Substring: Python
Uppercase: PYTHON IS FUN!
Lowercase: python is fun!
Replace 'fun' with 'awesome': Python is awesome!
My name is Alice and I am 25 years old.
```

---

4. Working With Numbers

Python supports various types of numbers, including integers, floats, and complex numbers. In this section, we'll focus on **integers** and **floats**, which are the most commonly used numeric types.

Types of Numbers

1. **Integers (`int`)**: Whole numbers, positive or negative, without decimals.

   ```
   x = 10
   y = -5
   ```

2. **Floats (`float`)**: Numbers with decimal points.

   ```
   pi = 3.14
   temperature = -10.5
   ```

3. **Complex Numbers**: Numbers with a real and imaginary part (e.g., `3 + 4j`). We won't cover these in detail here.

Basic Arithmetic Operations

Python supports the following arithmetic operations:

- Addition (`+`)
- Subtraction (`-`)

- Multiplication (`*`)
- Division (`/`)
- Floor Division (`//`)
- Modulus (`%`)
- Exponentiation (`**`)

```
a = 10
b = 3

print(a + b)   # Output: 13 (Addition)
print(a - b)   # Output: 7  (Subtraction)
print(a * b)   # Output: 30 (Multiplication)
print(a / b)   # Output: 3.333... (Division)
print(a // b)  # Output: 3  (Floor Division - rounds down to the nearest
integer)
print(a % b)   # Output: 1  (Modulus - remainder of division)
print(a ** b)  # Output: 1000 (Exponentiation - a raised to the power of b)
```

```
13
7
30
3.3333333333333335
3
1
1000
```

Order of Operations (PEMDAS/BODMAS)

Python follows the standard mathematical order of operations:
4. **P**arentheses
5. **E**xponents
6. **M**ultiplication and **D**ivision (from left to right)
7. **A**ddition and **S**ubtraction (from left to right)

Example:

```
result = 10 + 3 * 2 ** 2  # 2 ** 2 = 4 → 3 * 4 = 12 → 10 + 12 = 22
print(result)  # Output: 22
```

```
22
```

You can use parentheses to change the order of operations:

```
result = (10 + 3) * 2 ** 2  # 10 + 3 = 13 → 2 ** 2 = 4 → 13 * 4 = 52
print(result)  # Output: 52
```

```
52
```

Type Conversion Between Numbers

You can convert between integers and floats using the `int()` and `float()` functions:

```
x = 10
y = 3.14

# Convert float to integer
print(int(y))  # Output: 3 (truncates the decimal part)

# Convert integer to float
print(float(x))  # Output: 10.0
```

```
3
10.0
```

Common Math Functions

Python provides a built-in `math` module for advanced mathematical operations. To use it, you need to import the module:

```
import math
```

Here are some commonly used functions:

8. `math.sqrt()` : Square root.

```
   print(math.sqrt(16))  # Output: 4.0
```

```
4.0
```

9. `math.pow()` : Exponentiation.

```
    print(math.pow(2, 3))  # Output: 8.0 (2 raised to the power of 3)
```

```
8.0
```

10. `math.floor()` : Rounds a number down to the nearest integer.

```
    print(math.floor(3.7))  # Output: 3
```

```
3
```

11. `math.ceil()` : Rounds a number up to the nearest integer.

```
    print(math.ceil(3.2))  # Output: 4
```

```
4
```

12. `math.fabs()` : Absolute value.

```
    print(math.fabs(-10))  # Output: 10.0
```

```
10.0
```

13. `math.pi` **and** `math.e` : Constants for π and e.

```
    print(math.pi)  # Output: 3.141592653589793
    print(math.e)   # Output: 2.718281828459045
```

```
3.141592653589793
2.718281828459045
```

Handling Large Numbers

Python can handle very large integers without any issues:

```
large_number = 123456789012345678901234567890
print(large_number)  # Output: 123456789012345678901234567890
```

```
123456789012345678901234567890
```

For very large floats, you can use scientific notation:

```python
scientific_number = 1.23e6  # 1.23 * 10^6
print(scientific_number)  # Output: 1230000.0
```

```
1230000.0
```

Example Program

```python
# Working with Numbers
a = 10
b = 3

# Arithmetic operations
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)

# Order of operations
result = (a + b) * 2 ** 2
print("Result of (a + b) * 2 ** 2:", result)

# Type conversion
x = 3.14
print("Convert float to int:", int(x))
print("Convert int to float:", float(a))

# Math module
import math
print("Square root of 16:", math.sqrt(16))
print("2 raised to the power of 3:", math.pow(2, 3))
print("Floor of 3.7:", math.floor(3.7))
print("Ceiling of 3.2:", math.ceil(3.2))
print("Absolute value of -10:", math.fabs(-10))
print("Value of pi:", math.pi)
```

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Exponentiation: 1000
Result of (a + b) * 2 ** 2: 52
Convert float to int: 3
Convert int to float: 10.0
Square root of 16: 4.0
2 raised to the power of 3: 8.0
Floor of 3.7: 3
Ceiling of 3.2: 4
Absolute value of -10: 10.0
Value of pi: 3.141592653589793
```

---

5. Getting Input From Users

In Python, you can interact with users by taking input from them using the `input()` function. This allows your program to dynamically respond to user-provided data.

The `input()` Function

The `input()` function reads a line of text from the user and returns it as a **string**. You can optionally provide a prompt to guide the user.

Syntax:

```
input(prompt)
```

- `prompt` : A string that is displayed to the user before they enter their input (optional).

Example:

```
name = input("Enter your name:  ")
print("Hello,", name)
```

```
Enter your name:  Attila
```

```
Hello, Attila
```

- When you run this program, it will display `"Enter your name: "` and wait for the user to type something. After the user presses Enter, the input is stored in the variable `name`.

Important Notes About `input()`

1. **Input is Always a String**: The `input()` function always returns the user's input as a string, even if the user enters a number.

```python
age = input("Enter your age: ")
print(type(age))  # Output: <class 'str'>
```

```
Enter your age:  23


<class 'str'>
```

2. **Converting Input to Other Data Types**: If you need the input as a number (e.g., integer or float), you must explicitly convert it using `int()` or `float()`.

```python
age = int(input("Enter your age: "))
print(type(age))  # Output: <class 'int'>
```

```
Enter your age:  23


<class 'int'>
```

Be careful when converting input, as invalid input (e.g., entering text when a number is expected) will cause an error. We'll cover error handling later.

Example: Simple Calculator

Let's create a simple program that takes two numbers from the user and performs basic arithmetic operations:

```python
# Simple Calculator
num1 = float(input("Enter the first number: "))
```

```python
num2 = float(input("Enter the second number: "))

print("Addition:", num1 + num2)
print("Subtraction:", num1 - num2)
print("Multiplication:", num1 * num2)
print("Division:", num1 / num2)
```

```
Enter the first number:  19
Enter the second number:  91


Addition: 110.0
Subtraction: -72.0
Multiplication: 1729.0
Division: 0.2087912087912088
```

Handling Multiple Inputs

If you want the user to enter multiple values at once, you can use the `split()` method to separate the input into a list of strings. Then, convert them to the desired data type.

Example:

```python
# Taking multiple inputs
values = input("Enter two numbers separated by a space: ").split()
num1 = float(values[0])
num2 = float(values[1])

print("Sum:", num1 + num2)
```

```
Enter two numbers separated by a space:  10 20


Sum: 30.0
```

- If the user enters `10 20`, the program will output `Sum: 30`.

Example: User Registration

Here’s a program that collects user information and displays it back:

```python
# User Registration
name = input("Enter your name: ")
age = int(input("Enter your age: "))
email = input("Enter your email: ")

print("\nUser Details:")
print(f"Name: {name}")
print(f"Age: {age}")
print(f"Email: {email}")
```

```
Enter your name:  attila
Enter your age:  23
Enter your email:  attilaasghari@gmail.com



User Details:
Name: attila
Age: 23
Email: attilaasghari@gmail.com
```

Error Handling for User Input

When converting user input to numbers, invalid input (e.g., entering text instead of a number) will cause a `ValueError`. We'll cover error handling in detail later, but here's a basic example using a `try-except` block:

```python
try:
    age = int(input("Enter your age: "))
    print("Your age is:", age)
except ValueError:
    print("Invalid input! Please enter a valid number.")
```

```
Enter your age:  hello


Invalid input! Please enter a valid number.
```

Example Program

Here's a complete example that combines everything we've learned:

```
# Getting Input From Users
name = input("Enter your name: ")
age = int(input("Enter your age: "))
height = float(input("Enter your height in meters: "))

print("\nUser Profile:")
print(f"Name: {name}")
print(f"Age: {age}")
print(f"Height: {height} meters")

# Simple calculation
birth_year = 2023 - age
print(f"You were born in {birth_year}.")
```

```
Enter your name:  attila
Enter your age:  23
Enter your height in meters:  1.85



User Profile:
Name: attila
Age: 23
Height: 1.85 meters
You were born in 2000.
```

---

6. Lists

A **list** is a versatile and widely used data structure in Python. It is an ordered, mutable (changeable) collection of items. Lists can store elements of different data types, including numbers, strings, and even other lists.

Creating Lists

Lists are created by enclosing elements in square brackets `[]`, separated by commas.

Syntax:

```
my_list = [element1, element2, element3]
```

Examples:

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

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

# Mixed data types
mixed_list = [1, "apple", 3.14, True]

# Nested list (list inside a list)
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Accessing List Elements

You can access elements in a list using their **index**. Python uses **zero-based indexing**, meaning the first element has an index of `0`.

Syntax:

```
my_list[index]
```

Examples:

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

print(fruits[0])  # Output: apple
print(fruits[1])  # Output: banana
print(fruits[2])  # Output: cherry
```

```
apple
banana
cherry
```

- **Negative Indexing**: You can also use negative indices to access elements from the end of the list.

```python
  print(fruits[-1])  # Output: cherry (last element)
  print(fruits[-2])  # Output: banana (second last element)
```

```
cherry
banana
```

- **Slicing**: You can extract a sublist using slicing. The syntax is `[start:end:step]`.

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

  print(numbers[2:5])   # Output: [3, 4, 5] (elements from index 2 to 4)
  print(numbers[:4])    # Output: [1, 2, 3, 4] (elements from start to index
3)
  print(numbers[5:])    # Output: [6, 7, 8, 9] (elements from index 5 to
end)
  print(numbers[::2])   # Output: [1, 3, 5, 7, 9] (every second element)
```

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

Modifying Lists

Lists are **mutable**, meaning you can change their elements after creation.

1. **Updating an Element**:

```
   fruits = ["apple", "banana", "cherry"]
   fruits[1] = "blueberry"
   print(fruits)  # Output: ["apple", "blueberry", "cherry"]
```

```
['apple', 'blueberry', 'cherry']
```

2. **Adding Elements**:
   - `append()` : Adds an element to the end of the list.

```
     fruits.append("orange")
     print(fruits)  # Output: ["apple", "blueberry", "cherry", "orange"]
```

```
['apple', 'blueberry', 'cherry', 'orange']
```

- `insert()` : Inserts an element at a specific index.

```
     fruits.insert(1, "mango")
     print(fruits)  # Output: ["apple", "mango", "blueberry", "cherry",
"orange"]
```

```
['apple', 'mango', 'blueberry', 'cherry', 'orange']
```

3. **Removing Elements**:
    - `remove()` : Removes the first occurrence of a specific value.

```
    fruits.remove("blueberry")
    print(fruits)  # Output: ["apple", "mango", "cherry", "orange"]
```

```
['apple', 'mango', 'cherry', 'orange']
```

- `pop()` : Removes and returns the element at a specific index (or the last element if no index is provided).

```
    removed_fruit = fruits.pop(1)
    print(removed_fruit)  # Output: mango
    print(fruits)         # Output: ["apple", "cherry", "orange"]
```

```
mango
['apple', 'cherry', 'orange']
```

- `del` : Deletes an element or a slice of elements.

```
    del fruits[0]
    print(fruits)  # Output: ["cherry", "orange"]
```

```
['cherry', 'orange']
```

4. **Clearing the List**:
    - `clear()` : Removes all elements from the list.

```
    fruits.clear()
    print(fruits)  # Output: []
```

```
[]
```

List Operations

1. **Concatenation**: Combine two lists using the `+` operator.

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

```
[1, 2, 3, 4, 5, 6]
```

2. **Repetition**: Repeat a list using the `*` operator.

```
repeated = [1, 2] * 3
print(repeated)  # Output: [1, 2, 1, 2, 1, 2]
```

```
[1, 2, 1, 2, 1, 2]
```

3. **Length**: Use the `len()` function to get the number of elements in a list.

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

```
5
```

4. **Membership**: Check if an element exists in a list using the `in` keyword.

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)  # Output: True
print("mango" in fruits)   # Output: False
```

```
True
False
```

List Methods

Here are some commonly used list methods:

1. `sort()`: Sorts the list in ascending order (or alphabetically for strings).

```
    numbers = [3, 1, 4, 1, 5, 9]
    numbers.sort()
    print(numbers)  # Output: [1, 1, 3, 4, 5, 9]
```

```
[1, 1, 3, 4, 5, 9]
```

2. `reverse()` : Reverses the order of the list.

```
    numbers.reverse()
    print(numbers)  # Output: [9, 5, 4, 3, 1, 1]
```

```
[9, 5, 4, 3, 1, 1]
```

3. `copy()` : Returns a shallow copy of the list.

```
    new_numbers = numbers.copy()
    print(new_numbers)  # Output: [9, 5, 4, 3, 1, 1]
```

```
[9, 5, 4, 3, 1, 1]
```

4. `index()` : Returns the index of the first occurrence of a value.

```
    print(numbers.index(4))  # Output: 2
```

```
2
```

5. `count()` : Returns the number of occurrences of a value.

```
    print(numbers.count(1))  # Output: 2
```

```
2
```

Example Program

```
# Working with Lists
fruits = ["apple", "banana", "cherry"]

# Accessing elements
```

```python
print("First fruit:", fruits[0])
print("Last fruit:", fruits[-1])

# Modifying lists
fruits.append("orange")
fruits.insert(1, "mango")
fruits.remove("banana")
removed_fruit = fruits.pop(2)

# List operations
print("Fruits:", fruits)
print("Number of fruits:", len(fruits))
print("Is 'apple' in the list?", "apple" in fruits)

# Sorting and reversing
fruits.sort()
print("Sorted fruits:", fruits)
fruits.reverse()
print("Reversed fruits:", fruits)
```

```
First fruit: apple
Last fruit: cherry
Fruits: ['apple', 'mango', 'orange']
Number of fruits: 3
Is 'apple' in the list? True
Sorted fruits: ['apple', 'mango', 'orange']
Reversed fruits: ['orange', 'mango', 'apple']
```

---

7. List Functions

Python provides a variety of built-in functions and methods to work with lists. These functions allow you to manipulate, analyze, and transform lists efficiently. In this section, we'll explore some of the most commonly used list functions and methods.

Common List Functions

1. `len()` : Returns the number of elements in a list.

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

```
5
```

2. `max()` : Returns the largest element in a list.

```
    print(max(numbers))  # Output: 5
```

```
5
```

3. `min()` : Returns the smallest element in a list.

```
    print(min(numbers))  # Output: 1
```

```
1
```

4. `sum()` : Returns the sum of all elements in a list (only for numeric lists).

```
    print(sum(numbers))  # Output: 15
```

```
15
```

5. `sorted()` : Returns a new sorted list without modifying the original list.

```
    unsorted = [3, 1, 4, 1, 5, 9]
    sorted_list = sorted(unsorted)
    print(sorted_list)  # Output: [1, 1, 3, 4, 5, 9]
    print(unsorted)     # Output: [3, 1, 4, 1, 5, 9] (original list
unchanged)
```

```
[1, 1, 3, 4, 5, 9]
[3, 1, 4, 1, 5, 9]
```

6. `any()` : Returns `True` if at least one element in the list is `True` (or truthy).

```
    boolean_list = [False, True, False]
    print(any(boolean_list))  # Output: True
```

```
True
```

7. `all()` : Returns `True` if all elements in the list are `True` (or truthy).

```
    print(all(boolean_list))  # Output: False
```

```
False
```

Common List Methods

8. `append()` : Adds an element to the end of the list.

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

```
['apple', 'banana', 'cherry']
```

9. `extend()` : Adds all elements of an iterable (e.g., list, tuple) to the end of the list.

```
    fruits.extend(["orange", "mango"])
    print(fruits)  # Output: ["apple", "banana", "cherry", "orange", "mango"]
```

```
['apple', 'banana', 'cherry', 'orange', 'mango']
```

10. `insert()` : Inserts an element at a specific index.

```
    fruits.insert(1, "blueberry")
    print(fruits)  # Output: ["apple", "blueberry", "banana", "cherry",
"orange", "mango"]
```

```
['apple', 'blueberry', 'banana', 'cherry', 'orange', 'mango']
```

11. `remove()` : Removes the first occurrence of a specific value.

```
    fruits.remove("banana")
    print(fruits)  # Output: ["apple", "blueberry", "cherry", "orange",
"mango"]
```

```
['apple', 'blueberry', 'cherry', 'orange', 'mango']
```

12. `pop()` : Removes and returns the element at a specific index (or the last element if no index is provided).

```
    removed_fruit = fruits.pop(2)
    print(removed_fruit)  # Output: cherry
    print(fruits)         # Output: ["apple", "blueberry", "orange", "mango"]
```

```
cherry
['apple', 'blueberry', 'orange', 'mango']
```

13. `clear()` : Removes all elements from the list.

```
    fruits.clear()
    print(fruits)  # Output: []
```

```
[]
```

14. `index()` : Returns the index of the first occurrence of a value.

```
    numbers = [10, 20, 30, 20, 40]
    print(numbers.index(20))  # Output: 1
```

```
1
```

15. `count()` : Returns the number of occurrences of a value.

```
    print(numbers.count(20))  # Output: 2
```

```
2
```

16. `sort()` : Sorts the list in place (modifies the original list).

```
    numbers.sort()
    print(numbers)  # Output: [10, 20, 20, 30, 40]
```

```
[10, 20, 20, 30, 40]
```

- You can also sort in descending order:

```
    numbers.sort(reverse=True)
    print(numbers)  # Output: [40, 30, 20, 20, 10]
```

```
[40, 30, 20, 20, 10]
```

17. `reverse()` : Reverses the order of the list in place.

```
    numbers.reverse()
    print(numbers)  # Output: [10, 20, 20, 30, 40]
```

```
[10, 20, 20, 30, 40]
```

18. `copy()` : Returns a shallow copy of the list.

```
    new_numbers = numbers.copy()
    print(new_numbers)  # Output: [10, 20, 20, 30, 40]
```

```
[10, 20, 20, 30, 40]
```

List Comprehensions

List comprehensions provide a concise way to create lists. They are often used to apply an operation to each element in a list or to filter elements.

Syntax:

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

Examples:
19. Create a list of squares:

```
    squares = [x ** 2 for x in range(1, 6)]
    print(squares)  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

20. Filter even numbers:

```
    even_numbers = [x for x in range(10) if x % 2 == 0]
    print(even_numbers)  # Output: [0, 2, 4, 6, 8]
```

```
[0, 2, 4, 6, 8]
```

21. Convert strings to uppercase:

```
    fruits = ["apple", "banana", "cherry"]
    uppercase_fruits = [fruit.upper() for fruit in fruits]
    print(uppercase_fruits)  # Output: ["APPLE", "BANANA", "CHERRY"]
```

```
['APPLE', 'BANANA', 'CHERRY']
```

Example Program

```
# List Functions and Methods
numbers = [3, 1, 4, 1, 5, 9]

# Common functions
print("Length:", len(numbers))
print("Max:", max(numbers))
print("Min:", min(numbers))
print("Sum:", sum(numbers))
print("Sorted:", sorted(numbers))

# Common methods
numbers.append(2)
print("After append:", numbers)

numbers.extend([7, 8])
print("After extend:", numbers)

numbers.insert(2, 10)
print("After insert:", numbers)

numbers.remove(1)
print("After remove:", numbers)

popped = numbers.pop(3)
print("Popped element:", popped)
print("After pop:", numbers)

print("Index of 5:", numbers.index(5))
print("Count of 1:", numbers.count(1))

numbers.sort()
print("Sorted list:", numbers)

numbers.reverse()
print("Reversed list:", numbers)

# List comprehension
squares = [x ** 2 for x in numbers]
print("Squares:", squares)
```

```
Length: 6
Max: 9
Min: 1
Sum: 23
Sorted: [1, 1, 3, 4, 5, 9]
After append: [3, 1, 4, 1, 5, 9, 2]
After extend: [3, 1, 4, 1, 5, 9, 2, 7, 8]
After insert: [3, 1, 10, 4, 1, 5, 9, 2, 7, 8]
After remove: [3, 10, 4, 1, 5, 9, 2, 7, 8]
Popped element: 1
After pop: [3, 10, 4, 5, 9, 2, 7, 8]
Index of 5: 3
Count of 1: 0
Sorted list: [2, 3, 4, 5, 7, 8, 9, 10]
Reversed list: [10, 9, 8, 7, 5, 4, 3, 2]
Squares: [100, 81, 64, 49, 25, 16, 9, 4]
```

---

8. Tuples

A **tuple** is an ordered, immutable (unchangeable) collection of elements. Tuples are similar to lists, but unlike lists, once a tuple is created, its elements cannot be modified, added, or removed. Tuples are often used for fixed data that shouldn't change, such as coordinates, dates, or configurations.

Creating Tuples

Tuples are created by enclosing elements in parentheses `()`, separated by commas. If a tuple has only one element, you must include a trailing comma to distinguish it from a regular value.

Syntax:

```
my_tuple = (element1, element2, element3)
```

Examples:

```
# Tuple of integers
numbers = (1, 2, 3, 4, 5)

# Tuple of strings
fruits = ("apple", "banana", "cherry")

# Mixed data types
mixed_tuple = (1, "apple", 3.14, True)

# Single-element tuple
single_element = (42,)  # Note the trailing comma
```

- Without the trailing comma, Python will treat it as a regular value:

```
  not_a_tuple = (42)  # This is an integer, not a tuple
```

Accessing Tuple Elements

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

1. **Indexing**: Access elements using their index (zero-based).

```
   fruits = ("apple", "banana", "cherry")
   print(fruits[0])  # Output: apple
   print(fruits[2])  # Output: cherry
```

```
apple
cherry
```

2. **Negative Indexing**: Access elements from the end of the tuple.

```
   print(fruits[-1])  # Output: cherry (last element)
   print(fruits[-2])  # Output: banana (second last element)
```

```
cherry
banana
```

3. **Slicing**: Extract a subtuple using slicing.

```
   numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)
   print(numbers[2:5])   # Output: (3, 4, 5) (elements from index 2 to 4)
   print(numbers[:4])    # Output: (1, 2, 3, 4) (elements from start to
index 3)
   print(numbers[5:])    # Output: (6, 7, 8, 9) (elements from index 5 to
end)
   print(numbers[::2])   # Output: (1, 3, 5, 7, 9) (every second element)
```

```
(3, 4, 5)
(1, 2, 3, 4)
(6, 7, 8, 9)
(1, 3, 5, 7, 9)
```

Tuples Are Immutable

Unlike lists, tuples cannot be modified after creation. This means you cannot:

- Add or remove elements.
- Change existing elements.

Example:

```
fruits = ("apple", "banana", "cherry")
# fruits[1] = "blueberry"  # This will raise a TypeError
```

Tuple Operations

4. **Concatenation**: Combine two tuples using the `+` operator.

```
   tuple1 = (1, 2, 3)
   tuple2 = (4, 5, 6)
   combined = tuple1 + tuple2
   print(combined)  # Output: (1, 2, 3, 4, 5, 6)
```

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

5. **Repetition**: Repeat a tuple using the `*` operator.

```
   repeated = (1, 2) * 3
```

```
    print(repeated)  # Output: (1, 2, 1, 2, 1, 2)
```

```
(1, 2, 1, 2, 1, 2)
```

6. **Membership**: Check if an element exists in a tuple using the `in` keyword.

```
    fruits = ("apple", "banana", "cherry")
    print("banana" in fruits)  # Output: True
    print("mango" in fruits)   # Output: False
```

```
True
False
```

7. **Length**: Use the `len()` function to get the number of elements in a tuple.

```
    print(len(fruits))  # Output: 3
```

```
3
```

Tuple Methods

Since tuples are immutable, they have fewer methods compared to lists. The most commonly used methods are:

8. **`count()`**: Returns the number of occurrences of a value.

```
    numbers = (1, 2, 3, 1, 2, 1)
    print(numbers.count(1))  # Output: 3
```

```
3
```

9. **`index()`**: Returns the index of the first occurrence of a value.

```
    print(numbers.index(2))  # Output: 1
```

```
1
```

When to Use Tuples

- Use tuples when you want to ensure the data remains constant and cannot be modified.
- Tuples are faster than lists for fixed data because of their immutability.
- Use tuples as keys in dictionaries (since lists cannot be used as keys due to their mutability).

Example:

```
# Using a tuple as a dictionary key
location = {
    (40.7128, -74.0060): "New York",
    (34.0522, -118.2437): "Los Angeles"
}
print(location[(40.7128, -74.0060)])  # Output: New York
```

```
New York
```

Unpacking Tuples

You can unpack a tuple into multiple variables. This is useful for assigning values from a tuple to individual variables.

Example:

```
coordinates = (10.0, 20.0)
x, y = coordinates
print("x:", x)  # Output: x: 10.0
print("y:", y)  # Output: y: 20.0
```

```
x: 10.0
y: 20.0
```

Example Program

```
# Working with Tuples
fruits = ("apple", "banana", "cherry")

# Accessing elements
print("First fruit:", fruits[0])
print("Last fruit:", fruits[-1])
```

```python
# Slicing
print("First two fruits:", fruits[:2])

# Tuple operations
numbers = (1, 2, 3)
repeated = numbers * 2
print("Repeated tuple:", repeated)

# Tuple methods
print("Count of 'banana':", fruits.count("banana"))
print("Index of 'cherry':", fruits.index("cherry"))

# Unpacking
x, y, z = fruits
print("Unpacked values:", x, y, z)

# Using tuples as dictionary keys
location = {
    (40.7128, -74.0060): "New York",
    (34.0522, -118.2437): "Los Angeles"
}
print("Location:", location[(40.7128, -74.0060)])
```

```
First fruit: apple
Last fruit: cherry
First two fruits: ('apple', 'banana')
Repeated tuple: (1, 2, 3, 1, 2, 3)
Count of 'banana': 1
Index of 'cherry': 2
Unpacked values: apple banana cherry
Location: New York
```

---

9. Functions

A **function** is a reusable block of code that performs a specific task. Functions help organize code, avoid repetition, and make programs easier to read and maintain. In Python, you can define your own functions using the `def` keyword.

Defining a Function

To define a function, use the `def` keyword followed by the function name, parentheses `()`, and a colon `:`. The code block inside the function is indented.

Syntax:

```python
def function_name(parameters):
    # Code to execute
    return result  # Optional
```

- `function_name` : The name of the function (follows the same rules as variable names).
- `parameters` : Inputs to the function (optional). These are variables that the function uses to perform its task.
- `return` : Specifies the value the function should return (optional). If omitted, the function returns `None` .

Calling a Function

To use a function, you "call" it by writing its name followed by parentheses `()` . If the function has parameters, you pass arguments inside the parentheses.

Example:

```python
# Define a function
def greet():
    print("Hello, World!")

# Call the function
greet()  # Output: Hello, World!
```

```
Hello, World!
```

Function Parameters and Arguments

Parameters are variables listed in the function definition. Arguments are the actual values passed to the function when it is called.

Example:

```python
# Function with parameters
def greet(name):
    print(f"Hello, {name}!")

# Call the function with an argument
```

```python
greet("Alice")  # Output: Hello, Alice!
greet("Bob")    # Output: Hello, Bob!
```

```
Hello, Alice!
Hello, Bob!
```

Returning Values

Use the `return` statement to send a value back to the caller. A function can return any type of data, including numbers, strings, lists, or even other functions.

Example:

```python
# Function that returns a value
def add(a, b):
    return a + b

# Call the function and store the result
result = add(3, 5)
print(result)  # Output: 8
```

```
8
```

- If a function doesn't have a `return` statement, it implicitly returns `None`.

Default Parameters

You can provide default values for parameters. If the caller doesn't pass an argument for that parameter, the default value is used.

Example:

```python
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()          # Output: Hello, Guest!
greet("Alice")   # Output: Hello, Alice!
```

```
Hello, Guest!
Hello, Alice!
```

Keyword Arguments

When calling a function, you can specify arguments by their parameter names. This allows you to pass arguments in any order.

Example:

```python
def describe_pet(pet_name, animal_type="dog"):
    print(f"I have a {animal_type} named {pet_name}.")

# Using keyword arguments
describe_pet(pet_name="Max", animal_type="cat")  # Output: I have a cat named Max.
describe_pet(animal_type="hamster", pet_name="Bella")  # Output: I have a hamster named Bella.
```

```
I have a cat named Max.
I have a hamster named Bella.
```

Variable-Length Arguments

Sometimes, you may not know how many arguments will be passed to a function. Python allows you to handle this using:

- `*args` : Collects additional positional arguments as a tuple.
- `kwargs`**: Collects additional keyword arguments as a dictionary.

Example:

```python
# Using *args
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))  # Output: 6

# Using **kwargs
def describe_pet(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

describe_pet(name="Max", animal_type="dog", age=3)
# Output:
# name: Max
# animal_type: dog
# age: 3
```

```
6
name: Max
animal_type: dog
age: 3
```

Scope of Variables

- **Local Variables**: Variables defined inside a function are local to that function and cannot be accessed outside it.
- **Global Variables**: Variables defined outside all functions are global and can be accessed anywhere in the program.

Example:

```python
x = 10  # Global variable

def my_function():
    y = 5  # Local variable
    print(x)  # Access global variable
    print(y)  # Access local variable

my_function()
print(x)  # Output: 10
# print(y)  # This will raise an error (y is local to my_function)
```

```
10
5
10
```

Lambda Functions

A **lambda function** is a small, anonymous function defined using the `lambda` keyword. It can have any number of arguments but only one expression.

Syntax:

```python
lambda arguments: expression
```

Example:

```
# Lambda function to add two numbers
add = lambda a, b: a + b
print(add(3, 5))  # Output: 8
```

```
8
```

Example Program

```
# Working with Functions

# Define a function
def greet(name="Guest"):
    print(f"Hello, {name}!")

# Call the function
greet()          # Output: Hello, Guest!
greet("Alice")   # Output: Hello, Alice!

# Function with return value
def add(a, b):
    return a + b

result = add(3, 5)
print("Sum:", result)  # Output: Sum: 8

# Function with *args
def multiply(*args):
    product = 1
    for num in args:
        product *= num
    return product

print("Product:", multiply(2, 3, 4))  # Output: Product: 24

# Lambda function
square = lambda x: x ** 2
print("Square of 5:", square(5))  # Output: Square of 5: 25
```

```
Hello, Guest!
Hello, Alice!
Sum: 8
```

```
Product: 24
Square of 5: 25
```

---

10. Return Statement

The `return` **statement** is used in functions to send a value back to the caller. It also terminates the execution of the function, meaning any code after the `return` statement will not be executed.

Purpose of the Return Statement

1. **Return a Value**: The primary purpose of the `return` statement is to return a value (or multiple values) from a function to the caller.
2. **Exit a Function**: The `return` statement immediately exits the function, even if there is code after it.

Syntax

```
def function_name(parameters):
    # Code to execute
    return value  # Value to return
```

- If no value is specified, the function returns `None`.

Returning a Single Value

You can return a single value, such as a number, string, or boolean.

Example:

```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # Output: 8
```

```
8
```

Returning Multiple Values

Python allows you to return multiple values from a function by separating them with commas. These values are returned as a **tuple**.

Example:

```python
def calculate(a, b):
    sum = a + b
    difference = a - b
    product = a * b
    return sum, difference, product

result = calculate(10, 5)
print(result)  # Output: (15, 5, 50)

# Unpack the returned tuple
sum, difference, product = calculate(10, 5)
print("Sum:", sum)           # Output: Sum: 15
print("Difference:", difference)  # Output: Difference: 5
print("Product:", product)  # Output: Product: 50
```

```
(15, 5, 50)
Sum: 15
Difference: 5
Product: 50
```

Returning None

If a function does not have a `return` statement or has a `return` statement without a value, it returns `None`.

Example:

```python
def greet(name):
    print(f"Hello, {name}!")

result = greet("Alice")
print(result)  # Output: None
```

```
Hello, Alice!
None
```

Early Return

You can use the `return` statement to exit a function early based on a condition.

Example:

```python
def is_positive(number):
    if number > 0:
        return True
    return False

print(is_positive(10))  # Output: True
print(is_positive(-5))  # Output: False
```

```
True
False
```

Returning Complex Data Types

You can return complex data types like lists, dictionaries, or even other functions.

Example:

```python
# Returning a list
def get_even_numbers(limit):
    return [x for x in range(limit) if x % 2 == 0]

print(get_even_numbers(10))  # Output: [0, 2, 4, 6, 8]

# Returning a dictionary
def create_person(name, age):
    return {"name": name, "age": age}

print(create_person("Alice", 25))  # Output: {'name': 'Alice', 'age': 25}
```

```
[0, 2, 4, 6, 8]
{'name': 'Alice', 'age': 25}
```

Returning Functions

You can also return a function from another function. This is useful in advanced programming techniques like closures and decorators.

Example:

```python
def create_multiplier(factor):
    def multiplier(number):
        return number * factor
    return multiplier

double = create_multiplier(2)
print(double(5))  # Output: 10
```

```
10
```

Example Program

```python
# Working with the Return Statement

# Function to return a single value
def add(a, b):
    return a + b

print("Sum:", add(3, 5))  # Output: Sum: 8

# Function to return multiple values
def calculate(a, b):
    sum = a + b
    difference = a - b
    product = a * b
    return sum, difference, product

sum, difference, product = calculate(10, 5)
print("Sum:", sum)           # Output: Sum: 15
print("Difference:", difference)  # Output: Difference: 5
print("Product:", product)  # Output: Product: 50

# Function with early return
def is_positive(number):
    if number > 0:
        return True
    return False

print("Is 10 positive?", is_positive(10))  # Output: Is 10 positive? True
print("Is -5 positive?", is_positive(-5))  # Output: Is -5 positive? False
```

```python
# Function returning a list
def get_even_numbers(limit):
    return [x for x in range(limit) if x % 2 == 0]

print("Even numbers up to 10:", get_even_numbers(10))  # Output: [0, 2, 4,
6, 8]

# Function returning a dictionary
def create_person(name, age):
    return {"name": name, "age": age}

print("Person:", create_person("Alice", 25))  # Output: {'name': 'Alice',
'age': 25}

# Function returning another function
def create_multiplier(factor):
    def multiplier(number):
        return number * factor
    return multiplier

double = create_multiplier(2)
print("Double of 5:", double(5))  # Output: Double of 5: 10
```

```
Sum: 8
Sum: 15
Difference: 5
Product: 50
Is 10 positive? True
Is -5 positive? False
Even numbers up to 10: [0, 2, 4, 6, 8]
Person: {'name': 'Alice', 'age': 25}
Double of 5: 10
```

---

11. If Statements

If statements are used to make decisions in your code. They allow you to execute a block of code only if a certain condition is true. If statements are a fundamental part of programming and are used to control the flow of your program.

Syntax of an If Statement

The basic structure of an `if` statement is as follows:

```python
if condition:
    # Code to execute if the condition is true
```

- `condition`: An expression that evaluates to `True` or `False`.
- **Indentation**: The code block under the `if` statement must be indented (usually by 4 spaces).

Example of a Simple If Statement

```python
age = 18

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

```
You are an adult.
```

- If `age` is greater than or equal to 18, the message `"You are an adult."` will be printed. Otherwise, nothing happens.

Adding an Else Clause

The `else` clause is used to execute a block of code when the `if` condition is `False`.

Syntax:

```python
if condition:
    # Code to execute if the condition is true
else:
    # Code to execute if the condition is false
```

Example:

```python
age = 15

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

```
You are a minor.
```

- If `age` is less than 18, the message `"You are a minor."` will be printed.

Using Elif for Multiple Conditions

The `elif` (short for "else if") clause is used to check multiple conditions. It is placed between the `if` and `else` clauses.

Syntax:

```
if condition1:
    # Code to execute if condition1 is true
elif condition2:
    # Code to execute if condition2 is true
else:
    # Code to execute if all conditions are false
```

Example:

```
age = 25

if age < 13:
    print("You are a child.")
elif age < 18:
    print("You are a teenager.")
else:
    print("You are an adult.")
```

```
You are an adult.
```

- This program checks multiple conditions and prints the appropriate message based on the value of `age` .

Nested If Statements

You can nest `if` statements inside other `if` statements to create more complex decision-making logic.

Example:

```python
age = 20
has_license = True

if age >= 18:
    if has_license:
        print("You can drive.")
    else:
        print("You are old enough to drive but don't have a license.")
else:
    print("You are too young to drive.")
```

```
You can drive.
```

Logical Operators in If Statements

You can use logical operators (`and` , `or` , `not`) to combine multiple conditions.

1. `and` : Both conditions must be true.

    ```python
    age = 20
    has_license = True

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

```
You can drive.
```

2. `or` : At least one condition must be true.

    ```python
    age = 16
    has_parental_consent = True

    if age >= 18 or has_parental_consent:
        print("You can participate.")
    ```

```
You can participate.
```

3. `not` : Inverts the condition.

    ```python
    is_raining = False
    ```

```python
    if not is_raining:
        print("Let's go outside!")
```

```
Let's go outside!
```

Truthy and Falsy Values

In Python, conditions are evaluated based on whether they are "truthy" or "falsy":

- **Falsy Values**: `False`, `0`, `""` (empty string), `None`, `[]` (empty list), `{}` (empty dictionary), etc.
- **Truthy Values**: Everything else.

Example:

```python
name = ""

if name:
    print("Hello, " + name)
else:
    print("Name is empty.")
```

```
Name is empty.
```

- Since `name` is an empty string (falsy), the program will print `"Name is empty."`.

Example Program

```python
# Working with If Statements

# Simple If Statement
age = 18
if age >= 18:
    print("You are an adult.")

# If-Else Statement
age = 15
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

```python
# If-Elif-Else Statement
age = 25
if age < 13:
    print("You are a child.")
elif age < 18:
    print("You are a teenager.")
else:
    print("You are an adult.")

# Nested If Statements
age = 20
has_license = True
if age >= 18:
    if has_license:
        print("You can drive.")
    else:
        print("You are old enough to drive but don't have a license.")
else:
    print("You are too young to drive.")

# Logical Operators
age = 20
has_license = True
if age >= 18 and has_license:
    print("You can drive.")

# Truthy and Falsy Values
name = ""
if name:
    print("Hello, " + name)
else:
    print("Name is empty.")
```

```
You are an adult.
You are a minor.
You are an adult.
You can drive.
You can drive.
Name is empty.
```

---

12. If Statements & Comparisons

In Python, **comparison operators** are used to compare values and make decisions in `if` statements. These operators evaluate to `True` or `False`, which determines whether a block of code is executed.

Comparison Operators

Here are the most commonly used comparison operators:

Operator	Description	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`10 > 5`	`True`
`<`	Less than	`10 < 5`	`False`
`>=`	Greater than or equal to	`10 >= 10`	`True`
`<=`	Less than or equal to	`10 <= 5`	`False`

Using Comparisons in If Statements

Comparison operators are often used in `if` statements to make decisions based on the relationship between values.

Example:

```python
x = 10
y = 5

if x > y:
    print("x is greater than y")
else:
    print("x is not greater than y")
```

```
x is greater than y
```

Chaining Comparisons

You can chain multiple comparisons using logical operators (`and` , `or` , `not`) to create more complex conditions.

Example:

```python
x = 10
y = 5
z = 7

if x > y and y < z:
    print("x is greater than y, and y is less than z")
```

```
x is greater than y, and y is less than z
```

Chaining Comparisons

You can chain multiple comparisons using logical operators (`and` , `or` , `not`) to create more complex conditions.

Example:

```python
x = 10
y = 5
z = 7

if x > y and y < z:
    print("x is greater than y, and y is less than z")
```

```
x is greater than y, and y is less than z
```

Comparing Strings

You can also use comparison operators with strings. Strings are compared lexicographically (based on their Unicode values).

Example:

```python
name1 = "Alice"
name2 = "Bob"

if name1 < name2:
    print(f"{name1} comes before {name2} in the dictionary.")
else:
    print(f"{name1} comes after {name2} in the dictionary.")
```

```
Alice comes before Bob in the dictionary.
```

Comparing Lists

Lists can also be compared using comparison operators. Python compares lists element by element.

Example:

```python
list1 = [1, 2, 3]
list2 = [1, 2, 4]

if list1 < list2:
    print("list1 is less than list2")
```

```
list1 is less than list2
```

Using `in` and `not in` for Membership

The `in` and `not in` operators are used to check if a value exists (or does not exist) in a sequence (e.g., list, tuple, string).

Example:

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

if "banana" in fruits:
    print("Banana is in the list.")

if "mango" not in fruits:
    print("Mango is not in the list.")
```

```
Banana is in the list.
Mango is not in the list.
```

Example Program

```python
# Working with If Statements & Comparisons

# Comparing numbers
x = 10
y = 5

if x > y:
    print("x is greater than y")
else:
    print("x is not greater than y")

# Chaining comparisons
z = 7
if x > y and y < z:
    print("x is greater than y, and y is less than z")

# Comparing strings
name1 = "Alice"
name2 = "Bob"

if name1 < name2:
    print(f"{name1} comes before {name2} in the dictionary.")
else:
    print(f"{name1} comes after {name2} in the dictionary.")

# Comparing lists
list1 = [1, 2, 3]
list2 = [1, 2, 4]

if list1 < list2:
    print("list1 is less than list2")

# Membership testing
fruits = ["apple", "banana", "cherry"]

if "banana" in fruits:
    print("Banana is in the list.")

if "mango" not in fruits:
    print("Mango is not in the list.")
```

```
x is greater than y
x is greater than y, and y is less than z
Alice comes before Bob in the dictionary.
list1 is less than list2
Banana is in the list.
Mango is not in the list.
```

13. Dictionaries

A **dictionary** is a collection of key-value pairs. It is an unordered, mutable (changeable), and indexed data structure. Dictionaries are optimized for retrieving values when the key is known. Each key in a dictionary must be unique, and it maps to a specific value.

Creating a Dictionary

Dictionaries are created using curly braces `{}` or the `dict()` constructor. Each key-value pair is separated by a colon `:`, and pairs are separated by commas.

Syntax:

```
my_dict = {
    key1: value1,
    key2: value2,
    key3: value3
}
```

Example:

```
# Dictionary of person details
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Using dict() constructor
person = dict(name="Alice", age=25, city="New York")
```

Accessing Dictionary Values

You can access the value associated with a key using square brackets `[]` or the `get()` method.

Example:

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing values
print(person["name"])  # Output: Alice
print(person.get("age"))  # Output: 25
```

```
Alice
25
```

- If the key does not exist, using `[]` will raise a `KeyError`, while `get()` will return `None` (or a default value you specify).

```
  print(person.get("country", "Unknown"))  # Output: Unknown
```

```
Unknown
```

Adding or Updating Dictionary Entries

You can add a new key-value pair or update an existing one by assigning a value to a key.

Example:

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Adding a new key-value pair
person["country"] = "USA"

# Updating an existing key
person["age"] = 26

print(person)
# Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

```
{'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

Removing Dictionary Entries

You can remove a key-value pair using:

1. `del` : Deletes the key-value pair.

```
    del person["city"]
    print(person)  # Output: {'name': 'Alice', 'age': 26, 'country': 'USA'}
```

```
{'name': 'Alice', 'age': 26, 'country': 'USA'}
```

2. `pop()` : Removes the key-value pair and returns the value.

```
    age = person.pop("age")
    print(age)  # Output: 26
    print(person)  # Output: {'name': 'Alice', 'country': 'USA'}
```

```
26
{'name': 'Alice', 'country': 'USA'}
```

3. `popitem()` : Removes and returns the last inserted key-value pair (Python 3.7+).

```
    last_item = person.popitem()
    print(last_item)  # Output: ('country', 'USA')
    print(person)  # Output: {'name': 'Alice'}
```

```
('country', 'USA')
{'name': 'Alice'}
```

4. `clear()` : Removes all key-value pairs from the dictionary.

```
    person.clear()
    print(person)  # Output: {}
```

```
{}
```

Dictionary Methods

Here are some commonly used dictionary methods:

1. `keys()` : Returns a list of all keys in the dictionary.

```
   print(person.keys())  # Output: dict_keys(['name', 'age', 'city'])
```

```
dict_keys([])
```

2. `values()` : Returns a list of all values in the dictionary.

```
   print(person.values())  # Output: dict_values(['Alice', 25, 'New York'])
```

```
dict_values([])
```

3. `items()` : Returns a list of key-value pairs as tuples.

```
   print(person.items())  # Output: dict_items([('name', 'Alice'), ('age',
25), ('city', 'New York')])
```

```
dict_items([])
```

4. `update()` : Merges another dictionary into the current one.

```
   person.update({"country": "USA", "age": 26})
   print(person)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York',
'country': 'USA'}
```

```
{'country': 'USA', 'age': 26}
```

5. `copy()` : Returns a shallow copy of the dictionary.

```
   person_copy = person.copy()
   print(person_copy)
```

```
{'country': 'USA', 'age': 26}
```

Iterating Over a Dictionary

You can iterate over a dictionary using a `for` loop. By default, the loop iterates over the keys.

Example:

```python
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Iterate over keys
for key in person:
    print(key)

# Iterate over values
for value in person.values():
    print(value)

# Iterate over key-value pairs
for key, value in person.items():
    print(f"{key}: {value}")
```

```
name
age
city
Alice
25
New York
name: Alice
age: 25
city: New York
```

Dictionary Comprehensions

Similar to list comprehensions, dictionary comprehensions allow you to create dictionaries in a concise way.

Syntax:

```python
{key_expression: value_expression for item in iterable}
```

Example:

```python
# Create a dictionary of squares
squares = {x: x ** 2 for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Nested Dictionaries

Dictionaries can contain other dictionaries, allowing you to create complex data structures.

Example:

```
students = {
    "Alice": {"age": 25, "grade": "A"},
    "Bob": {"age": 22, "grade": "B"},
    "Charlie": {"age": 23, "grade": "C"}
}

# Accessing nested dictionary values
print(students["Alice"]["age"])  # Output: 25
```

```
25
```

Example Program

```
# Working with Dictionaries

# Creating a dictionary
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing values
print("Name:", person["name"])
print("Age:", person.get("age"))

# Adding/updating entries
person["country"] = "USA"
person["age"] = 26

# Removing entries
del person["city"]
age = person.pop("age")
last_item = person.popitem()
```

```
# Dictionary methods
print("Keys:", person.keys())
print("Values:", person.values())
print("Items:", person.items())

# Iterating over a dictionary
for key, value in person.items():
    print(f"{key}: {value}")

# Dictionary comprehension
squares = {x: x ** 2 for x in range(1, 6)}
print("Squares:", squares)

# Nested dictionaries
students = {
    "Alice": {"age": 25, "grade": "A"},
    "Bob": {"age": 22, "grade": "B"},
    "Charlie": {"age": 23, "grade": "C"}
}
print("Alice's age:", students["Alice"]["age"])
```

```
Name: Alice
Age: 25
Keys: dict_keys(['name'])
Values: dict_values(['Alice'])
Items: dict_items([('name', 'Alice')])
name: Alice
Squares: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
Alice's age: 25
```

---

14. While Loop

A **while loop** is used to repeatedly execute a block of code as long as a condition is `True`. It is useful when you don't know in advance how many times the loop needs to run.

Syntax of a While Loop

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

- `condition`: An expression that evaluates to `True` or `False`. The loop continues as long as the condition is `True`.
- **Indentation**: The code block under the `while` statement must be indented (usually by 4 spaces).

Example of a Simple While Loop

```
count = 0

while count < 5:
    print("Count:", count)
    count += 1  # Increment count
```

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

- The loop runs as long as `count < 5` is `True`. Once `count` reaches 5, the condition becomes `False`, and the loop stops.

Infinite While Loop

If the condition of a `while` loop is always `True`, the loop will run indefinitely. This is called an **infinite loop**.

Example:

```
while True:
    print("This is an infinite loop!")
```

- To stop an infinite loop, you can use `Ctrl+C` in the terminal or add a `break` statement (explained later).

Breaking Out of a While Loop

You can use the `break` statement to exit a loop prematurely, even if the condition is still `True`.

Example:

```
count = 0

while True:
    print("Count:", count)
    count += 1
    if count >= 5:
        break  # Exit the loop
```

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

Skipping Iterations with Continue

The `continue` statement skips the rest of the code in the current iteration and moves to the next iteration of the loop.

Example:

```
count = 0

while count < 5:
    count += 1
    if count == 3:
        continue  # Skip the rest of the code for this iteration
    print("Count:", count)
```

```
Count: 1
Count: 2
Count: 4
Count: 5
```

- When `count` is 3, the `continue` statement skips the `print()` statement.

While Loop with Else

You can add an `else` block to a `while` loop. The `else` block executes when the loop condition becomes `False`. However, if the loop is exited using a `break` statement, the `else` block is skipped.

Example:

```
count = 0

while count < 5:
    print("Count:", count)
    count += 1
else:
    print("Loop finished!")
```

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
Loop finished!
```

Nested While Loops

You can nest `while` loops inside other `while` loops to create more complex logic.

Example:

```
i = 1

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

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
i: 2, j: 2
i: 2, j: 3
i: 3, j: 1
```

```
i: 3, j: 2
i: 3, j: 3
```

Practical Example: Guessing Game

Here’s a simple guessing game using a `while` loop:

```python
secret_number = 7
guess = None

while guess != secret_number:
    guess = int(input("Guess the secret number (between 1 and 10): "))
    if guess < secret_number:
        print("Too low!")
    elif guess > secret_number:
        print("Too high!")
    else:
        print("Congratulations! You guessed it!")
```

```
Guess the secret number (between 1 and 10):  5


Too low!


Guess the secret number (between 1 and 10):  9


Too high!


Guess the secret number (between 1 and 10):  7


Congratulations! You guessed it!
```

Example Program

```python
# Working with While Loops

# Simple While Loop
count = 0
```

```python
while count < 5:
    print("Count:", count)
    count += 1

# Infinite While Loop with Break
count = 0
while True:
    print("Count:", count)
    count += 1
    if count >= 5:
        break

# While Loop with Continue
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue
    print("Count:", count)

# While Loop with Else
count = 0
while count < 5:
    print("Count:", count)
    count += 1
else:
    print("Loop finished!")

# Nested While Loops
i = 1
while i <= 3:
    j = 1
    while j <= 3:
        print(f"i: {i}, j: {j}")
        j += 1
    i += 1

# Guessing Game
secret_number = 7
guess = None
while guess != secret_number:
    guess = int(input("Guess the secret number (between 1 and 10): "))
    if guess < secret_number:
        print("Too low!")
    elif guess > secret_number:
        print("Too high!")
    else:
        print("Congratulations! You guessed it!")
```

16. For Loops

A **for loop** is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence. For loops are commonly used when you know in advance how many times you want to repeat a task.

Syntax of a For Loop

```
for item in sequence:
    # Code to execute
```

- `item` : A variable that takes the value of each element in the sequence during each iteration.
- `sequence` : A collection of items (e.g., list, tuple, string, or range).
- **Indentation**: The code block under the `for` statement must be indented (usually by 4 spaces).

Iterating Over a List

You can use a `for` loop to iterate over a list and perform an action for each item.

Example:

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

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

```
apple
banana
cherry
```

Iterating Over a String

A string is a sequence of characters, so you can iterate over each character in a string.

Example:

```
message = "Hello"

for char in message:
    print(char)
```

```
H
e
l
l
o
```

Using the `range()` Function

The `range()` function generates a sequence of numbers, which is often used in `for` loops.

Syntax:

```
range(start, stop, step)
```

- `start` : The starting value (inclusive). Default is `0` .
- `stop` : The ending value (exclusive).
- `step` : The increment between numbers. Default is `1` .

Examples:

1. Iterate over a range of numbers:

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

```
0
1
2
3
4
```

2. Specify a start and stop:

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

```
2
3
4
5
```

3. Specify a step:

```
    for i in range(1, 10, 2):
        print(i)
```

```
1
3
5
7
9
```

Nested For Loops

You can nest `for` loops inside other `for` loops to create more complex logic.

Example:

```
for i in range(3):
    for j in range(3):
        print(f"i: {i}, j: {j}")
```

```
i: 0, j: 0
i: 0, j: 1
i: 0, j: 2
i: 1, j: 0
i: 1, j: 1
i: 1, j: 2
i: 2, j: 0
i: 2, j: 1
i: 2, j: 2
```

Breaking Out of a For Loop

You can use the `break` statement to exit a `for` loop prematurely.

Example:

```
for i in range(10):
    if i == 5:
        break  # Exit the loop
    print(i)
```

```
0
1
2
3
4
```

Skipping Iterations with Continue

The `continue` statement skips the rest of the code in the current iteration and moves to the next iteration.

Example:

```
for i in range(5):
    if i == 2:
        continue  # Skip the rest of the code for this iteration
    print(i)
```

```
0
1
3
4
```

For Loop with Else

You can add an `else` block to a `for` loop. The `else` block executes when the loop finishes normally (i.e., without a `break` statement).

Example:

```
for i in range(3):
    print(i)
else:
    print("Loop finished!")
```

```
0
1
2
Loop finished!
```

Practical Example: Summing Numbers

Here's an example of using a `for` loop to calculate the sum of numbers in a list:

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

for num in numbers:
    total += num

print("Sum:", total)
```

```
Sum: 15
```

Example Program

```
# Working with For Loops

# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Iterating over a string
message = "Hello"
for char in message:
    print(char)

# Using range()
for i in range(5):
    print(i)

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

for i in range(1, 10, 2):
    print(i)
```

```python
# Nested For Loops
for i in range(3):
    for j in range(3):
        print(f"i: {i}, j: {j}")

# Breaking out of a loop
for i in range(10):
    if i == 5:
        break
    print(i)

# Skipping iterations with continue
for i in range(5):
    if i == 2:
        continue
    print(i)

# For Loop with Else
for i in range(3):
    print(i)
else:
    print("Loop finished!")

# Summing numbers in a list
numbers = [1, 2, 3, 4, 5]
total = 0
for num in numbers:
    total += num
print("Sum:", total)
```

---

17. Exponent Function

The **exponent function** is used to raise a number to a specified power. In Python, you can calculate exponents using the `**` operator or the built-in `pow()` function. Additionally, the `math` module provides more advanced exponentiation capabilities.

Using the `` Operator**

The `**` operator is the simplest way to calculate exponents in Python.

Syntax:

```
base ** exponent
```

Examples:

```python
# Calculate 2 raised to the power of 3
result = 2 ** 3
print(result)  # Output: 8

# Calculate 5 raised to the power of 2
result = 5 ** 2
print(result)  # Output: 25
```

```
8
25
```

Using the `pow()` Function

The `pow()` function is a built-in function that calculates the power of a number. It takes two arguments: the base and the exponent.

Syntax:

```python
pow(base, exponent)
```

Examples:

```python
# Calculate 2 raised to the power of 3
result = pow(2, 3)
print(result)  # Output: 8

# Calculate 5 raised to the power of 2
result = pow(5, 2)
print(result)  # Output: 25
```

```
8
25
```

Using the `math.pow()` Function

The `math` module provides a `pow()` function that works similarly to the built-in `pow()` function but always returns a float.

Syntax:

```
math.pow(base, exponent)
```

Steps:

1. Import the `math` module.
2. Use `math.pow()` to calculate the exponent.

Example:

```
import math

# Calculate 2 raised to the power of 3
result = math.pow(2, 3)
print(result)  # Output: 8.0

# Calculate 5 raised to the power of 2
result = math.pow(5, 2)
print(result)  # Output: 25.0
```

```
8.0
25.0
```

Handling Negative Exponents

You can use negative exponents to calculate the reciprocal of a number raised to a power.

Examples:

```
# Calculate 2 raised to the power of -3
result = 2 ** -3
print(result)  # Output: 0.125

# Calculate 5 raised to the power of -2
result = pow(5, -2)
print(result)  # Output: 0.04
```

```
0.125
0.04
```

Handling Fractional Exponents

Fractional exponents allow you to calculate roots. For example, raising a number to the power of `1/2` calculates its square root.

Examples:

```
# Calculate the square root of 16
result = 16 ** 0.5
print(result)  # Output: 4.0

# Calculate the cube root of 27
result = 27 ** (1/3)
print(result)  # Output: 3.0
```

```
4.0
3.0
```

Practical Example: Custom Exponent Function

You can create a custom function to calculate exponents. This is useful if you want to add additional logic or error handling.

Example:

```
def exponent(base, power):
    return base ** power

# Calculate 2 raised to the power of 3
result = exponent(2, 3)
print(result)  # Output: 8
```

```
8
```

Example Program

```
# Working with Exponent Functions

# Using the ** operator
```

```python
result = 2 ** 3
print("2 ** 3 =", result)  # Output: 8

result = 5 ** 2
print("5 ** 2 =", result)  # Output: 25

# Using the pow() function
result = pow(2, 3)
print("pow(2, 3) =", result)  # Output: 8

result = pow(5, 2)
print("pow(5, 2) =", result)  # Output: 25

# Using math.pow()
import math
result = math.pow(2, 3)
print("math.pow(2, 3) =", result)  # Output: 8.0

result = math.pow(5, 2)
print("math.pow(5, 2) =", result)  # Output: 25.0

# Handling negative exponents
result = 2 ** -3
print("2 ** -3 =", result)  # Output: 0.125

result = pow(5, -2)
print("pow(5, -2) =", result)  # Output: 0.04

# Handling fractional exponents
result = 16 ** 0.5
print("16 ** 0.5 =", result)  # Output: 4.0

result = 27 ** (1/3)
print("27 ** (1/3) =", result)  # Output: 3.0

# Custom exponent function
def exponent(base, power):
    return base ** power

result = exponent(2, 3)
print("exponent(2, 3) =", result)  # Output: 8
```

```
2 ** 3 = 8
5 ** 2 = 25
pow(2, 3) = 8
pow(5, 2) = 25
math.pow(2, 3) = 8.0
math.pow(5, 2) = 25.0
2 ** -3 = 0.125
```

```
pow(5, -2) = 0.04
16 ** 0.5 = 4.0
27 ** (1/3) = 3.0
exponent(2, 3) = 8
```

---

18. 2D Lists & Nested Loops

A **2D list** (or **list of lists**) is a list where each element is itself a list. This is often used to represent grids, matrices, or tables. To work with 2D lists, you typically use **nested loops**—a loop inside another loop—to iterate over the rows and columns.

Creating a 2D List

You can create a 2D list by nesting lists inside another list.

Example:

```
# A 2D list representing a 3x3 grid
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Accessing Elements in a 2D List

To access an element in a 2D list, use two indices: the first for the row and the second for the column.

Syntax:

```
matrix[row][column]
```

Example:

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

```
# Access the element in the second row, third column
print(matrix[1][2])  # Output: 6
```

```
6
```

Iterating Over a 2D List Using Nested Loops

To iterate over all elements in a 2D list, use a **nested loop**:

- The outer loop iterates over the rows.
- The inner loop iterates over the columns.

Example:

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

for row in matrix:
    for element in row:
        print(element, end=" ")  # Print elements in the same row
    print()  # Move to the next line after each row
```

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

Modifying a 2D List

You can modify elements in a 2D list using their indices.

Example:

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

```
# Change the element in the first row, second column
matrix[0][1] = 10

print(matrix)
# Output: [[1, 10, 3], [4, 5, 6], [7, 8, 9]]
```

```
[[1, 10, 3], [4, 5, 6], [7, 8, 9]]
```

Creating a 2D List Using List Comprehension

You can use **list comprehension** to create a 2D list in a concise way.

Example:

```
# Create a 3x3 matrix with all elements set to 0
matrix = [[0 for _ in range(3)] for _ in range(3)]
print(matrix)
# Output: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

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

Practical Example: Matrix Addition

Here's an example of adding two 2D lists (matrices) element-wise:

```
# Define two 2x2 matrices
matrix1 = [
    [1, 2],
    [3, 4]
]

matrix2 = [
    [5, 6],
    [7, 8]
]

# Create a result matrix with the same dimensions
result = [[0 for _ in range(2)] for _ in range(2)]

# Perform matrix addition
for i in range(2):  # Iterate over rows
    for j in range(2):  # Iterate over columns
```

```
        result[i][j] = matrix1[i][j] + matrix2[i][j]

print(result)
# Output: [[6, 8], [10, 12]]
```

```
[[6, 8], [10, 12]]
```

Example Program

```
# Working with 2D Lists & Nested Loops

# Creating a 2D list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing elements
print("Element at row 1, column 2:", matrix[1][2])  # Output: 6

# Iterating over a 2D list
print("Matrix elements:")
for row in matrix:
    for element in row:
        print(element, end=" ")
    print()

# Modifying a 2D list
matrix[0][1] = 10
print("Modified matrix:")
for row in matrix:
    for element in row:
        print(element, end=" ")
    print()

# Creating a 2D list using list comprehension
matrix2 = [[0 for _ in range(3)] for _ in range(3)]
print("2D list created using list comprehension:")
for row in matrix2:
    for element in row:
        print(element, end=" ")
    print()

# Matrix addition
```

```
matrix1 = [
    [1, 2],
    [3, 4]
]

matrix2 = [
    [5, 6],
    [7, 8]
]

result = [[0 for _ in range(2)] for _ in range(2)]

for i in range(2):
    for j in range(2):
        result[i][j] = matrix1[i][j] + matrix2[i][j]

print("Result of matrix addition:")
for row in result:
    for element in row:
        print(element, end=" ")
    print()
```

```
Element at row 1, column 2: 6
Matrix elements:
1 2 3
4 5 6
7 8 9
Modified matrix:
1 10 3
4 5 6
7 8 9
2D list created using list comprehension:
0 0 0
0 0 0
0 0 0
Result of matrix addition:
6 8
10 12
```

---

19. Comments

Comments are notes or explanations added to your code to make it easier to understand. They are ignored by the Python interpreter and are only meant for humans (developers, collaborators, or your future self). Comments are essential for writing clean, maintainable, and readable code.

Types of Comments

Python supports two types of comments:

1. **Single-line comments**: Used for short explanations or notes.
2. **Multi-line comments**: Used for longer descriptions or documentation.

Single-Line Comments

Single-line comments start with the `#` symbol. Everything after `#` on that line is ignored by the Python interpreter.

Syntax:

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

Example:

```python
# Calculate the sum of two numbers
a = 5
b = 10
sum = a + b  # Store the result in the variable 'sum'
print(sum)
```

```
15
```

Multi-Line Comments

Python doesn't have a specific syntax for multi-line comments. However, you can use multi-line strings (enclosed in triple quotes `'''` or `"""`) to create block comments. These are not technically comments but are treated as strings and ignored if not assigned to a variable.

Syntax:

```python
"""
This is a multi-line comment.
It can span multiple lines.
"""
```

Example:

```
"""
This program calculates the area of a rectangle.
It takes the length and width as input and prints the area.
"""
length = 10
width = 5
area = length * width
print("Area:", area)
```

```
Area: 50
```

Best Practices for Using Comments

1. **Explain Why, Not What**: Comments should explain why the code is written a certain way, not what the code does (unless the code is complex or non-obvious).

```
   # Bad: This adds 5 and 10
   sum = 5 + 10

   # Good: Calculate the total cost including tax
   total_cost = price + (price * tax_rate)
```

2. **Keep Comments Up-to-Date**: If you change the code, make sure to update the comments to reflect the changes.
3. **Avoid Over-Commenting**: Don't add comments for every line of code. Only comment when necessary to clarify complex logic or decisions.
4. **Use Comments for TODOs**: Use comments to mark areas of the code that need improvement or additional work.

```
   # TODO: Optimize this function for better performance
   def calculate_sum(numbers):
       return sum(numbers)
```

Inline Comments

Inline comments are placed on the same line as the code. They should be used sparingly and only to clarify complex or non-obvious code.

Example:

```
x = 10  # Initialize x with a value of 10
```

Docstrings

Docstrings are a special type of multi-line comment used to document functions, classes, and modules. They are enclosed in triple quotes and are accessible at runtime using the `__doc__` attribute.

Example:

```
def add(a, b):
    """
    This function adds two numbers and returns the result.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of a and b.
    """
    return a + b

# Access the docstring
print(add.__doc__)
```

```
    This function adds two numbers and returns the result.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of a and b.
```

Example Program

```
# Working with Comments

# Single-line comment
# This program calculates the area of a rectangle
```

```python
# Multi-line comment
"""
This program takes the length and width of a rectangle as input,
calculates the area, and prints the result.
"""

# Variables
length = 10  # Length of the rectangle
width = 5    # Width of the rectangle

# Calculate area
area = length * width  # Area = length * width

# Print the result
print("Area:", area)

# Function with a docstring
def multiply(a, b):
    """
    This function multiplies two numbers and returns the result.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The product of a and b.
    """
    return a * b

# Access the docstring
print(multiply.__doc__)
```

```
Area: 50

    This function multiplies two numbers and returns the result.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The product of a and b.
```

20. Try / Except

The **try/except** block is used in Python to handle exceptions (errors) that occur during the execution of a program. Instead of crashing the program, you can catch and handle exceptions gracefully, allowing the program to continue running or provide meaningful feedback to the user.

What Are Exceptions?

Exceptions are errors that occur during the execution of a program. Examples include:

- `ZeroDivisionError` : Division by zero.
- `TypeError` : Performing an operation on incompatible types.
- `ValueError` : Passing an invalid value to a function.
- `FileNotFoundError` : Trying to open a file that doesn't exist.

Syntax of Try/Except

The basic structure of a `try/except` block is as follows:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Code to handle the exception
```

- `try` **block**: Contains the code that might raise an exception.
- `except` **block**: Contains the code to handle the exception if it occurs.

Handling Specific Exceptions

You can specify the type of exception to catch in the `except` block. This allows you to handle different exceptions differently.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a number.")
```

```
Enter a number:  0


Error: Cannot divide by zero.
```

- If the user enters `0`, the program will catch the `ZeroDivisionError` and print `"Error: Cannot divide by zero."`
- If the user enters a non-numeric value, the program will catch the `ValueError` and print `"Error: Invalid input. Please enter a number."`

Handling Multiple Exceptions in One Block

You can handle multiple exceptions in a single `except` block by specifying them as a tuple.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except (ZeroDivisionError, ValueError):
    print("Error: Invalid input or division by zero.")
```

```
Enter a number:  a


Error: Invalid input or division by zero.
```

Using a Generic Exception

You can use a generic `except` block to catch all exceptions. However, this is generally not recommended because it can hide unexpected errors.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except:
    print("An error occurred.")
```

```
Enter a number:  0


An error occurred.
```

The Else Block

The `else` block is executed if no exceptions occur in the `try` block. It is useful for code that should only run if the `try` block succeeds.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a number.")
else:
    print("Result:", result)
```

```
Enter a number:  0


Error: Cannot divide by zero.
```

The Finally Block

The `finally` block is executed no matter what—whether an exception occurs or not. It is typically used for cleanup actions, such as closing files or releasing resources.

Example:

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
finally:
    file.close()
    print("File closed.")
```

```
File closed.
```

Raising Exceptions

You can raise exceptions manually using the `raise` keyword. This is useful for enforcing constraints or signaling errors in your code.

Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)  # Output: Cannot divide by zero.
```

```
Cannot divide by zero.
```

Custom Exceptions

You can define your own exceptions by creating a new class that inherits from Python's built-in `Exception` class.

Example:

```
class NegativeNumberError(Exception):
    pass

def check_positive(number):
    if number < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")
```

```
try:
    check_positive(-5)
except NegativeNumberError as e:
    print(e)  # Output: Negative numbers are not allowed.
```

```
Negative numbers are not allowed.
```

Example Program

```
# Working with Try/Except

# Handling specific exceptions
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a number.")

# Using else and finally
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
else:
    print("File read successfully.")
finally:
    file.close()
    print("File closed.")

# Raising exceptions
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)
```

```
# Custom exceptions
class NegativeNumberError(Exception):
    pass

def check_positive(number):
    if number < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")

try:
    check_positive(-5)
except NegativeNumberError as e:
    print(e)
```

```
Enter a number:  0


Error: Cannot divide by zero.

File read successfully.
File closed.
Cannot divide by zero.
Negative numbers are not allowed.
```

---

21. Reading Files

Reading files is a common task in programming. Python provides built-in functions to open, read, and manipulate files. Files can contain text, data, or any other information, and reading them allows you to process their contents in your program.

Opening a File

To read a file, you first need to open it using the `open()` function. The `open()` function returns a file object, which provides methods for reading and manipulating the file.

Syntax:

```
file = open("filename", "mode")
```

- `filename` : The name of the file (including the path if necessary).
- `mode` : The mode in which the file is opened. For reading, use `"r"` (read mode).

Example:

```
file = open("example.txt", "r")
```

Reading the Entire File

You can read the entire contents of a file using the `read()` method.

Example:

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

```
Hello from the file
this is the example file for working with files in the python
```

- `read()` : Reads the entire file as a single string.
- `close()` : Closes the file to free up system resources.

Reading Line by Line

You can read a file line by line using the `readline()` method or iterate over the file object directly.

1. **Using** `readline()` :

```
    file = open("example.txt", "r")
    line = file.readline()
    while line:
        print(line, end="")  # end="" prevents extra newlines
        line = file.readline()
    file.close()
```

```
Hello from the file
this is the example file for working with files in the python
```

2. **Using a** `for` **loop**:

```
    file = open("example.txt", "r")
    for line in file:
        print(line, end="")
    file.close()
```

```
Hello from the file
this is the example file for working with files in the python
```

Reading All Lines into a List

You can read all lines of a file into a list using the `readlines()` method.

Example:

```
file = open("example.txt", "r")
lines = file.readlines()
for line in lines:
    print(line, end="")
file.close()
```

```
Hello from the file
this is the example file for working with files in the python
```

- `readlines()` : Returns a list where each element is a line from the file.

Using `with` for File Handling

The `with` statement is the recommended way to work with files. It automatically closes the file when the block inside `with` is exited, even if an exception occurs.

Syntax:

```
with open("filename", "mode") as file:
    # Code to work with the file
```

Example:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
Hello from the file
this is the example file for working with files in the python
```

- No need to call `close()` explicitly—it's handled automatically.

Handling File Not Found Errors

If the file does not exist, Python will raise a `FileNotFoundError`. You can handle this using a `try/except` block.

Example:

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Error: File not found.")
```

```
Hello from the file
this is the example file for working with files in the python
```

Reading Specific Parts of a File

You can read a specific number of characters from a file using the `read(size)` method, where `size` is the number of characters to read.

Example:

```
with open("example.txt", "r") as file:
    first_10_chars = file.read(10)
    print(first_10_chars)
```

```
Hello from
```

Example Program

```python
# Working with Reading Files

# Reading the entire file
with open("example.txt", "r") as file:
    content = file.read()
    print("Entire file content:")
    print(content)

# Reading line by line
with open("example.txt", "r") as file:
    print("\nFile content line by line:")
    for line in file:
        print(line, end="")

# Reading all lines into a list
with open("example.txt", "r") as file:
    lines = file.readlines()
    print("\nFile content as a list of lines:")
    for line in lines:
        print(line, end="")

# Handling file not found errors
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("\nError: File not found.")

# Reading specific parts of a file
with open("example.txt", "r") as file:
    first_10_chars = file.read(10)
    print("\nFirst 10 characters of the file:")
    print(first_10_chars)
```

```
Entire file content:
Hello from the file
this is the example file for working with files in the python



File content line by line:
Hello from the file
this is the example file for working with files in the python

File content as a list of lines:
Hello from the file
this is the example file for working with files in the python
```

```
Error: File not found.

First 10 characters of the file:
Hello from
```

---

22. Writing to Files

Writing to files is a common task in programming. Python provides built-in functions to open, write, and manipulate files. You can create new files, overwrite existing files, or append to existing files.

Opening a File for Writing

To write to a file, you need to open it in write mode (`"w"`) or append mode (`"a"`). The `open()` function returns a file object, which provides methods for writing to the file.

Syntax:

```python
file = open("filename", "mode")
```

- `filename` : The name of the file (including the path if necessary).
- `mode` :
  - `"w"` : Write mode (overwrites the file if it exists or creates a new file if it doesn't).
  - `"a"` : Append mode (adds to the end of the file if it exists or creates a new file if it doesn't).

Example:

```python
file = open("example.txt", "w")
```

Writing to a File

You can write to a file using the `write()` method. This method writes a string to the file.

Example:

```python
file = open("example.txt", "w")
file.write("Hello, World!\n")
file.write("This is a new line.")
file.close()

# Read file content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
Hello, World!
This is a new line.
```

- `write()` : Writes a string to the file.
- `close()` : Closes the file to free up system resources.

Appending to a File

To add content to the end of a file without overwriting it, open the file in append mode (`"a"`).

Example:

```
file = open("example.txt", "a")
file.write("\nThis line is appended.")
file.close()

# Read file content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
Hello, World!
This is a new line.
This line is appended.
```

Using `with` for File Handling

The `with` statement is the recommended way to work with files. It automatically closes the file when the block inside `with` is exited, even if an exception occurs.

Syntax:

```
with open("filename", "mode") as file:
    # Code to work with the file
```

Example:

```
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a new line.")

# Read file content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
Hello, World!
This is a new line.
```

Writing Multiple Lines

You can write multiple lines to a file using the `writelines()` method. This method takes a list of strings and writes them to the file.

Example:

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]

with open("example.txt", "w") as file:
    file.writelines(lines)

# Read file content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
Line 1
Line 2
Line 3
```

Handling File Errors

If there’s an issue with the file (e.g., permission errors), Python will raise an exception. You can handle these errors using a `try/except` block.

Example:

```
try:
    with open("test.txt", "w") as file:
        file.write("Hello, World!")
except IOError:
    print("Error: Could not write to the file.")
```

```
Error: Could not write to the file.
```

Example Program

```
# Working with Writing Files

# Writing to a new file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a new line.")

# Appending to an existing file
with open("example.txt", "a") as file:
    file.write("\nThis line is appended.")

# Writing multiple lines
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("example.txt", "w") as file:
    file.writelines(lines)

# Handling file errors
try:
    with open("example.txt", "w") as file:
        file.write("Hello, World!")
except IOError:
    print("Error: Could not write to the file.")

# Reading the file to verify its contents
with open("example.txt", "r") as file:
    content = file.read()
    print("File content:")
    print(content)
```

```
File content:
Hello, World!
```

---

23. Modules & Pip

Modules and **pip** are essential tools in Python for organizing code and managing external libraries. Modules allow you to reuse code across multiple programs, while pip is the package installer for Python, enabling you to install and manage third-party libraries.

What Are Modules?

A **module** is a file containing Python code (functions, classes, or variables) that can be imported and used in other programs. Modules help you organize your code into reusable components.

Creating a Module

To create a module, simply save your Python code in a `.py` file. For example, save the following code in a file named `mymodule.py`:

```python
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b
```

Importing a Module

You can import a module using the `import` statement. Once imported, you can access its functions, classes, or variables using the dot notation.

Example:

```python
import mymodule

# Using functions from the module
print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.add(3, 5))       # Output: 8
```

```
Hello, Alice!
8
```

Importing Specific Functions

You can import specific functions or variables from a module using the `from ... import` statement.

Example:

```python
from mymodule import greet, add

# Using the imported functions
print(greet("Bob"))  # Output: Hello, Bob!
print(add(2, 4))     # Output: 6
```

```
Hello, Bob!
6
```

Renaming Imports

You can rename a module or function when importing it using the `as` keyword.

Example:

```python
import mymodule as mm

print(mm.greet("Charlie"))  # Output: Hello, Charlie!
```

```
Hello, Charlie!
```

Standard Library Modules

Python comes with a rich set of built-in modules, known as the **Standard Library**. These modules provide functionality for tasks like math, file handling, and working with dates.

Example:

```python
import math

print(math.sqrt(16))  # Output: 4.0
```

```
4.0
```

What is Pip?

Pip is the package installer for Python. It allows you to install, upgrade, and manage third-party libraries and packages from the **Python Package Index (PyPI)**.

Installing Packages with Pip

To install a package, use the following command in your terminal or command prompt:

```
pip install package_name
```

Example:

```
pip install requests
```

Using Installed Packages

Once a package is installed, you can import and use it in your Python programs.

Example:

```
import requests

response = requests.get("https://www.example.com")
print(response.status_code)  # Output: 200 (if the request is successful)
```

```
200
```

Listing Installed Packages

To see a list of installed packages, use the following command:

```
pip list
```

Upgrading Packages

To upgrade an installed package to the latest version, use:

```
pip install --upgrade package_name
```

Example:

```
pip install --upgrade requests
```

Uninstalling Packages

To uninstall a package, use:

```
pip uninstall package_name
```

Example:

```
pip uninstall requests
```

Creating a Requirements File

A `requirements.txt` file lists all the dependencies for a project. You can generate this file using:

```
pip freeze > requirements.txt
```

To install all dependencies from a `requirements.txt` file, use:

```
pip install -r requirements.txt
```

Example Program

```
# Working with Modules & Pip

# Importing a custom module
```

```python
import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.add(3, 5))       # Output: 8

# Importing specific functions
from mymodule import greet, add

print(greet("Bob"))  # Output: Hello, Bob!
print(add(2, 4))     # Output: 6

# Renaming imports
import mymodule as mm

print(mm.greet("Charlie"))  # Output: Hello, Charlie!

# Using a standard library module
import math

print(math.sqrt(16))  # Output: 4.0

# Using an installed package (e.g., requests)
import requests

response = requests.get("https://www.example.com")
print(response.status_code)  # Output: 200 (if the request is successful)
```

```
Hello, Alice!
8
Hello, Bob!
6
Hello, Charlie!
4.0
200
```

---

24. Classes & Objects

Classes and **objects** are the foundation of **object-oriented programming (OOP)** in Python. A class is a blueprint for creating objects, and an object is an instance of a class. OOP allows you to structure your code in a way that models real-world entities and their relationships.

What is a Class?

A **class** is a template or blueprint that defines the properties (attributes) and behaviors (methods) of objects. It encapsulates data and functionality into a single unit.

Defining a Class

To define a class, use the `class` keyword followed by the class name. By convention, class names are written in **CamelCase**.

Syntax:

```
class ClassName:
    # Class attributes and methods
```

Example:

```
class Dog:
    # Class attribute (shared by all instances)
    species = "Canis familiaris"

    # Constructor method (initializes object attributes)
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Instance method
    def bark(self):
        return f"{self.name} says woof!"
```

What is an Object?

An **object** is an instance of a class. You can create multiple objects from a single class, each with its own unique attributes.

Creating Objects

To create an object, call the class name as if it were a function. This invokes the `__init__` method (constructor) to initialize the object.

Example:

```
# Create objects of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Access attributes
print(dog1.name)  # Output: Buddy
print(dog2.age)   # Output: 5

# Call methods
print(dog1.bark())  # Output: Buddy says woof!
```

```
Buddy
5
Buddy says woof!
```

The `self` Parameter

The `self` parameter refers to the current instance of the class. It is used to access instance attributes and methods within the class.

Class Attributes vs. Instance Attributes

- **Class attributes**: Shared by all instances of the class.
- **Instance attributes**: Unique to each instance.

Example:

```
print(dog1.species)  # Output: Canis familiaris (class attribute)
print(dog2.species)  # Output: Canis familiaris (class attribute)

dog1.species = "Golden Retriever"  # Modifying instance attribute
print(dog1.species)  # Output: Golden Retriever
print(dog2.species)  # Output: Canis familiaris (unchanged)
```

```
Canis familiaris
Canis familiaris
Golden Retriever
Canis familiaris
```

Adding Methods to a Class

Methods are functions defined inside a class. They define the behavior of objects.

Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} says woof!"

    def get_age(self):
        return f"{self.name} is {self.age} years old."

# Create an object
dog = Dog("Buddy", 3)

# Call methods
print(dog.bark())       # Output: Buddy says woof!
print(dog.get_age())    # Output: Buddy is 3 years old.
```

```
Buddy says woof!
Buddy is 3 years old.
```

The `__str__` Method

The `__str__` method is a special method that returns a string representation of the object. It is called when you use the `print()` function or `str()` on the object.

Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old."

# Create an object
dog = Dog("Buddy", 3)
```

```
# Print the object
print(dog)  # Output: Buddy is 3 years old.
```

```
Buddy is 3 years old.
```

Inheritance

Inheritance allows you to create a new class (child class) that inherits attributes and methods from an existing class (parent class). This promotes code reuse and modularity.

Example:

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

# Child class
class Dog(Animal):
    def speak(self):
        return f"{self.name} says woof!"

# Create objects
animal = Animal("Generic Animal")
dog = Dog("Buddy")

# Call methods
print(animal.speak())  # Output: Generic Animal makes a sound.
print(dog.speak())     # Output: Buddy says woof!
```

```
Generic Animal makes a sound.
Buddy says woof!
```

Example Program

```
# Working with Classes & Objects

# Define a class
```

```python
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        return f"{self.name} says woof!"

    # Special method for string representation
    def __str__(self):
        return f"{self.name} is {self.age} years old."

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Access attributes
print(dog1.name)  # Output: Buddy
print(dog2.age)   # Output: 5

# Call methods
print(dog1.bark())  # Output: Buddy says woof!
print(dog2.bark())  # Output: Max says woof!

# Print objects
print(dog1)  # Output: Buddy is 3 years old.
print(dog2)  # Output: Max is 5 years old.

# Inheritance example
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def speak(self):
        return f"{self.name} says woof!"

# Create objects
animal = Animal("Generic Animal")
dog = Dog("Buddy")

# Call methods
```

```
print(animal.speak())  # Output: Generic Animal makes a sound.
print(dog.speak())     # Output: Buddy says woof!
```

```
Buddy
5
Buddy says woof!
Max says woof!
Buddy is 3 years old.
Max is 5 years old.
Generic Animal makes a sound.
Buddy says woof!
```

---

25. Object Functions

Object functions (also called **methods**) are functions defined within a class that operate on the attributes of an object. They define the behavior of objects and allow you to perform actions or computations using the object's data.

Defining Object Functions

Object functions are defined inside a class and take `self` as their first parameter. The `self` parameter refers to the instance of the class and allows you to access its attributes and other methods.

Syntax:

```
class ClassName:
    def method_name(self, parameters):
        # Code to execute
```

Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Object function
    def bark(self):
        return f"{self.name} says woof!"

    # Another object function
    def get_age(self):
        return f"{self.name} is {self.age} years old."
```

Calling Object Functions

To call an object function, use the dot notation on an instance of the class.

Example:

```
# Create an object
dog = Dog("Buddy", 3)

# Call object functions
print(dog.bark())      # Output: Buddy says woof!
print(dog.get_age())   # Output: Buddy is 3 years old.
```

```
Buddy says woof!
Buddy is 3 years old.
```

Modifying Object Attributes

Object functions can modify the attributes of an object.

Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Object function to update age
    def birthday(self):
        self.age += 1
        return f"{self.name} is now {self.age} years old."

# Create an object
dog = Dog("Buddy", 3)

# Call the birthday function
print(dog.birthday())  # Output: Buddy is now 4 years old.
print(dog.birthday())  # Output: Buddy is now 5 years old.
```

```
Buddy is now 4 years old.
Buddy is now 5 years old.
```

Using Object Functions with Parameters

Object functions can take additional parameters to perform more complex operations.

Example:

```python
class Calculator:
    def __init__(self, initial_value=0):
        self.value = initial_value

    # Object function with parameters
    def add(self, number):
        self.value += number
        return self.value

    def subtract(self, number):
        self.value -= number
        return self.value

# Create an object
calc = Calculator(10)

# Call object functions with parameters
print(calc.add(5))      # Output: 15
print(calc.subtract(3)) # Output: 12
```

```
15
12
```

Special Object Functions

Python provides special object functions (also called **magic methods** or **dunder methods**) that allow you to define how objects behave in certain situations, such as addition, comparison, or string representation.

Common Special Methods:

1. `__str__` : Returns a string representation of the object (used by `print()` and `str()`).
2. `__len__` : Returns the length of the object (used by `len()`).

3. `__add__`: Defines behavior for the `+` operator.
4. `__eq__`: Defines behavior for the `==` operator.

Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Special method for string representation
    def __str__(self):
        return f"{self.name} is {self.age} years old."

    # Special method for addition
    def __add__(self, other):
        return Dog(f"{self.name} and {other.name}", self.age + other.age)

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Use special methods
print(dog1)  # Output: Buddy is 3 years old.

combined_dog = dog1 + dog2
print(combined_dog)  # Output: Buddy and Max is 8 years old.
```

```
Buddy is 3 years old.
Buddy and Max is 8 years old.
```

Example Program

```
# Working with Object Functions

# Define a class
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Object function
    def bark(self):
        return f"{self.name} says woof!"
```

```python
    # Object function to update age
    def birthday(self):
        self.age += 1
        return f"{self.name} is now {self.age} years old."

    # Special method for string representation
    def __str__(self):
        return f"{self.name} is {self.age} years old."

# Create an object
dog = Dog("Buddy", 3)

# Call object functions
print(dog.bark())      # Output: Buddy says woof!
print(dog.birthday())  # Output: Buddy is now 4 years old.
print(dog.birthday())  # Output: Buddy is now 5 years old.

# Print the object
print(dog)  # Output: Buddy is 5 years old.

# Using object functions with parameters
class Calculator:
    def __init__(self, initial_value=0):
        self.value = initial_value

    def add(self, number):
        self.value += number
        return self.value

    def subtract(self, number):
        self.value -= number
        return self.value

# Create an object
calc = Calculator(10)

# Call object functions with parameters
print(calc.add(5))      # Output: 15
print(calc.subtract(3)) # Output: 12

# Special object functions
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old."

    def __add__(self, other):
        return Dog(f"{self.name} and {other.name}", self.age + other.age)

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Use special methods
print(dog1)  # Output: Buddy is 3 years old.

combined_dog = dog1 + dog2
print(combined_dog)  # Output: Buddy and Max is 8 years old.
```

```
Buddy says woof!
Buddy is now 4 years old.
Buddy is now 5 years old.
Buddy is 5 years old.
15
12
Buddy is 3 years old.
Buddy and Max is 8 years old.
```

---

26. Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class (called a **child class** or **subclass**) based on an existing class (called a **parent class** or **superclass**). The child class inherits attributes and methods from the parent class, promoting code reuse and modularity.

Why Use Inheritance?

- **Code Reuse**: Avoid duplicating code by inheriting common functionality from a parent class.
- **Modularity**: Organize code into logical hierarchies.
- **Extensibility**: Add or modify functionality in the child class without affecting the parent class.

Syntax of Inheritance

To create a child class, specify the parent class in parentheses after the child class name.

Syntax:

```python
class ParentClass:
    # Parent class attributes and methods

class ChildClass(ParentClass):
    # Child class attributes and methods
```

Example of Inheritance

Here's a simple example where a `Dog` class inherits from an `Animal` class:

```python
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

# Child class
class Dog(Animal):
    def speak(self):
        return f"{self.name} says woof!"

# Create objects
animal = Animal("Generic Animal")
dog = Dog("Buddy")

# Call methods
print(animal.speak())  # Output: Generic Animal makes a sound.
print(dog.speak())     # Output: Buddy says woof!
```

```
Generic Animal makes a sound.
Buddy says woof!
```

- The `Dog` class inherits the `__init__` method and the `name` attribute from the `Animal` class.
- The `Dog` class overrides the `speak` method to provide its own implementation.

The `super()` Function

The `super()` function allows you to call methods from the parent class. This is useful when you want to extend the functionality of a parent method in the child class.

Example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call the parent class's __init__ method
        self.breed = breed

    def speak(self):
        return f"{self.name} says woof!"

# Create an object
dog = Dog("Buddy", "Golden Retriever")

# Access attributes and methods
print(dog.name)   # Output: Buddy
print(dog.breed)  # Output: Golden Retriever
print(dog.speak())  # Output: Buddy says woof!
```

```
Buddy
Golden Retriever
Buddy says woof!
```

Method Overriding

When a child class defines a method with the same name as a method in the parent class, the child class's method **overrides** the parent class's method.

Example:

```python
class Animal:
    def speak(self):
        return "Animal sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"

# Create objects
animal = Animal()
dog = Dog()

print(animal.speak())  # Output: Animal sound
print(dog.speak())     # Output: Woof!
```

```
Animal sound
Woof!
```

Adding New Methods in the Child Class

You can add new methods in the child class that are not present in the parent class.

Example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def fetch(self):
        return f"{self.name} fetches the ball."

# Create an object
dog = Dog("Buddy")

# Call methods
print(dog.speak())  # Output: Buddy makes a sound.
print(dog.fetch())  # Output: Buddy fetches the ball.
```

```
Buddy makes a sound.
Buddy fetches the ball.
```

Multi-Level Inheritance

In multi-level inheritance, a child class inherits from another child class, creating a chain of inheritance.

Example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):
    def speak(self):
        return f"{self.name} says woof!"

class Puppy(Dog):
    def play(self):
        return f"{self.name} is playing."

# Create an object
puppy = Puppy("Max")

# Call methods
print(puppy.speak())  # Output: Max says woof!
print(puppy.play())   # Output: Max is playing.
```

```
Max says woof!
Max is playing.
```

Multiple Inheritance

Python supports **multiple inheritance**, where a child class can inherit from more than one parent class.

Example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Pet:
```

```python
    def play(self):
        return f"{self.name} is playing."

class Dog(Animal, Pet):
    def speak(self):
        return f"{self.name} says woof!"

# Create an object
dog = Dog("Buddy")

# Call methods
print(dog.speak())  # Output: Buddy says woof!
print(dog.play())   # Output: Buddy is playing.
```

```
Buddy says woof!
Buddy is playing.
```

Example Program

```python
# Working with Inheritance

# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

# Child class
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
        return f"{self.name} says woof!"

    def fetch(self):
        return f"{self.name} fetches the ball."

# Create objects
animal = Animal("Generic Animal")
dog = Dog("Buddy", "Golden Retriever")
```

```python
# Call methods
print(animal.speak())  # Output: Generic Animal makes a sound.
print(dog.speak())     # Output: Buddy says woof!
print(dog.fetch())     # Output: Buddy fetches the ball.

# Multi-level inheritance
class Puppy(Dog):
    def play(self):
        return f"{self.name} is playing."

puppy = Puppy("Max", "Labrador")
print(puppy.speak())  # Output: Max says woof!
print(puppy.play())   # Output: Max is playing.

# Multiple inheritance
class Pet:
    def play(self):
        return f"{self.name} is playing."

class Cat(Animal, Pet):
    def speak(self):
        return f"{self.name} says meow!"

cat = Cat("Whiskers")
print(cat.speak())  # Output: Whiskers says meow!
print(cat.play())   # Output: Whiskers is playing.
```

```
Generic Animal makes a sound.
Buddy says woof!
Buddy fetches the ball.
Max says woof!
Max is playing.
Whiskers says meow!
Whiskers is playing.
```

---

intermediate Python

---

1. Intro

The **Intermediate Python** section builds on the foundational knowledge covered in the **Beginner Python** section. Here, we'll explore more advanced concepts and techniques that will help you write cleaner, more efficient, and more professional Python code.

What to Expect in Intermediate Python

In this section, you'll learn about:

- Advanced data structures like **sets**, **collections**, and **itertools**.
- Powerful Python features like **lambda functions**, **decorators**, and **generators**.
- Techniques for handling **exceptions and errors** and **logging**.
- Working with **JSON** and generating **random numbers**.
- Understanding **function arguments** and the **asterisk (*) operator**.
- Concepts like **shallow vs deep copying** and **context managers**.
- Exploring **multithreading** and **multiprocessing** for concurrent programming.

By the end of this section, you'll have a deeper understanding of Python and be equipped to tackle more complex programming challenges.

Why Learn Intermediate Python?

1. **Write Cleaner Code**: Learn techniques to make your code more readable, modular, and maintainable.
2. **Improve Efficiency**: Use advanced features like generators and decorators to optimize your code.
3. **Handle Real-World Scenarios**: Master error handling, logging, and working with external data formats like JSON.
4. **Unlock Python's Full Potential**: Explore Python's powerful libraries and tools for tasks like multithreading and multiprocessing.

How to Use This Section

- **Follow Along**: Type out the examples and experiment with them to solidify your understanding.
- **Practice**: Try the exercises and challenges provided at the end of each topic.
- **Explore Further**: Use the official Python documentation and online resources to dive deeper into topics that interest you.

Example: A Taste of Intermediate Python

Here's a quick example to give you a taste of what's to come. We'll use a **lambda function** and the **map()** function to square a list of numbers:

```
# Using a lambda function with map()
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))

print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

- **Lambda functions**: Anonymous functions defined using the `lambda` keyword.
- **map()**: Applies a function to all items in an iterable (e.g., a list).

---

2. Lists

Lists are one of Python's most versatile and widely used data structures. In the **Beginner Python** section, we covered the basics of lists. Now, we'll dive deeper into advanced list operations, comprehensions, and performance considerations.

Recap: What Are Lists?

A **list** is an ordered, mutable collection of items. Lists can store elements of different data types and are defined using square brackets `[]`.

Example:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "apple", 3.14, True]
```

Advanced List Operations

1. **List Comprehensions**
   List comprehensions provide a concise way to create lists. They are faster and more readable than traditional loops.

   Syntax:

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

Example:

```
# Create a list of squares
squares = [x ** 2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]

# Filter even numbers
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8]
```

```
[1, 4, 9, 16, 25]
[0, 2, 4, 6, 8]
```

2. **Nested List Comprehensions**
   You can use nested list comprehensions to create lists of lists (2D lists).

   Example:

```
# Create a 3x3 matrix
matrix = [[i + j for j in range(3)] for i in range(3)]
print(matrix)
# Output: [[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

```
[[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

3. **Slicing with Steps**
   Slicing allows you to extract a portion of a list. You can also specify a step to skip elements.

   Example:

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

# Extract every second element
print(numbers[::2])  # Output: [0, 2, 4, 6, 8]

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

```
[0, 2, 4, 6, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

4. **List Unpacking**
   You can unpack a list into individual variables.

   Example:

```
fruits = ["apple", "banana", "cherry"]
a, b, c = fruits
print(a, b, c)  # Output: apple banana cherry
```

```
apple banana cherry
```

5. **Zip and Unzip Lists**
   The `zip()` function combines multiple lists into a list of tuples. You can also unzip them back into separate lists.

   Example:

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]

# Zip lists
zipped = list(zip(names, ages))
print(zipped)  # Output: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]

# Unzip lists
names_unzipped, ages_unzipped = zip(*zipped)
print(names_unzipped)  # Output: ('Alice', 'Bob', 'Charlie')
print(ages_unzipped)   # Output: (25, 30, 35)
```

```
[('Alice', 25), ('Bob', 30), ('Charlie', 35)]
('Alice', 'Bob', 'Charlie')
(25, 30, 35)
```

6. **List Methods**
   Python provides several built-in methods for working with lists. Here are some advanced ones:

   - `extend()` : Adds multiple elements to the end of a list.
   - `insert()` : Inserts an element at a specific index.
   - `pop()` : Removes and returns an element at a specific index.
   - `remove()` : Removes the first occurrence of a value.
   - `index()` : Returns the index of the first occurrence of a value.
   - `count()` : Returns the number of occurrences of a value.
   - `sort()` : Sorts the list in place.

- `reverse()` : Reverses the list in place.

Example:

```
numbers = [3, 1, 4, 1, 5, 9]

numbers.sort()
print(numbers)  # Output: [1, 1, 3, 4, 5, 9]

numbers.reverse()
print(numbers)  # Output: [9, 5, 4, 3, 1, 1]
```

```
[1, 1, 3, 4, 5, 9]
[9, 5, 4, 3, 1, 1]
```

Performance Considerations

- **Time Complexity**: Be aware of the time complexity of list operations. For example:
  - **Appending**: `O(1)`
  - **Inserting/Deleting**: `O(n)`
  - **Searching**: `O(n)`
- **Memory Usage**: Lists can consume a lot of memory for large datasets. Consider using **generators** or **arrays** (from the `array` module) for memory efficiency.

Example Program

```
# Working with Lists

# List comprehensions
squares = [x ** 2 for x in range(1, 6)]
print("Squares:", squares)  # Output: [1, 4, 9, 16, 25]

# Nested list comprehensions
matrix = [[i + j for j in range(3)] for i in range(3)]
print("Matrix:", matrix)  # Output: [[0, 1, 2], [1, 2, 3], [2, 3, 4]]

# Slicing with steps
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("Every second element:", numbers[::2])  # Output: [0, 2, 4, 6, 8]
print("Reversed list:", numbers[::-1])  # Output: [9, 8, 7, 6, 5, 4, 3, 2,
1, 0]
```

```python
# List unpacking
fruits = ["apple", "banana", "cherry"]
a, b, c = fruits
print("Unpacked:", a, b, c)  # Output: apple banana cherry

# Zip and unzip lists
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
zipped = list(zip(names, ages))
print("Zipped:", zipped)  # Output: [('Alice', 25), ('Bob', 30), ('Charlie',
35)]
names_unzipped, ages_unzipped = zip(*zipped)
print("Unzipped names:", names_unzipped)  # Output: ('Alice', 'Bob',
'Charlie')
print("Unzipped ages:", ages_unzipped)   # Output: (25, 30, 35)

# List methods
numbers = [3, 1, 4, 1, 5, 9]
numbers.sort()
print("Sorted:", numbers)  # Output: [1, 1, 3, 4, 5, 9]
numbers.reverse()
print("Reversed:", numbers)  # Output: [9, 5, 4, 3, 1, 1]
```

```
Squares: [1, 4, 9, 16, 25]
Matrix: [[0, 1, 2], [1, 2, 3], [2, 3, 4]]
Every second element: [0, 2, 4, 6, 8]
Reversed list: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Unpacked: apple banana cherry
Zipped: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
Unzipped names: ('Alice', 'Bob', 'Charlie')
Unzipped ages: (25, 30, 35)
Sorted: [1, 1, 3, 4, 5, 9]
Reversed: [9, 5, 4, 3, 1, 1]
```

---

3. Tuples

Tuples are another fundamental data structure in Python. They are similar to lists but with one key difference: **tuples are immutable**. This means that once a tuple is created, its elements cannot be changed, added, or removed. Tuples are often used for fixed collections of items, such as coordinates or database records.

What Are Tuples?

A **tuple** is an ordered, immutable collection of items. Tuples are defined using parentheses `()`.

Example:

```
coordinates = (10.0, 20.0)
fruits = ("apple", "banana", "cherry")
mixed = (1, "apple", 3.14, True)
```

Creating Tuples

You can create a tuple by enclosing elements in parentheses `()`. If a tuple has only one element, you must include a trailing comma to distinguish it from a regular value.

Example:

```
# Single-element tuple
single = (42,)

# Multiple elements
multiple = (1, 2, 3)
```

Accessing Tuple Elements

You can access tuple elements using indexing and slicing, just like lists.

Example:

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

# Access by index
print(fruits[0])  # Output: apple

# Negative indexing
print(fruits[-1])  # Output: cherry

# Slicing
print(fruits[1:3])  # Output: ('banana', 'cherry')
```

```
apple
cherry
('banana', 'cherry')
```

Tuples Are Immutable

Once a tuple is created, you cannot modify its elements. Attempting to do so will raise a `TypeError`.

Example:

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

# This will raise an error
fruits[0] = "orange"  # TypeError: 'tuple' object does not support item
assignment
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

Cell In[194], line 4
      1 fruits = ("apple", "banana", "cherry")
      3 # This will raise an error
----> 4 fruits[0] = "orange"


TypeError: 'tuple' object does not support item assignment
```

Tuple Operations

1. **Concatenation**: Combine two tuples using the `+` operator.

```
   tuple1 = (1, 2, 3)
   tuple2 = (4, 5, 6)
   combined = tuple1 + tuple2
   print(combined)  # Output: (1, 2, 3, 4, 5, 6)
```

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

2. **Repetition**: Repeat a tuple using the `*` operator.

```
   repeated = (1, 2) * 3
   print(repeated)  # Output: (1, 2, 1, 2, 1, 2)
```

```
(1, 2, 1, 2, 1, 2)
```

3. **Membership**: Check if an element exists in a tuple using the `in` keyword.

```
    fruits = ("apple", "banana", "cherry")
    print("banana" in fruits)  # Output: True
```

```
True
```

4. **Length**: Get the number of elements in a tuple using the `len()` function.

```
    print(len(fruits))  # Output: 3
```

```
3
```

Tuple Unpacking

You can unpack a tuple into individual variables. This is useful for assigning multiple values at once.

Example:

```
coordinates = (10.0, 20.0)
x, y = coordinates
print(x, y)  # Output: 10.0 20.0
```

```
10.0 20.0
```

Using Tuples as Dictionary Keys

Because tuples are immutable, they can be used as keys in dictionaries, unlike lists.

Example:

```
location = {
    (40.7128, -74.0060): "New York",
    (34.0522, -118.2437): "Los Angeles"
}
```

```
print(location[(40.7128, -74.0060)])  # Output: New York
```

```
New York
```

Tuple Methods

Tuples have only two built-in methods:

1. `count()` : Returns the number of occurrences of a value.

   ```
   numbers = (1, 2, 3, 1, 2, 1)
   print(numbers.count(1))  # Output: 3
   ```

```
3
```

2. `index()` : Returns the index of the first occurrence of a value.

   ```
   print(numbers.index(2))  # Output: 1
   ```

```
1
```

When to Use Tuples

- Use tuples when you need an immutable collection of items.
- Use tuples for fixed data, such as coordinates, database records, or function arguments.
- Use tuples as dictionary keys.

Example Program

```
# Working with Tuples

# Creating tuples
coordinates = (10.0, 20.0)
fruits = ("apple", "banana", "cherry")
single = (42,)
```

```python
# Accessing elements
print("First fruit:", fruits[0])  # Output: apple
print("Last fruit:", fruits[-1])  # Output: cherry
print("Sliced fruits:", fruits[1:3])  # Output: ('banana', 'cherry')

# Tuple operations
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined = tuple1 + tuple2
print("Combined tuple:", combined)  # Output: (1, 2, 3, 4, 5, 6)

repeated = (1, 2) * 3
print("Repeated tuple:", repeated)  # Output: (1, 2, 1, 2, 1, 2)

# Membership
print("Is 'banana' in fruits?", "banana" in fruits)  # Output: True

# Length
print("Number of fruits:", len(fruits))  # Output: 3

# Tuple unpacking
x, y = coordinates
print("Coordinates:", x, y)  # Output: 10.0 20.0

# Using tuples as dictionary keys
location = {
    (40.7128, -74.0060): "New York",
    (34.0522, -118.2437): "Los Angeles"
}
print("Location:", location[(40.7128, -74.0060)])  # Output: New York

# Tuple methods
numbers = (1, 2, 3, 1, 2, 1)
print("Count of 1:", numbers.count(1))  # Output: 3
print("Index of 2:", numbers.index(2))  # Output: 1
```

```
First fruit: apple
Last fruit: cherry
Sliced fruits: ('banana', 'cherry')
Combined tuple: (1, 2, 3, 4, 5, 6)
Repeated tuple: (1, 2, 1, 2, 1, 2)
Is 'banana' in fruits? True
Number of fruits: 3
Coordinates: 10.0 20.0
Location: New York
Count of 1: 3
Index of 2: 1
```

4. Dictionaries

Dictionaries are one of Python's most powerful and versatile data structures. They store data in **key-value pairs**, allowing you to quickly retrieve values based on their keys. Dictionaries are unordered (in Python versions before 3.7), mutable, and optimized for fast lookups.

What Are Dictionaries?

A **dictionary** is a collection of key-value pairs, where each key is unique. Dictionaries are defined using curly braces `{}` or the `dict()` constructor.

Example:

```python
# Using curly braces
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Using dict() constructor
person = dict(name="Alice", age=25, city="New York")
```

Accessing Dictionary Values

You can access values in a dictionary using their keys. If the key does not exist, Python will raise a `KeyError`. To avoid this, you can use the `get()` method, which returns `None` (or a default value) if the key is not found.

Example:

```python
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing values
print(person["name"])  # Output: Alice
print(person.get("age"))  # Output: 25

# Handling missing keys
print(person.get("country", "Unknown"))  # Output: Unknown
```

```
Alice
25
Unknown
```

Adding or Updating Dictionary Entries

You can add a new key-value pair or update an existing one by assigning a value to a key.

Example:

```python
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Adding a new key-value pair
person["country"] = "USA"

# Updating an existing key
person["age"] = 26

print(person)
# Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

```
{'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

Removing Dictionary Entries

You can remove key-value pairs using:

1. `del`: Deletes the key-value pair.

   ```python
   del person["city"]
   print(person)  # Output: {'name': 'Alice', 'age': 26, 'country': 'USA'}
   ```

   ```
   {'name': 'Alice', 'age': 26, 'country': 'USA'}
   ```

2. `pop()`: Removes the key-value pair and returns the value.

```
    age = person.pop("age")
    print(age)  # Output: 26
    print(person)  # Output: {'name': 'Alice', 'country': 'USA'}
```

```
26
{'name': 'Alice', 'country': 'USA'}
```

3. `popitem()` : Removes and returns the last inserted key-value pair (Python 3.7+).

```
    last_item = person.popitem()
    print(last_item)  # Output: ('country', 'USA')
    print(person)  # Output: {'name': 'Alice'}
```

```
('country', 'USA')
{'name': 'Alice'}
```

4. `clear()` : Removes all key-value pairs from the dictionary.

```
    person.clear()
    print(person)  # Output: {}
```

```
{}
```

Dictionary Methods

Here are some commonly used dictionary methods:

1. `keys()` : Returns a list of all keys.

```
    print(person.keys())  # Output: dict_keys(['name', 'age', 'city'])
```

```
dict_keys(['name', 'age', 'city', 'country'])
```

2. `values()` : Returns a list of all values.

```
    print(person.values())  # Output: dict_values(['Alice', 25, 'New York'])
```

```
dict_values(['Alice', 26, 'New York', 'USA'])
```

3. `items()` : Returns a list of key-value pairs as tuples.

```
   print(person.items())  # Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])
```

```
dict_items([('name', 'Alice'), ('age', 26), ('city', 'New York'), ('country', 'USA')])
```

4. `update()` : Merges another dictionary into the current one.

```
   person.update({"country": "USA", "age": 26})
   print(person)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

```
{'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

5. `copy()` : Returns a shallow copy of the dictionary.

```
   person_copy = person.copy()
   print(person_copy)
```

```
{'name': 'Alice', 'age': 26, 'city': 'New York', 'country': 'USA'}
```

Dictionary Comprehensions

Similar to list comprehensions, dictionary comprehensions allow you to create dictionaries in a concise way.

Syntax:

```
{key_expression: value_expression for item in iterable}
```

Example:

```
# Create a dictionary of squares
squares = {x: x ** 2 for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Nested Dictionaries

Dictionaries can contain other dictionaries, allowing you to create complex data structures.

Example:

```
students = {
    "Alice": {"age": 25, "grade": "A"},
    "Bob": {"age": 22, "grade": "B"},
    "Charlie": {"age": 23, "grade": "C"}
}

# Accessing nested dictionary values
print(students["Alice"]["age"])  # Output: 25
```

```
25
```

Example Program

```
# Working with Dictionaries

# Creating a dictionary
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Accessing values
print("Name:", person["name"])  # Output: Alice
print("Age:", person.get("age"))  # Output: 25

# Adding/updating entries
person["country"] = "USA"
person["age"] = 26

# Removing entries
del person["city"]
age = person.pop("age")
last_item = person.popitem()
```

```python
# Dictionary methods
print("Keys:", person.keys())  # Output: dict_keys(['name'])
print("Values:", person.values())  # Output: dict_values(['Alice'])
print("Items:", person.items())  # Output: dict_items([('name', 'Alice')])

# Dictionary comprehension
squares = {x: x ** 2 for x in range(1, 6)}
print("Squares:", squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Nested dictionaries
students = {
    "Alice": {"age": 25, "grade": "A"},
    "Bob": {"age": 22, "grade": "B"},
    "Charlie": {"age": 23, "grade": "C"}
}
print("Alice's age:", students["Alice"]["age"])  # Output: 25
```

```
Name: Alice
Age: 25
Keys: dict_keys(['name'])
Values: dict_values(['Alice'])
Items: dict_items([('name', 'Alice')])
Squares: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
Alice's age: 25
```

---

5. Sets

A **set** is an unordered collection of unique elements. Sets are useful for tasks that involve membership testing, removing duplicates, and performing mathematical operations like unions, intersections, and differences.

What Are Sets?

- Sets are defined using curly braces `{}` or the `set()` constructor.
- Sets do not allow duplicate elements. If you try to add a duplicate, it will be ignored.
- Sets are unordered, meaning the elements are not stored in any specific order.

Example:

```python
# Using curly braces
fruits = {"apple", "banana", "cherry"}
```

```python
# Using set() constructor
numbers = set([1, 2, 3, 4, 5])
```

Creating Sets

You can create a set by enclosing elements in curly braces `{}` or by passing an iterable (e.g., a list) to the `set()` constructor.

Example:

```python
# Creating a set with curly braces
fruits = {"apple", "banana", "cherry"}

# Creating a set with set() constructor
numbers = set([1, 2, 3, 4, 5])

# Creating an empty set
empty_set = set()  # Note: {} creates an empty dictionary, not a set
```

Adding and Removing Elements

1. **Adding Elements**: Use the `add()` method to add a single element or the `update()` method to add multiple elements.

```python
   fruits = {"apple", "banana", "cherry"}

   # Add a single element
   fruits.add("orange")

   # Add multiple elements
   fruits.update(["mango", "grape"])

   print(fruits)  # Output: {'apple', 'banana', 'cherry', 'orange', 'mango', 'grape'}
```

```
{'orange', 'cherry', 'apple', 'banana', 'mango', 'grape'}
```

2. **Removing Elements**:
   - `remove()` : Removes a specific element. Raises a `KeyError` if the element is not found.

```
    fruits.remove("banana")
    print(fruits)  # Output: {'apple', 'cherry', 'orange', 'mango',
'grape'}
```

```
{'orange', 'cherry', 'apple', 'mango', 'grape'}
```

- `discard()` : Removes a specific element if it exists. Does not raise an error if the element is not found.

```
    fruits.discard("banana")  # No error if "banana" is not in the set
```

- `clear()` : Removes all elements from the set.

```
    fruits.clear()
    print(fruits)  # Output: set()
```

```
set()
```

Set Operations

Sets support mathematical operations like unions, intersections, differences, and symmetric differences.

1. **Union (`|`)**: Combines elements from two sets.

```
    set1 = {1, 2, 3}
    set2 = {3, 4, 5}
    union_set = set1 | set2
    print(union_set)  # Output: {1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
```

2. **Intersection (`&`)**: Returns elements common to both sets.

```
    intersection_set = set1 & set2
    print(intersection_set)  # Output: {3}
```

```
{3}
```

3. **Difference (`-`)**: Returns elements in the first set that are not in the second set.

```
difference_set = set1 - set2
print(difference_set)  # Output: {1, 2}
```

```
{1, 2}
```

4. **Symmetric Difference (`^`)**: Returns elements that are in either set but not in both.

```
symmetric_difference_set = set1 ^ set2
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

```
{1, 2, 4, 5}
```

Set Methods

Here are some commonly used set methods:

1. `len()` : Returns the number of elements in the set.

```
print(len(fruits))  # Output: 5
```

```
5
```

2. `in` : Checks if an element exists in the set.

```
print("apple" in fruits)  # Output: True
```

```
True
```

3. `issubset()` : Checks if one set is a subset of another.

```
set1 = {1, 2}
set2 = {1, 2, 3, 4}
print(set1.issubset(set2))  # Output: True
```

```
True
```

4. `issuperset()` : Checks if one set is a superset of another.

```
    print(set2.issuperset(set1))  # Output: True
```

```
True
```

5. `isdisjoint()` : Checks if two sets have no common elements.

```
    set3 = {5, 6}
    print(set1.isdisjoint(set3))  # Output: True
```

```
True
```

Set Comprehensions

Similar to list comprehensions, set comprehensions allow you to create sets in a concise way.

Syntax:

```
{expression for item in iterable if condition}
```

Example:

```
# Create a set of squares
squares = {x ** 2 for x in range(1, 6)}
print(squares)  # Output: {1, 4, 9, 16, 25}
```

```
{1, 4, 9, 16, 25}
```

Example Program

```
# Working with Sets

# Creating sets
fruits = {"apple", "banana", "cherry"}
numbers = set([1, 2, 3, 4, 5])
```

```python
# Adding elements
fruits.add("orange")
fruits.update(["mango", "grape"])

# Removing elements
fruits.remove("banana")
fruits.discard("banana")  # No error if "banana" is not in the set
removed_fruit = fruits.pop()
fruits.clear()

# Set operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
intersection_set = set1 & set2
difference_set = set1 - set2
symmetric_difference_set = set1 ^ set2

# Set methods
print(len(fruits))  # Output: 0
print("apple" in fruits)  # Output: False
print(set1.issubset(set2))  # Output: False
print(set2.issuperset(set1))  # Output: False
print(set1.isdisjoint(set2))  # Output: False

# Set comprehension
squares = {x ** 2 for x in range(1, 6)}
print("Squares:", squares)  # Output: {1, 4, 9, 16, 25}
```

```
0
False
False
False
False
Squares: {1, 4, 9, 16, 25}
```

---

6. Strings

Strings are one of the most commonly used data types in Python. They are used to represent text and are defined using single quotes `' '`, double quotes `" "`, or triple quotes `''' '''` or `""" """`. In this section, we'll explore advanced string operations, formatting, and manipulation techniques.

What Are Strings?

A **string** is a sequence of characters. Strings are immutable, meaning once a string is created, it cannot be changed. However, you can create new strings based on existing ones.

Example:

```
# Using single quotes
message1 = 'Hello, World!'

# Using double quotes
message2 = "Python is fun!"

# Using triple quotes for multi-line strings
message3 = """This is a
multi-line string."""
```

Accessing String Characters

You can access individual characters in a string using indexing. Python uses **zero-based indexing**, meaning the first character has an index of `0`.

Example:

```
text = "Python"

# Accessing characters
print(text[0])  # Output: P
print(text[3])  # Output: h

# Negative indexing (starts from the end)
print(text[-1])  # Output: n
```

```
P
h
n
```

String Slicing

You can extract a substring using slicing. The syntax is `[start:stop:step]`.

Example:

```python
text = "Python Programming"

# Extract "Python"
print(text[0:6])  # Output: Python

# Extract "Programming"
print(text[7:18])  # Output: Programming

# Extract every second character
print(text[::2])  # Output: Pto rgamn
```

```
Python
Programming
Pto rgamn
```

String Methods

Python provides many built-in methods for working with strings. Here are some commonly used ones:

1. `upper()` : Converts the string to uppercase.

   ```python
   print("hello".upper())  # Output: HELLO
   ```

```
HELLO
```

2. `lower()` : Converts the string to lowercase.

   ```python
   print("HELLO".lower())  # Output: hello
   ```

```
hello
```

3. `strip()` : Removes leading and trailing whitespace.

   ```python
   print("  hello  ".strip())  # Output: hello
   ```

```
hello
```

4. `replace()` : Replaces a substring with another substring.

```
    print("hello world".replace("world", "Python"))  # Output: hello Python
```

```
hello Python
```

5. `split()` : Splits the string into a list of substrings based on a delimiter.

```
    print("apple,banana,cherry".split(","))  # Output: ['apple', 'banana',
'cherry']
```

```
['apple', 'banana', 'cherry']
```

6. `join()` : Joins a list of strings into a single string using a delimiter.

```
    print(", ".join(["apple", "banana", "cherry"]))  # Output: apple, banana,
cherry
```

```
apple, banana, cherry
```

7. `find()` : Returns the index of the first occurrence of a substring. Returns `-1` if not found.

```
    print("hello world".find("world"))  # Output: 6
```

```
6
```

8. `count()` : Returns the number of occurrences of a substring.

```
    print("hello world".count("l"))  # Output: 3
```

```
3
```

9. `startswith()` : Checks if the string starts with a specific substring.

```
    print("hello world".startswith("hello"))  # Output: True
```

```
True
```

10. `endswith()` : Checks if the string ends with a specific substring.

```python
print("hello world".endswith("world"))  # Output: True
```

```
True
```

String Formatting

Python provides several ways to format strings:

1. **f-strings** (Python 3.6+): Embed expressions inside string literals.

```python
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

```
My name is Alice and I am 25 years old.
```

2. `format()` **method**: Insert values into placeholders `{}`.

```python
print("My name is {} and I am {} years old.".format(name, age))
```

```
My name is Alice and I am 25 years old.
```

3. `%` **operator** (older style):

```python
print("My name is %s and I am %d years old." % (name, age))
```

```
My name is Alice and I am 25 years old.
```

Escape Characters

Escape characters are used to include special characters in strings:

- `\n` : Newline
- `\t` : Tab
- `\\` : Backslash
- `\"` : Double quote
- `\'` : Single quote

Example:

```
print("Hello\nWorld")  # Output: Hello
                       #         World
```

```
Hello
World
```

Raw Strings

Raw strings ignore escape characters and treat backslashes as literal characters. They are prefixed with an `r`.

Example:

```
print(r"C:\Users\Alice\Documents")  # Output: C:\Users\Alice\Documents
```

```
C:\Users\Alice\Documents
```

String Membership

You can check if a substring exists in a string using the `in` keyword.

Example:

```
text = "Python is fun"
print("fun" in text)  # Output: True
```

```
True
```

Example Program

```
# Working with Strings

# Creating strings
message1 = 'Hello, World!'
message2 = "Python is fun!"
```

```python
message3 = """This is a
multi-line string."""

# Accessing characters
text = "Python"
print("First character:", text[0])  # Output: P
print("Last character:", text[-1])  # Output: n

# String slicing
print("Sliced string:", text[0:4])  # Output: Pyth
print("Every second character:", text[::2])  # Output: Pto

# String methods
print("Uppercase:", "hello".upper())  # Output: HELLO
print("Lowercase:", "HELLO".lower())  # Output: hello
print("Stripped:", "  hello  ".strip())  # Output: hello
print("Replaced:", "hello world".replace("world", "Python"))  # Output:
hello Python
print("Split:", "apple,banana,cherry".split(","))  # Output: ['apple',
'banana', 'cherry']
print("Joined:", ", ".join(["apple", "banana", "cherry"]))  # Output: apple,
banana, cherry
print("Found at index:", "hello world".find("world"))  # Output: 6
print("Count of 'l':", "hello world".count("l"))  # Output: 3
print("Starts with 'hello':", "hello world".startswith("hello"))  # Output:
True
print("Ends with 'world':", "hello world".endswith("world"))  # Output: True

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

# Escape characters
print("Hello\nWorld")  # Output: Hello
                      #         World

# Raw strings
print(r"C:\Users\Alice\Documents")  # Output: C:\Users\Alice\Documents

# String membership
print("Is 'fun' in the text?", "fun" in "Python is fun")  # Output: True
```

```
First character: P
Last character: n
Sliced string: Pyth
Every second character: Pto
Uppercase: HELLO
Lowercase: hello
Stripped: hello
Replaced: hello Python
Split: ['apple', 'banana', 'cherry']
Joined: apple, banana, cherry
Found at index: 6
Count of 'l': 3
Starts with 'hello': True
Ends with 'world': True
My name is Alice and I am 25 years old.
My name is Alice and I am 25 years old.
My name is Alice and I am 25 years old.
Hello
World
C:\Users\Alice\Documents
Is 'fun' in the text? True
```

---

7. Collections

The `collections` **module** in Python provides specialized container data types that are alternatives to the built-in types like `list`, `tuple`, `dict`, and `set`. These data types are optimized for specific use cases and can make your code more efficient and readable.

What is the `collections` Module?

The `collections` module includes the following data structures:

1. **`namedtuple`**: Creates tuple-like objects with named fields.
2. **`deque`**: A double-ended queue for fast appends and pops.
3. **`Counter`**: A dictionary subclass for counting hashable objects.
4. **`defaultdict`**: A dictionary subclass that provides default values for missing keys.
5. **`OrderedDict`**: A dictionary subclass that remembers the order of insertion (less relevant in Python 3.7+, where regular dictionaries are ordered).
6. **`ChainMap`**: Combines multiple dictionaries into a single mapping.

---

1. `namedtuple`

A `namedtuple` is a factory function for creating tuple-like objects with named fields. It makes code more readable by allowing access to elements by name instead of index.

Syntax:

```python
from collections import namedtuple
NamedTuple = namedtuple("NamedTuple", ["field1", "field2", ...])
```

Example:

```python
from collections import namedtuple

# Define a namedtuple
Point = namedtuple("Point", ["x", "y"])

# Create an instance
p = Point(10, 20)

# Access fields by name
print(p.x, p.y)  # Output: 10 20
```

```
10 20
```

---

2. `deque`

A `deque` (double-ended queue) is optimized for fast appends and pops from both ends. It is more efficient than a list for operations that involve adding or removing elements from the beginning.

Syntax:

```python
from collections import deque
d = deque([iterable])
```

Example:

```python
from collections import deque

# Create a deque
d = deque([1, 2, 3])
```

```
# Append to the right
d.append(4)  # deque([1, 2, 3, 4])

# Append to the left
d.appendleft(0)  # deque([0, 1, 2, 3, 4])

# Pop from the right
d.pop()  # Returns 4, deque([0, 1, 2, 3])

# Pop from the left
d.popleft()  # Returns 0, deque([1, 2, 3])
```

```
0
```

---

3. `Counter`

A `Counter` is a dictionary subclass for counting hashable objects. It is useful for tallying occurrences of elements in a collection.

Syntax:

```
from collections import Counter
c = Counter([iterable])
```

Example:

```
from collections import Counter

# Count occurrences of elements
c = Counter(["apple", "banana", "apple", "cherry", "banana", "apple"])

print(c)  # Output: Counter({'apple': 3, 'banana': 2, 'cherry': 1})

# Most common elements
print(c.most_common(2))  # Output: [('apple', 3), ('banana', 2)]
```

```
Counter({'apple': 3, 'banana': 2, 'cherry': 1})
[('apple', 3), ('banana', 2)]
```

---

4. `defaultdict`

A `defaultdict` is a dictionary subclass that provides default values for missing keys. It eliminates the need to check if a key exists before accessing it.

Syntax:

```python
from collections import defaultdict
d = defaultdict(default_factory)
```

Example:

```python
from collections import defaultdict

# Default value for missing keys is an empty list
d = defaultdict(list)

# Add elements
d["fruits"].append("apple")
d["fruits"].append("banana")

print(d)  # Output: defaultdict(<class 'list'>, {'fruits': ['apple',
'banana']})
```

```
defaultdict(<class 'list'>, {'fruits': ['apple', 'banana']})
```

---

5. `OrderedDict`

An `OrderedDict` is a dictionary subclass that remembers the order of insertion. In Python 3.7+, regular dictionaries are ordered by default, so this is less commonly needed.

Syntax:

```python
from collections import OrderedDict
od = OrderedDict([items])
```

Example:

```python
from collections import OrderedDict

# Create an OrderedDict
od = OrderedDict()
od["a"] = 1
od["b"] = 2
od["c"] = 3
```

```python
print(od)  # Output: OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

```
OrderedDict({'a': 1, 'b': 2, 'c': 3})
```

---

6. `ChainMap`

A `ChainMap` combines multiple dictionaries into a single mapping. It is useful for searching through multiple dictionaries as if they were one.

Syntax:

```python
from collections import ChainMap
cm = ChainMap(dict1, dict2, ...)
```

Example:

```python
from collections import ChainMap

# Create dictionaries
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

# Combine dictionaries
cm = ChainMap(dict1, dict2)

# Access values
print(cm["a"])  # Output: 1 (from dict1)
print(cm["b"])  # Output: 2 (from dict1, first match)
print(cm["c"])  # Output: 4 (from dict2)
```

```
1
2
4
```

Example Program

```python
# Working with Collections

# namedtuple
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
p = Point(10, 20)
print("Point:", p.x, p.y)  # Output: 10 20

# deque
from collections import deque
d = deque([1, 2, 3])
d.append(4)
d.appendleft(0)
print("Deque:", d)  # Output: deque([0, 1, 2, 3, 4])

# Counter
from collections import Counter
c = Counter(["apple", "banana", "apple", "cherry", "banana", "apple"])
print("Counter:", c)  # Output: Counter({'apple': 3, 'banana': 2, 'cherry':
1})
print("Most common:", c.most_common(2))  # Output: [('apple', 3), ('banana',
2)]

# defaultdict
from collections import defaultdict
d = defaultdict(list)
d["fruits"].append("apple")
d["fruits"].append("banana")
print("DefaultDict:", d)  # Output: defaultdict(<class 'list'>, {'fruits':
['apple', 'banana']})

# OrderedDict
from collections import OrderedDict
od = OrderedDict()
od["a"] = 1
od["b"] = 2
od["c"] = 3
print("OrderedDict:", od)  # Output: OrderedDict([('a', 1), ('b', 2), ('c',
3)])

# ChainMap
from collections import ChainMap
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}
cm = ChainMap(dict1, dict2)
print("ChainMap:", cm["a"], cm["b"], cm["c"])  # Output: 1 2 4
```

```
Point: 10 20
Deque: deque([0, 1, 2, 3, 4])
Counter: Counter({'apple': 3, 'banana': 2, 'cherry': 1})
Most common: [('apple', 3), ('banana', 2)]
DefaultDict: defaultdict(<class 'list'>, {'fruits': ['apple', 'banana']})
```

```
OrderedDict: OrderedDict({'a': 1, 'b': 2, 'c': 3})
ChainMap: 1 2 4
```

---

8. Itertools

The `itertools` **module** in Python provides a collection of tools for working with iterators. These tools are designed to be fast, memory-efficient, and easy to use. They are particularly useful for tasks involving iteration, combinations, permutations, and more.

What is the `itertools` Module?

The `itertools` module includes functions for:

- **Infinite iterators**: Generate infinite sequences.
- **Combinatoric iterators**: Generate combinations, permutations, and Cartesian products.
- **Terminating iterators**: Process finite iterables in useful ways.

Common `itertools` Functions

1. **Infinite Iterators**
    - `count()` : Generates an infinite sequence of numbers.

```
    import itertools

    for i in itertools.count(start=1, step=2):
        if i > 10:
            break
        print(i, end=" ")  # Output: 1 3 5 7 9
```

```
1 3 5 7 9
```

- `cycle()` : Cycles through an iterable infinitely.

```
    for item in itertools.cycle(["A", "B", "C"]):
        if item == "C":
            break
        print(item, end=" ")  # Output: A B
```

```
A B
```

- `repeat()` : Repeats an element infinitely or a specified number of times.

```
    for item in itertools.repeat("Python", 3):
        print(item, end=" ")  # Output: Python Python Python
```

```
Python Python Python
```

2. **Combinatoric Iterators**
    - `product()` : Computes the Cartesian product of input iterables.

```
    for item in itertools.product("AB", repeat=2):
        print(item, end=" ")  # Output: ('A', 'A') ('A', 'B') ('B', 'A')
('B', 'B')
```

```
('A', 'A') ('A', 'B') ('B', 'A') ('B', 'B')
```

- `permutations()` : Generates all possible permutations of an iterable.

```
    for item in itertools.permutations("ABC", 2):
        print(item, end=" ")  # Output: ('A', 'B') ('A', 'C') ('B', 'A')
('B', 'C') ('C', 'A') ('C', 'B')
```

```
('A', 'B') ('A', 'C') ('B', 'A') ('B', 'C') ('C', 'A') ('C', 'B')
```

- `combinations()` : Generates all possible combinations of an iterable.

```
    for item in itertools.combinations("ABC", 2):
        print(item, end=" ")  # Output: ('A', 'B') ('A', 'C') ('B', 'C')
```

```
('A', 'B') ('A', 'C') ('B', 'C')
```

- `combinations_with_replacement()` : Generates combinations with repeated elements.

```
    for item in itertools.combinations_with_replacement("ABC", 2):
        print(item, end=" ")  # Output: ('A', 'A') ('A', 'B') ('A', 'C')
('B', 'B') ('B', 'C') ('C', 'C')
```

```
('A', 'A') ('A', 'B') ('A', 'C') ('B', 'B') ('B', 'C') ('C', 'C')
```

3. **Terminating Iterators**
    - `accumulate()` : Returns accumulated sums or results of a binary function.

```
    for item in itertools.accumulate([1, 2, 3, 4]):
        print(item, end=" ")  # Output: 1 3 6 10
```

```
1 3 6 10
```

- `chain()` : Chains multiple iterables together.

```
    for item in itertools.chain("ABC", "DEF"):
        print(item, end=" ")  # Output: A B C D E F
```

```
A B C D E F
```

- `compress()` : Filters elements using a boolean mask.

```
    for item in itertools.compress("ABCDEF", [1, 0, 1, 0, 1, 0]):
        print(item, end=" ")  # Output: A C E
```

```
A C E
```

- `dropwhile()` : Drops elements until a condition is false.

```
    for item in itertools.dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]):
        print(item, end=" ")  # Output: 6 4 1
```

```
6 4 1
```

- `takewhile()` : Takes elements until a condition is false.

```
    for item in itertools.takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]):
        print(item, end=" ")  # Output: 1 4
```

```
1 4
```

- `groupby()` : Groups elements by a key function.

```
    data = [("A", 1), ("A", 2), ("B", 3), ("B", 4)]
    for key, group in itertools.groupby(data, lambda x: x[0]):
        print(key, list(group))
    # Output:
    # A [('A', 1), ('A', 2)]
    # B [('B', 3), ('B', 4)]
```

```
A [('A', 1), ('A', 2)]
B [('B', 3), ('B', 4)]
```

Example Program

```
# Working with Itertools

# Infinite iterators
import itertools

print("Count:")
for i in itertools.count(start=1, step=2):
    if i > 10:
        break
    print(i, end=" ")  # Output: 1 3 5 7 9

print("\nCycle:")
for item in itertools.cycle(["A", "B", "C"]):
    if item == "C":
        break
    print(item, end=" ")  # Output: A B

print("\nRepeat:")
for item in itertools.repeat("Python", 3):
    print(item, end=" ")  # Output: Python Python Python

# Combinatoric iterators
print("\nProduct:")
for item in itertools.product("AB", repeat=2):
    print(item, end=" ")  # Output: ('A', 'A') ('A', 'B') ('B', 'A') ('B',
'B')

print("\nPermutations:")
for item in itertools.permutations("ABC", 2):
    print(item, end=" ")  # Output: ('A', 'B') ('A', 'C') ('B', 'A') ('B',
'C') ('C', 'A') ('C', 'B')

print("\nCombinations:")
```

```python
for item in itertools.combinations("ABC", 2):
    print(item, end=" ")  # Output: ('A', 'B') ('A', 'C') ('B', 'C')

print("\nCombinations with Replacement:")
for item in itertools.combinations_with_replacement("ABC", 2):
    print(item, end=" ")  # Output: ('A', 'A') ('A', 'B') ('A', 'C') ('B',
'B') ('B', 'C') ('C', 'C')

# Terminating iterators
print("\nAccumulate:")
for item in itertools.accumulate([1, 2, 3, 4]):
    print(item, end=" ")  # Output: 1 3 6 10

print("\nChain:")
for item in itertools.chain("ABC", "DEF"):
    print(item, end=" ")  # Output: A B C D E F

print("\nCompress:")
for item in itertools.compress("ABCDEF", [1, 0, 1, 0, 1, 0]):
    print(item, end=" ")  # Output: A C E

print("\nDropwhile:")
for item in itertools.dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]):
    print(item, end=" ")  # Output: 6 4 1

print("\nTakewhile:")
for item in itertools.takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]):
    print(item, end=" ")  # Output: 1 4

print("\nGroupby:")
data = [("A", 1), ("A", 2), ("B", 3), ("B", 4)]
for key, group in itertools.groupby(data, lambda x: x[0]):
    print(key, list(group))
# Output:
# A [('A', 1), ('A', 2)]
# B [('B', 3), ('B', 4)]
```

```
Count:
1 3 5 7 9
Cycle:
A B
Repeat:
Python Python Python
Product:
('A', 'A') ('A', 'B') ('B', 'A') ('B', 'B')
Permutations:
('A', 'B') ('A', 'C') ('B', 'A') ('B', 'C') ('C', 'A') ('C', 'B')
Combinations:
('A', 'B') ('A', 'C') ('B', 'C')
```

```
Combinations with Replacement:
('A', 'A') ('A', 'B') ('A', 'C') ('B', 'B') ('B', 'C') ('C', 'C')
Accumulate:
1 3 6 10
Chain:
A B C D E F
Compress:
A C E
Dropwhile:
6 4 1
Takewhile:
1 4
Groupby:
A [('A', 1), ('A', 2)]
B [('B', 3), ('B', 4)]
```

---

9. Lambda Functions

Lambda functions (also called **anonymous functions**) are small, inline functions defined using the `lambda` keyword. They are useful for short, throwaway functions that are used only once or in situations where defining a full function using `def` would be overkill.

What Are Lambda Functions?

A lambda function is a function without a name. It can take any number of arguments but can only have one expression. The result of the expression is automatically returned.

Syntax:

```
lambda arguments: expression
```

Example of a Lambda Function

Here's a simple lambda function that adds two numbers:

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

```
8
```

When to Use Lambda Functions

Lambda functions are typically used in situations where a small function is needed for a short period of time, such as:

- As an argument to higher-order functions like `map()`, `filter()`, and `sorted()`.
- For simple transformations or calculations.

Using Lambda Functions with `map()`

The `map()` function applies a function to all items in an iterable. Lambda functions are often used with `map()` for concise transformations.

Example:

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

Using Lambda Functions with `filter()`

The `filter()` function filters elements from an iterable based on a condition. Lambda functions are often used with `filter()` for concise filtering.

Example:

```
numbers = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4]
```

```
[2, 4]
```

Using Lambda Functions with `sorted()`

The `sorted()` function sorts an iterable. Lambda functions can be used to define custom sorting keys.

Example:

```
students = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 22},
    {"name": "Charlie", "age": 23}
]

# Sort by age
sorted_students = sorted(students, key=lambda x: x["age"])
print(sorted_students)
# Output: [{'name': 'Bob', 'age': 22}, {'name': 'Charlie', 'age': 23},
{'name': 'Alice', 'age': 25}]
```

```
[{'name': 'Bob', 'age': 22}, {'name': 'Charlie', 'age': 23}, {'name':
'Alice', 'age': 25}]
```

Lambda Functions in List Comprehensions

Lambda functions can also be used in list comprehensions for concise transformations.

Example:

```
numbers = [1, 2, 3, 4, 5]
squared = [(lambda x: x ** 2)(x) for x in numbers]
print(squared)  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

Limitations of Lambda Functions

- **Single Expression**: Lambda functions can only contain a single expression. They cannot include statements like `if`, `for`, or `while`.
- **Readability**: Overusing lambda functions can make code harder to read. For complex logic, it's better to use a regular function defined with `def`.

Example Program

```
# Working with Lambda Functions

# Basic lambda function
add = lambda x, y: x + y
print("Add:", add(3, 5))  # Output: 8

# Using lambda with map()
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print("Squared:", squared)  # Output: [1, 4, 9, 16, 25]

# Using lambda with filter()
evens = list(filter(lambda x: x % 2 == 0, numbers))
print("Evens:", evens)  # Output: [2, 4]

# Using lambda with sorted()
students = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 22},
    {"name": "Charlie", "age": 23}
]
sorted_students = sorted(students, key=lambda x: x["age"])
print("Sorted Students:", sorted_students)
# Output: [{'name': 'Bob', 'age': 22}, {'name': 'Charlie', 'age': 23},
{'name': 'Alice', 'age': 25}]

# Using lambda in list comprehensions
squared = [(lambda x: x ** 2)(x) for x in numbers]
print("Squared (List Comprehension):", squared)  # Output: [1, 4, 9, 16, 25]
```

```
Add: 8
Squared: [1, 4, 9, 16, 25]
Evens: [2, 4]
Sorted Students: [{'name': 'Bob', 'age': 22}, {'name': 'Charlie', 'age':
23}, {'name': 'Alice', 'age': 25}]
Squared (List Comprehension): [1, 4, 9, 16, 25]
```

---

10. Exceptions and Errors

In Python, **exceptions** are events that occur during the execution of a program that disrupt the normal flow of instructions. When an exception occurs, Python raises an **error**, which can be caught and handled to prevent the program from crashing.

Types of Errors

1. **Syntax Errors**: Occur when the code violates Python's syntax rules. These are detected before the program runs.

```
print("Hello, World"  # Missing closing parenthesis
```

2. **Runtime Errors (Exceptions)**: Occur during the execution of the program. Examples include:
   - `ZeroDivisionError` : Division by zero.
   - `TypeError` : Performing an operation on incompatible types.
   - `ValueError` : Passing an invalid value to a function.
   - `FileNotFoundError` : Trying to open a file that doesn't exist.

Handling Exceptions with `try/except`

To handle exceptions, use a `try/except` block. The `try` block contains the code that might raise an exception, and the `except` block contains the code to handle the exception.

Syntax:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Code to handle the exception
```

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero!")
```

```
Error: Division by zero!
```

Handling Multiple Exceptions

You can handle multiple exceptions by specifying multiple `except` blocks or using a tuple.

Example:

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Error: Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Error: Division by zero.")
```

```
Enter a number:  a


Error: Invalid input. Please enter a number.
```

The `else` Block

The `else` block is executed if no exceptions occur in the `try` block. It is useful for code that should only run if the `try` block succeeds.

Example:

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Error: Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Error: Division by zero.")
else:
    print("Result:", result)
```

```
Enter a number:  2


Result: 5.0
```

The `finally` Block

The `finally` block is executed no matter what—whether an exception occurs or not. It is typically used for cleanup actions, such as closing files or releasing resources.

Example:

```python
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
finally:
    file.close()
    print("File closed.")
```

```
Hello, World!
File closed.
```

Raising Exceptions

You can raise exceptions manually using the `raise` keyword. This is useful for enforcing constraints or signaling errors in your code.

Example:

```python
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)  # Output: Cannot divide by zero.
```

```
Cannot divide by zero.
```

Custom Exceptions

You can define your own exceptions by creating a new class that inherits from Python's built-in `Exception` class.

Example:

```python
class NegativeNumberError(Exception):
    pass

def check_positive(number):
    if number < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")

try:
    check_positive(-5)
except NegativeNumberError as e:
    print(e)  # Output: Negative numbers are not allowed.
```

```
Negative numbers are not allowed.
```

Example Program

```python
# Working with Exceptions and Errors

# Handling exceptions
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Error: Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Error: Division by zero.")
else:
    print("Result:", result)
finally:
    print("Execution complete.")

# Raising exceptions
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)  # Output: Cannot divide by zero.

# Custom exceptions
class NegativeNumberError(Exception):
    pass
```

```python
def check_positive(number):
    if number < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")

try:
    check_positive(-5)
except NegativeNumberError as e:
    print(e)  # Output: Negative numbers are not allowed.
```

```
Enter a number:  3


Result: 3.3333333333333335
Execution complete.
Cannot divide by zero.
Negative numbers are not allowed.
```

---

11. Logging

Logging is a way to track events that occur during the execution of a program. It is essential for debugging, monitoring, and understanding the flow of your application. Python provides a built-in `logging` module that makes it easy to add logging to your code.

Why Use Logging?

- **Debugging**: Log messages can help you identify and fix issues in your code.
- **Monitoring**: Logs provide insights into the behavior of your application in production.
- **Auditing**: Logs can be used to track user actions and system events.

Logging Levels

The `logging` module provides several levels of logging, each representing the severity of the event being logged. The levels, in increasing order of severity, are:

1. `DEBUG` : Detailed information for debugging.
2. `INFO` : General information about the program's execution.
3. `WARNING` : Indicates a potential issue that doesn't prevent the program from running.
4. `ERROR` : Indicates a more serious issue that may prevent part of the program from functioning.

5. `CRITICAL` : Indicates a critical issue that may prevent the entire program from functioning.

Basic Logging

To use the `logging` module, you first need to configure it. By default, the `logging` module logs messages with a severity level of `WARNING` or higher.

Example:

```python
import logging

# Basic logging
logging.warning("This is a warning message.")
logging.error("This is an error message.")
logging.critical("This is a critical message.")
```

```
WARNING:root:This is a warning message.
ERROR:root:This is an error message.
CRITICAL:root:This is a critical message.
```

Configuring Logging

You can configure the logging module to change the logging level, format, and output destination.

Example:

```python
import logging

# Configure logging
logging.basicConfig(
    level=logging.DEBUG,  # Set the logging level
    format="%(asctime)s - %(levelname)s - %(message)s",  # Set the log
format
    filename="app.log",  # Log to a file
    filemode="w"  # Overwrite the log file each time
)

# Log messages
logging.debug("This is a debug message.")
logging.info("This is an info message.")
logging.warning("This is a warning message.")
logging.error("This is an error message.")
logging.critical("This is a critical message.")
```

```
WARNING:root:This is a warning message.
ERROR:root:This is an error message.
CRITICAL:root:This is a critical message.
```

- The log messages will be written to `app.log` in the specified format.

Logging to Console and File

You can configure logging to output messages to both the console and a file using `handlers`.

Example:

```
import logging

# Create a logger
logger = logging.getLogger("my_logger")
logger.setLevel(logging.DEBUG)

# Create a console handler
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)

# Create a file handler
file_handler = logging.FileHandler("app.log")
file_handler.setLevel(logging.DEBUG)

# Create a formatter
formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

# Add the formatter to the handlers
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

# Add the handlers to the logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)

# Log messages
logger.debug("This is a debug message.")
logger.info("This is an info message.")
logger.warning("This is a warning message.")
logger.error("This is an error message.")
logger.critical("This is a critical message.")
```

```
DEBUG:my_logger:This is a debug message.
INFO:my_logger:This is an info message.
2025-02-17 02:33:57,533 - WARNING - This is a warning message.
WARNING:my_logger:This is a warning message.
2025-02-17 02:33:57,535 - ERROR - This is an error message.
ERROR:my_logger:This is an error message.
2025-02-17 02:33:57,539 - CRITICAL - This is a critical message.
CRITICAL:my_logger:This is a critical message.
```

- **File Output** (`app.log`):

```
with open('app.log', 'r') as file:
    content = file.read()
    print(content)
```

```
2025-02-17 02:33:57,514 - DEBUG - This is a debug message.
2025-02-17 02:33:57,531 - INFO - This is an info message.
2025-02-17 02:33:57,533 - WARNING - This is a warning message.
2025-02-17 02:33:57,535 - ERROR - This is an error message.
2025-02-17 02:33:57,539 - CRITICAL - This is a critical message.
```

Logging Exceptions

You can log exceptions using the `logging.exception()` method, which automatically includes the exception traceback.

Example:

```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG, format="%(asctime)s - %(levelname)s
- %(message)s")

try:
    result = 10 / 0
except ZeroDivisionError:
    logging.exception("An error occurred:")
```

```
ERROR:root:An error occurred:
Traceback (most recent call last):
  File "C:\Users\attila\AppData\Local\Temp\ipykernel_10184\3197086528.py",
line 7, in <module>
    result = 10 / 0
             ~~~^~~
ZeroDivisionError: division by zero
```

Example Program

```python
# Working with Logging

import logging

# Configure logging
logging.basicConfig(
    level=logging.DEBUG,
    format="%(asctime)s - %(levelname)s - %(message)s",
    filename="app.log",
    filemode="w"
)

# Log messages
logging.debug("This is a debug message.")
logging.info("This is an info message.")
logging.warning("This is a warning message.")
logging.error("This is an error message.")
logging.critical("This is a critical message.")

# Logging exceptions
try:
    result = 10 / 0
except ZeroDivisionError:
    logging.exception("An error occurred:")
```

```
WARNING:root:This is a warning message.
ERROR:root:This is an error message.
CRITICAL:root:This is a critical message.
ERROR:root:An error occurred:
Traceback (most recent call last):
  File "C:\Users\attila\AppData\Local\Temp\ipykernel_10184\351276769.py",
line 22, in <module>
    result = 10 / 0
```

```
             ~~~^~~
ZeroDivisionError: division by zero
```

---

12. JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is widely used for transmitting data between a server and a web application, as well as for configuration files and data storage.

What is JSON?

JSON is a text format that represents data as key-value pairs. It is based on a subset of JavaScript but is language-independent. JSON data is often stored in `.json` files or transmitted as strings.

Example JSON:

```json
{
    "name": "Alice",
    "age": 25,
    "is_student": false,
    "courses": ["Math", "Science"],
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
```

JSON Data Types

JSON supports the following data types:

- **Strings**: Enclosed in double quotes (`" "`).
- **Numbers**: Integers or floating-point numbers.
- **Booleans**: `true` or `false` .
- **Arrays**: Ordered lists of values, enclosed in square brackets (`[]`).
- **Objects**: Unordered collections of key-value pairs, enclosed in curly braces (`{}`).
- `null` : Represents an empty or non-existent value.

Working with JSON in Python

Python provides the `json` module to encode and decode JSON data. The two main functions are:

- `json.dumps()` : Converts a Python object to a JSON-formatted string.
- `json.loads()` : Converts a JSON-formatted string to a Python object.

Encoding Python Objects to JSON

Use `json.dumps()` to convert a Python object (e.g., dictionary, list) to a JSON string.

Example:

```python
import json

# Python dictionary
data = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "courses": ["Math", "Science"],
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}

# Convert to JSON string
json_string = json.dumps(data, indent=4)  # indent for pretty printing
print(json_string)
```

```
{
    "name": "Alice",
    "age": 25,
    "is_student": false,
    "courses": [
        "Math",
        "Science"
    ],
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
```

Decoding JSON to Python Objects

Use `json.loads()` to convert a JSON string to a Python object.

Example:

```python
import json

# JSON string
json_string = '''
{
    "name": "Alice",
    "age": 25,
    "is_student": false,
    "courses": ["Math", "Science"],
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}
'''

# Convert to Python dictionary
data = json.loads(json_string)
print(data)
```

```
{'name': 'Alice', 'age': 25, 'is_student': False, 'courses': ['Math',
'Science'], 'address': {'city': 'New York', 'zip': '10001'}}
```

Reading and Writing JSON Files

You can read JSON data from a file and write JSON data to a file using the `json.load()` and `json.dump()` functions.

1. **Reading from a JSON File**:

```python
import json

# Read JSON data from a file
```

```
    with open("data.json", "r") as file:
        data = json.load(file)
        print(data)
```

```
{'name': 'Alice', 'age': 25, 'is_student': False, 'courses': ['Math',
'Science'], 'address': {'city': 'New York', 'zip': '10001'}}
```

2. **Writing to a JSON File**:

```
    import json

    # Python dictionary
    data = {
        "name": "Attila",
        "age": 23,
        "is_student": False,
        "courses": ["Math", "Statics"],
        "address": {
            "city": "Urmia",
            "zip": "50708"
        }
    }

    # Write JSON data to a file
    with open("data.json", "w") as file:
        json.dump(data, file, indent=4)
```

Handling Custom Objects

By default, the `json` module cannot serialize custom Python objects. To handle this, you can define a custom encoder by subclassing `json.JSONEncoder` or by using the `default` parameter in `json.dumps()` .

Example:

```
import json
from datetime import datetime

# Custom object
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```python
# Custom encoder function
def person_encoder(obj):
    if isinstance(obj, Person):
        return {"name": obj.name, "age": obj.age}
    raise TypeError(f"Object of type {type(obj)} is not JSON serializable")

# Create a Person object
person = Person("Alice", 25)

# Convert to JSON string
json_string = json.dumps(person, default=person_encoder, indent=4)
print(json_string)
```

```
{
    "name": "Alice",
    "age": 25
}
```

Example Program

```python
# Working with JSON

import json

# Python dictionary
data = {
    "name": "Alice",
    "age": 25,
    "is_student": False,
    "courses": ["Math", "Science"],
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}

# Convert to JSON string
json_string = json.dumps(data, indent=4)
print("JSON String:")
print(json_string)

# Convert JSON string to Python dictionary
data_parsed = json.loads(json_string)
print("\nParsed Data:")
print(data_parsed)
```

```python
# Write JSON data to a file
with open("data.json", "w") as file:
    json.dump(data, file, indent=4)

# Read JSON data from a file
with open("data.json", "r") as file:
    data_from_file = json.load(file)
    print("\nData from File:")
    print(data_from_file)

# Handling custom objects
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def person_encoder(obj):
    if isinstance(obj, Person):
        return {"name": obj.name, "age": obj.age}
    raise TypeError(f"Object of type {type(obj)} is not JSON serializable")

person = Person("Alice", 25)
json_string_custom = json.dumps(person, default=person_encoder, indent=4)
print("\nCustom Object JSON String:")
print(json_string_custom)
```

```
JSON String:
{
    "name": "Alice",
    "age": 25,
    "is_student": false,
    "courses": [
        "Math",
        "Science"
    ],
    "address": {
        "city": "New York",
        "zip": "10001"
    }
}

Parsed Data:
{'name': 'Alice', 'age': 25, 'is_student': False, 'courses': ['Math',
'Science'], 'address': {'city': 'New York', 'zip': '10001'}}

Data from File:
{'name': 'Alice', 'age': 25, 'is_student': False, 'courses': ['Math',
'Science'], 'address': {'city': 'New York', 'zip': '10001'}}
```

```
Custom Object JSON String:
{
    "name": "Alice",
    "age": 25
}
```

---

13. Random Numbers

Generating random numbers is a common task in programming, whether for simulations, games, or security applications. Python provides the `random` module, which includes functions for generating random numbers, shuffling sequences, and selecting random elements.

The `random` Module

The `random` module is part of Python's standard library and provides various functions for working with randomness. To use it, you need to import the module:

```python
import random
```

Generating Random Numbers

1. **`random.random()`** : Generates a random float between 0.0 and 1.0.

   ```python
   print(random.random())  # Output: e.g., 0.3745401188473625
   ```

```
0.8484852847223121
```

2. **`random.uniform(a, b)`** : Generates a random float between `a` and `b`.

   ```python
   print(random.uniform(1.5, 4.5))  # Output: e.g., 2.345678901234567
   ```

```
2.384758360282017
```

3. **`random.randint(a, b)`** : Generates a random integer between `a` and `b` (inclusive).

```
print(random.randint(1, 10))  # Output: e.g., 7
```

```
8
```

4. `random.randrange(start, stop, step)` : Generates a random integer from a range.

```
print(random.randrange(0, 100, 5))  # Output: e.g., 45
```

```
65
```

Selecting Random Elements

1. `random.choice(seq)` : Selects a random element from a sequence (e.g., list, tuple, string).

```
fruits = ["apple", "banana", "cherry"]
print(random.choice(fruits))  # Output: e.g., "banana"
```

```
banana
```

2. `random.choices(seq, k=n)` : Selects `n` random elements from a sequence (with replacement).

```
print(random.choices(fruits, k=2))  # Output: e.g., ["cherry", "apple"]
```

```
['cherry', 'apple']
```

3. `random.sample(seq, k=n)` : Selects `n` unique random elements from a sequence (without replacement).

```
print(random.sample(fruits, 2))  # Output: e.g., ["banana", "cherry"]
```

```
['apple', 'banana']
```

Shuffling Sequences

4. `random.shuffle(seq)` : Shuffles a sequence in place (modifies the original sequence).

```
    numbers = [1, 2, 3, 4, 5]
    random.shuffle(numbers)
    print(numbers)  # Output: e.g., [3, 1, 5, 2, 4]
```

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

5. `random.sample(seq, k=len(seq))` : Returns a shuffled version of the sequence without modifying the original.

```
    shuffled = random.sample(numbers, k=len(numbers))
    print(shuffled)  # Output: e.g., [4, 2, 5, 1, 3]
```

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

Seeding Random Numbers

The `random.seed()` function initializes the random number generator with a specific seed value. This ensures that the sequence of random numbers is reproducible.

Example:

```
random.seed(42)  # Set the seed
print(random.random())  # Output: 0.6394267984578837
print(random.random())  # Output: 0.025010755222666936

random.seed(42)  # Reset the seed
print(random.random())  # Output: 0.6394267984578837 (same as before)
```

```
0.6394267984578837
0.025010755222666936
0.6394267984578837
```

Example Program

```
# Working with Random Numbers

import random
```

```python
# Generating random numbers
print("Random float between 0.0 and 1.0:", random.random())
print("Random float between 1.5 and 4.5:", random.uniform(1.5, 4.5))
print("Random integer between 1 and 10:", random.randint(1, 10))
print("Random integer from range 0 to 100 (step 5):", random.randrange(0,
100, 5))

# Selecting random elements
fruits = ["apple", "banana", "cherry"]
print("Random choice from fruits:", random.choice(fruits))
print("Random choices (with replacement):", random.choices(fruits, k=2))
print("Random sample (without replacement):", random.sample(fruits, 2))

# Shuffling sequences
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print("Shuffled numbers:", numbers)

# Seeding random numbers
random.seed(42)
print("Random number with seed 42:", random.random())
random.seed(42)
print("Random number with seed 42 (again):", random.random())
```

```
Random float between 0.0 and 1.0: 0.025010755222666936
Random float between 1.5 and 4.5: 2.3250879551073576
Random integer between 1 and 10: 4
Random integer from range 0 to 100 (step 5): 20
Random choice from fruits: cherry
Random choices (with replacement): ['apple', 'cherry']
Random sample (without replacement): ['cherry', 'apple']
Shuffled numbers: [2, 3, 1, 4, 5]
Random number with seed 42: 0.6394267984578837
Random number with seed 42 (again): 0.6394267984578837
```

---

14. Decorators

Decorators are a powerful and flexible feature in Python that allow you to modify or extend the behavior of functions or methods without changing their actual code. They are often used for logging, access control, memoization, and more.

What Are Decorators?

A decorator is a function that takes another function as input, adds some functionality to it, and returns a new function. Decorators are applied using the `@` symbol.

Example:

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

How Decorators Work

1. The decorator function (`my_decorator`) takes a function (`func`) as an argument.
2. Inside the decorator, a new function (`wrapper`) is defined that adds some behavior before and/or after calling the original function.
3. The decorator returns the `wrapper` function.
4. When the decorated function (`say_hello`) is called, the `wrapper` function is executed instead.

Decorators with Arguments

If the decorated function takes arguments, the `wrapper` function must accept those arguments and pass them to the original function.

Example:

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

```
Something is happening before the function is called.
Hello, Alice!
Something is happening after the function is called.
```

Chaining Decorators

You can apply multiple decorators to a single function. The decorators are applied from bottom to top.

Example:

```python
def decorator1(func):
    def wrapper():
        print("Decorator 1")
        func()
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2")
        func()
    return wrapper

@decorator1
@decorator2
def say_hello():
    print("Hello!")

say_hello()
```

```
Decorator 1
Decorator 2
Hello!
```

Decorators with Arguments

You can create decorators that accept arguments by adding an extra layer of nesting.

Example:

```python
def repeat(num_times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

```
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

Built-in Decorators

Python provides some built-in decorators, such as:

1. `@staticmethod`: Defines a static method that does not depend on the instance or class.
2. `@classmethod`: Defines a class method that takes the class as its first argument.
3. `@property`: Defines a method as a property, allowing it to be accessed like an attribute.

Example:

```python
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method.")

    @classmethod
    def class_method(cls):
        print(f"This is a class method of {cls.__name__}.")

    @property
    def my_property(self):
        return "This is a property."

# Usage
MyClass.static_method()  # Output: This is a static method.
MyClass.class_method()   # Output: This is a class method of MyClass.

obj = MyClass()
print(obj.my_property)   # Output: This is a property.
```

```
This is a static method.
This is a class method of MyClass.
This is a property.
```

Example Program

```
# Working with Decorators

# Basic decorator
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()

# Decorator with arguments
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def greet(name):
    print(f"Hello, {name}!")
```

```python
greet("Alice")

# Chaining decorators
def decorator1(func):
    def wrapper():
        print("Decorator 1")
        func()
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2")
        func()
    return wrapper

@decorator1
@decorator2
def say_hello():
    print("Hello!")

say_hello()

# Decorator with arguments
def repeat(num_times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")

# Built-in decorators
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method.")

    @classmethod
    def class_method(cls):
        print(f"This is a class method of {cls.__name__}.")

    @property
    def my_property(self):
        return "This is a property."

# Usage
MyClass.static_method()  # Output: This is a static method.
MyClass.class_method()   # Output: This is a class method of MyClass.

obj = MyClass()
print(obj.my_property)   # Output: This is a property.
```

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
Something is happening before the function is called.
Hello, Alice!
Something is happening after the function is called.
Decorator 1
Decorator 2
Hello!
Hello, Alice!
Hello, Alice!
Hello, Alice!
This is a static method.
This is a class method of MyClass.
This is a property.
```

---

15. Generators

Generators are a special type of iterator in Python that allow you to iterate over a sequence of values without storing the entire sequence in memory. They are defined using functions and the `yield` keyword. Generators are particularly useful for working with large datasets or infinite sequences.

What Are Generators?

A generator is a function that returns an iterator. Instead of using `return` to produce a value, a generator uses `yield`. When a generator function is called, it returns a generator object that can be iterated over.

Example:

```
def simple_generator():
    yield 1
    yield 2
```

```python
    yield 3

# Create a generator object
gen = simple_generator()

# Iterate over the generator
for value in gen:
    print(value)
```

```
1
2
3
```

How Generators Work

1. When a generator function is called, it returns a generator object but does not start execution.
2. The generator function runs until it encounters a `yield` statement, which produces a value and pauses the function.
3. The function resumes execution from where it left off when the next value is requested.

Advantages of Generators

- **Memory Efficiency**: Generators produce values on-the-fly, so they don't store the entire sequence in memory.
- **Lazy Evaluation**: Values are computed only when needed, making generators ideal for large or infinite sequences.

Creating Generators

4. **Using `yield`**:
   Define a generator function using the `yield` keyword.

   Example:

```python
  def count_up_to(n):
      count = 1
      while count <= n:
          yield count
          count += 1
```

```python
    # Create a generator object
    gen = count_up_to(5)

    # Iterate over the generator
    for value in gen:
        print(value)
```

```
1
2
3
4
5
```

5. **Generator Expressions**:
   Similar to list comprehensions, but use parentheses `()` instead of square brackets `[]`.

   Example:

```python
    gen = (x ** 2 for x in range(5))

    # Iterate over the generator
    for value in gen:
        print(value)
```

```
0
1
4
9
16
```

Infinite Generators

Generators can be used to create infinite sequences because they produce values on-the-fly.

Example:

```python
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

```python
# Create a generator object
gen = infinite_sequence()

# Print the first 5 values
for _ in range(5):
    print(next(gen))
```

```
0
1
2
3
4
```

Sending Values to Generators

You can send values to a generator using the `send()` method. This allows two-way communication between the generator and the caller.

Example:

```python
def generator_with_send():
    value = yield
    yield f"Received: {value}"

# Create a generator object
gen = generator_with_send()

# Start the generator
next(gen)

# Send a value to the generator
result = gen.send("Hello")
print(result)
```

```
Received: Hello
```

Example Program

```python
# Working with Generators

# Simple generator
```

```python
def simple_generator():
    yield 1
    yield 2
    yield 3

gen = simple_generator()
print("Simple Generator:")
for value in gen:
    print(value)

# Generator with yield
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

gen = count_up_to(5)
print("\nCount Up To 5:")
for value in gen:
    print(value)

# Generator expression
gen = (x ** 2 for x in range(5))
print("\nGenerator Expression:")
for value in gen:
    print(value)

# Infinite generator
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1

gen = infinite_sequence()
print("\nInfinite Generator (First 5 Values):")
for _ in range(5):
    print(next(gen))

# Sending values to a generator
def generator_with_send():
    value = yield
    yield f"Received: {value}"

gen = generator_with_send()
next(gen)
result = gen.send("Hello")
```

```
print("\nGenerator with Send:")
print(result)
```

```
Simple Generator:
1
2
3

Count Up To 5:
1
2
3
4
5

Generator Expression:
0
1
4
9
16

Infinite Generator (First 5 Values):
0
1
2
3
4

Generator with Send:
Received: Hello
```

---

16. Threading vs Multiprocessing

In Python, **threading** and **multiprocessing** are two approaches to achieve concurrency and parallelism. They allow you to run multiple tasks simultaneously, but they work differently and are suited for different types of problems.

What is Concurrency?

Concurrency is the ability of a program to manage multiple tasks at the same time. It doesn't necessarily mean that tasks are executed simultaneously; instead, the program switches between tasks to make progress on all of them.

What is Parallelism?

Parallelism is the ability of a program to execute multiple tasks simultaneously, typically by leveraging multiple CPU cores.

Threading

- **Threads** are lightweight processes that share the same memory space.
- Threading is suitable for **I/O-bound tasks** (e.g., reading/writing files, network requests) where the program spends time waiting for external resources.
- Python’s Global Interpreter Lock (GIL) prevents multiple threads from executing Python bytecode simultaneously, which can limit the performance of CPU-bound tasks.

Example:

```python
import threading
import time

def print_numbers():
    for i in range(5):
        print(f"Thread 1: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"Thread 2: {letter}")
        time.sleep(1)

# Create threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start threads
thread1.start()
thread2.start()

# Wait for threads to finish
thread1.join()
thread2.join()

print("Done!")
```

```
Thread 1: 0
Thread 2: A
Thread 1: 1
Thread 2: B
Thread 1: 2
Thread 2: C
Thread 1: 3
Thread 2: D
Thread 1: 4
Thread 2: E
Done!
```

Multiprocessing

- **Processes** are independent instances of a program that run in separate memory spaces.
- Multiprocessing is suitable for **CPU-bound tasks** (e.g., mathematical computations) where the program benefits from using multiple CPU cores.
- Each process has its own Python interpreter and memory space, so the GIL is not a limitation.

Example:

```python
import multiprocessing
import time

def print_numbers():
    for i in range(5):
        print(f"Process 1: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"Process 2: {letter}")
        time.sleep(1)

# Create processes
process1 = multiprocessing.Process(target=print_numbers)
process2 = multiprocessing.Process(target=print_letters)

# Start processes
process1.start()
process2.start()

# Wait for processes to finish
```

```
process1.join()
process2.join()

print("Done!")
```

```
Process 1: 0
Process 2: A
Process 1: 1
Process 2: B
Process 1: 2
Process 2: C
Process 1: 3
Process 2: D
Process 1: 4Process 2: E

Done!
```

Key Differences Between Threading and Multiprocessing

Feature	Threading	Multiprocessing
Memory	Threads share the same memory space.	Processes have separate memory spaces.
GIL	Affected by the GIL (limits CPU-bound tasks).	Not affected by the GIL.
Use Case	Best for I/O-bound tasks.	Best for CPU-bound tasks.
Overhead	Low overhead.	Higher overhead due to separate memory spaces.
Scalability	Limited by the GIL.	Scales well with multiple CPU cores.

When to Use Threading vs Multiprocessing

- **Use Threading**:
  - For I/O-bound tasks (e.g., file I/O, network requests).
  - When tasks involve waiting for external resources.
  - When you need to share data between tasks (since threads share memory).
- **Use Multiprocessing**:
  - For CPU-bound tasks (e.g., mathematical computations).
  - When you need to leverage multiple CPU cores.

- When tasks are independent and don't need to share data.

Example Program

```python
# Working with Threading and Multiprocessing

import threading
import multiprocessing
import time

# Threading example
def print_numbers():
    for i in range(5):
        print(f"Thread 1: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"Thread 2: {letter}")
        time.sleep(1)

print("Threading Example:")
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
print("Threading Done!\n")

# Multiprocessing example
def print_numbers():
    for i in range(5):
        print(f"Process 1: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"Process 2: {letter}")
        time.sleep(1)

print("Multiprocessing Example:")
process1 = multiprocessing.Process(target=print_numbers)
process2 = multiprocessing.Process(target=print_letters)
```

```
process1.start()
process2.start()

process1.join()
process2.join()
print("Multiprocessing Done!")
```

```
Threading Example:
Thread 1: 0
Thread 2: A
Thread 1: 1
Thread 2: B
Thread 1: 2
Thread 2: C
Thread 1: 3
Thread 2: D
Thread 1: 4
Thread 2: E
Threading Done!

Multiprocessing Example:
Process 1: 0
Process 2: A
Process 1: 1
Process 2: B
Process 1: 2
Process 2: C
Process 1: 3Process 2: D

Process 1: 4
Process 2: E
Multiprocessing Done!
```

---

17. Multithreading

Multithreading is a technique that allows a program to run multiple threads concurrently. Threads are lightweight processes that share the same memory space, making them ideal for **I/O-bound tasks** (e.g., file I/O, network requests) where the program spends time waiting for external resources.

What is a Thread?

A **thread** is the smallest unit of execution within a process. Multiple threads can exist within the same process and share resources such as memory and file handles.

Advantages of Multithreading

- **Concurrency**: Allows multiple tasks to run concurrently, improving responsiveness.
- **Resource Sharing**: Threads share the same memory space, making it easier to share data between tasks.
- **Efficiency**: Threads are lightweight compared to processes, so creating and switching between threads is faster.

Limitations of Multithreading in Python

- **Global Interpreter Lock (GIL)**: Python's GIL prevents multiple threads from executing Python bytecode simultaneously, which can limit the performance of **CPU-bound tasks**.
- **Thread Safety**: Shared data between threads can lead to race conditions if not properly synchronized.

Creating Threads

Python provides the `threading` module to work with threads. You can create a thread by subclassing `threading.Thread` or by passing a target function to the `threading.Thread` constructor.

Example:

```
import threading
import time

def print_numbers():
    for i in range(5):
        print(f"Thread 1: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"Thread 2: {letter}")
        time.sleep(1)

# Create threads
thread1 = threading.Thread(target=print_numbers)
```

```
thread2 = threading.Thread(target=print_letters)

# Start threads
thread1.start()
thread2.start()

# Wait for threads to finish
thread1.join()
thread2.join()

print("Done!")
```

```
Thread 1: 0
Thread 2: A
Thread 1: 1
Thread 2: B
Thread 1: 2
Thread 2: C
Thread 1: 3
Thread 2: D
Thread 1: 4
Thread 2: E
Done!
```

Thread Synchronization

When multiple threads access shared resources, you need to synchronize their access to avoid race conditions. Python provides several synchronization primitives, such as **locks**, **semaphores**, and **conditions**.

Example: Using a Lock

```
import threading

# Shared resource
counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

# Create threads
```

```
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

# Start threads
thread1.start()
thread2.start()

# Wait for threads to finish
thread1.join()
thread2.join()

print("Counter:", counter)  # Output: Counter: 200000
```

```
Counter: 200000
```

Daemon Threads

A **daemon thread** is a thread that runs in the background and does not prevent the program from exiting. When the main program exits, all daemon threads are automatically terminated.

Example:

```
import threading
import time

def daemon_task():
    while True:
        print("Daemon thread is running...")
        time.sleep(1)

# Create a daemon thread
daemon_thread = threading.Thread(target=daemon_task, daemon=True)

# Start the daemon thread
daemon_thread.start()

# Main program
print("Main program is running...")
time.sleep(3)
print("Main program is done.")
```

```
Daemon thread is running...
Main program is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Main program is done.
```

Thread Pools

A **thread pool** is a collection of pre-initialized threads that are ready to perform tasks. Python's `concurrent.futures` module provides a `ThreadPoolExecutor` for managing thread pools.

Example:

```python
from concurrent.futures import ThreadPoolExecutor
import time

def task(name):
    print(f"Task {name} started")
    time.sleep(2)
    print(f"Task {name} finished")

# Create a thread pool with 3 threads
with ThreadPoolExecutor(max_workers=3) as executor:
    # Submit tasks to the thread pool
    futures = [executor.submit(task, i) for i in range(5)]

print("All tasks completed.")
```

```
Task 0 started
Task 1 started
Task 2 started
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Task 0 finished
Task 3 started
Task 1 finished
Task 4 started
Task 2 finished
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Task 3 finished
Task 4 finished
All tasks completed.
```

Example Program

```python
# Working with Multithreading

import threading
import time
from concurrent.futures import ThreadPoolExecutor

# Basic threading example
def print_numbers():
    for i in range(5):
        print(f"Thread 1: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"Thread 2: {letter}")
        time.sleep(1)

print("Basic Threading Example:")
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

```python
print("Basic Threading Done!\n")

# Thread synchronization example
counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

print("Thread Synchronization Example:")
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
print("Counter:", counter)
print("Thread Synchronization Done!\n")

# Daemon thread example
def daemon_task():
    while True:
        print("Daemon thread is running...")
        time.sleep(1)

print("Daemon Thread Example:")
daemon_thread = threading.Thread(target=daemon_task, daemon=True)
daemon_thread.start()

print("Main program is running...")
time.sleep(3)
print("Main program is done.\n")

# Thread pool example
def task(name):
    print(f"Task {name} started")
    time.sleep(2)
    print(f"Task {name} finished")

print("Thread Pool Example:")
with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(task, i) for i in range(5)]

print("All tasks completed.")
```

```
Daemon thread is running...
Basic Threading Example:
Thread 1: 0
Thread 2: A
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Thread 1: 1
Thread 2: B
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Thread 1: 2
Thread 2: C
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Thread 1: 3
Thread 2: D
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Thread 1: 4
Thread 2: E
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Basic Threading Done!

Thread Synchronization Example:
Daemon thread is running...
Counter: 200000
Thread Synchronization Done!

Daemon Thread Example:
Daemon thread is running...
Main program is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Main program is done.

Thread Pool Example:
Task 0 started
Daemon thread is running...
Task 1 started
Task 2 started
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Task 0 finished
Task 3 started
Task 1 finished
Task 4 started
Daemon thread is running...
Task 2 finished
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Task 3 finished
Task 4 finished
Daemon thread is running...
All tasks completed.
```

---

18. Multiprocessing in Python

Multiprocessing is a Python module that allows you to create processes that can run concurrently, taking advantage of multiple CPU cores. This is particularly useful for CPU-bound tasks (tasks that require heavy computation) because it enables true parallel execution, unlike threading, which is limited by Python's Global Interpreter Lock (GIL).

Key Concepts in Multiprocessing

1. **Process**: A process is an instance of a program that runs independently. Each process has its own memory space, which means it doesn’t share data with other processes by default.
2. **Parallelism**: Multiprocessing enables parallelism, where multiple tasks are executed simultaneously on different CPU cores.
3. **Inter-Process Communication (IPC)**: Processes can communicate with each other using mechanisms like `Queue`, `Pipe`, or shared memory.
4. **GIL (Global Interpreter Lock)**: The GIL prevents multiple threads from executing Python bytecode simultaneously in a single process. Multiprocessing avoids this limitation by using separate processes.

Basic Usage of Multiprocessing

To use the `multiprocessing` module, you typically follow these steps:

1. Import the `multiprocessing` module.
2. Define a function that will run in a separate process.
3. Create a `Process` object and specify the target function.
4. Start the process using the `start()` method.
5. Optionally, wait for the process to finish using the `join()` method.

Here’s an example:

```python
import multiprocessing
import time

def worker_function(name):
    print(f"Process {name} started")
    time.sleep(2)  # Simulate some work
    print(f"Process {name} finished")

if __name__ == "__main__":
    # Create two processes
    process1 = multiprocessing.Process(target=worker_function, args=
("Process 1",))
    process2 = multiprocessing.Process(target=worker_function, args=
("Process 2",))

    # Start the processes
    process1.start()
    process2.start()

    # Wait for the processes to finish
    process1.join()
    process2.join()

    print("All processes finished")
```

```
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Process Process 1 started
Process Process 2 started
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Process Process 1 finished
Process Process 2 finished
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
All processes finished
```

Notice

there is " Daemon thread is running... " in the out put.

Why This Happens in Jupyter Notebook

Jupyter Notebooks run on an IPython kernel, which has its own event loop and threading model. When you use the `multiprocessing` module, it can sometimes interfere with the kernel's behavior, leading to unexpected output or behavior, such as the repeated "Daemon thread is running..." messages.This could solved by restarting the kernel in Jupyter Notebook.

Key Components of Multiprocessing

6. `Process` **Class**:
    - Used to create and manage processes.
    - Key methods:

    - `start()` : Starts the process.
    - `join()` : Waits for the process to complete.
    - `is_alive()` : Checks if the process is still running.
7. `Queue` :
    - A thread-safe way to share data between processes.
    - Example:

```
    import multiprocessing

    def worker(q):
        q.put("Hello from the worker process!")

    if __name__ == "__main__":
        q = multiprocessing.Queue()
        p = multiprocessing.Process(target=worker, args=(q,))
        p.start()
        print(q.get())  # Output: Hello from the worker process!
        p.join()
```

```
Hello from the worker process!
```

8. `Pool` :
    - A pool of worker processes for parallel execution of a function across multiple inputs.
    - Example:

```
    import multiprocessing

    def square(x):
        return x * x

    if __name__ == "__main__":
        with multiprocessing.Pool(processes=4) as pool:
            results = pool.map(square, range(10))
        print(results)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

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

9. `Pipe` :
    - A two-way communication channel between processes.
    - Example:

```
    import multiprocessing
```

```python
    def worker(conn):
        conn.send("Message from worker")
        conn.close()

    if __name__ == "__main__":
        parent_conn, child_conn = multiprocessing.Pipe()
        p = multiprocessing.Process(target=worker, args=(child_conn,))
        p.start()
        print(parent_conn.recv())  # Output: Message from worker
        p.join()
```

```
Daemon thread is running...
Message from worker
Daemon thread is running...
```

10. **Shared Memory**:

- Allows processes to share data using `Value` and `Array`.
- Example:

```python
    import multiprocessing

    def worker(val):
        val.value += 1

    if __name__ == "__main__":
        shared_value = multiprocessing.Value("i", 0)  # 'i' for integer
        p = multiprocessing.Process(target=worker, args=(shared_value,))
        p.start()
        p.join()
        print(shared_value.value)  # Output: 1
```

```
1
```

Advantages of Multiprocessing

- True parallel execution for CPU-bound tasks.
- Avoids the GIL limitation.
- Each process has its own memory space, reducing the risk of data corruption.

Disadvantages of Multiprocessing

- Higher memory usage compared to threading.

- Inter-process communication can be complex.
- Slower to start compared to threads due to the overhead of creating new processes.

---

19. Function Arguments in Python

In Python, functions can accept arguments (also called parameters) to make them more flexible and reusable. Understanding how to work with function arguments is essential for writing clean and efficient code. Python supports several types of function arguments:

1. **Positional Arguments**
2. **Keyword Arguments**
3. **Default Arguments**
4. **Variable-Length Arguments (`*args` and** \`kwargs\`)**
5. **Keyword-Only Arguments**
6. **Positional-Only Arguments (Python 3.8+)**

Let's explore each of these in detail.

---

1. Positional Arguments

Positional arguments are the most common type of arguments. They are passed to a function in the order they are defined.

```
def greet(name, message):
    print(f"{message}, {name}!")

greet("Alice", "Hello")  # Output: Hello, Alice!
```

```
Hello, Alice!
```

- The order of arguments matters. `"Alice"` is assigned to `name`, and `"Hello"` is assigned to `message`.

---

2. Keyword Arguments

Keyword arguments are passed with a keyword (i.e., the parameter name) and can be in any order.

```
greet(message="Hi", name="Bob")  # Output: Hi, Bob!
```

```
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
Hi, Bob!
```

- Here, the order doesn't matter because the arguments are explicitly named.

---

3. Default Arguments

Default arguments allow you to define a default value for a parameter. If the caller doesn't provide a value, the default is used.

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("Alice")  # Output: Hello, Alice!
greet("Bob", "Hi")  # Output: Hi, Bob!
```

```
Daemon thread is running...
Daemon thread is running...
Hello, Alice!
Hi, Bob!
```

- `message` has a default value of `"Hello"`, so it's optional.

Note: Default arguments are evaluated only once when the function is defined, not each time the function is called. Be careful with mutable default arguments (e.g., lists or dictionaries).

---

4. Variable-Length Arguments

Python allows you to handle an arbitrary number of arguments using `*args` and `**kwargs`.

- `*args`: Used to pass a variable number of positional arguments. It collects them into a tuple.
- `kwargs`**: Used to pass a variable number of keyword arguments. It collects them into a dictionary.

Example with `*args`:

```
def add(*args):
    return sum(args)

print(add(1, 2, 3))  # Output: 6
print(add(4, 5, 6, 7))  # Output: 22
```

```
Daemon thread is running...
Daemon thread is running...
6
22
```

Example with `kwargs`:

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York
```

```
name: Alice
age: 30
city: New York
```

---

5. Keyword-Only Arguments

Keyword-only arguments are arguments that can only be passed using the keyword syntax. They are defined after a `*` in the function signature.

```
def greet(*, name, message):
    print(f"{message}, {name}!")

greet(name="Alice", message="Hi")  # Output: Hi, Alice!
# greet("Alice", "Hi")  # This would raise a TypeError
```

```
Hi, Alice!
```

- The `*` enforces that all arguments after it must be passed as keyword arguments.

6. Positional-Only Arguments (Python 3.8+)

Positional-only arguments are arguments that can only be passed by position. They are defined before a `/` in the function signature.

```python
def greet(name, /, message):
    print(f"{message}, {name}!")

greet("Alice", message="Hi")  # Output: Hi, Alice!
# greet(name="Alice", message="Hi")  # This would raise a TypeError
```

```
Hi, Alice!
```

- The `/` enforces that all arguments before it must be passed as positional arguments.

Combining All Types of Arguments

You can combine all these types of arguments in a single function. The order of parameters must follow this rule:

1. Positional-only arguments (before `/`).
2. Regular positional arguments.
3. `*args` (variable-length positional arguments).
4. Keyword-only arguments (after `*`).
5. `**kwargs` (variable-length keyword arguments).

Example:

```python
def example(a, b, /, c, d=4, *args, e, f=6, **kwargs):
    print(f"a: {a}, b: {b}, c: {c}, d: {d}, args: {args}, e: {e}, f: {f},
kwargs: {kwargs}")

example(1, 2, 3, e=5, extra="hello")
# Output:
# a: 1, b: 2, c: 3, d: 4, args: (), e: 5, f: 6, kwargs: {'extra': 'hello'}
```

```
a: 1, b: 2, c: 3, d: 4, args: (), e: 5, f: 6, kwargs: {'extra': 'hello'}
```

Best Practices for Function Arguments

- Use descriptive names for parameters to improve readability.
- Avoid mutable default arguments (e.g., `def func(arg=[])`) to prevent unexpected behavior.
- Use `*args` and `**kwargs` sparingly, as they can make the function signature less clear.
- Use keyword-only arguments to enforce clarity and prevent misuse.

---

20. The Asterisk (`*`) Operator in Python

The asterisk (`*`) operator is a versatile symbol in Python with multiple uses depending on the context. Here are the main ways it is used:

1. **Multiplication and Exponentiation**
2. **Unpacking Iterables**
3. **Extended Unpacking (Python 3+)**
4. **Variable-Length Arguments in Functions (`*args`)**
5. **Unpacking in Function Calls**
6. **Keyword Argument Unpacking (``)****
7. **Keyword-Only Arguments in Functions**

Let's explore each of these in detail.

---

1. Multiplication and Exponentiation

The `*` operator is used for **multiplication** and the `**` operator is used for **exponentiation**.

```python
# Multiplication
result = 5 * 3
print(result)  # Output: 15

# Exponentiation
result = 2 ** 3
print(result)  # Output: 8
```

```
15
8
```

---

2. Unpacking Iterables

The `*` operator can be used to **unpack iterables** (e.g., lists, tuples) into individual elements.

```
numbers = [1, 2, 3]
print(*numbers)  # Output: 1 2 3
```

```
1 2 3
```

This is equivalent to:

```
print(1, 2, 3)
```

```
1 2 3
```

---

3. Extended Unpacking (Python 3+)

Python 3 introduced **extended unpacking**, which allows you to unpack parts of an iterable.

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

```
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
1
[2, 3, 4]
5
```

- `*middle` captures all the elements between the first and last elements.

---

4. Variable-Length Arguments in Functions (`*args`)

The `*` operator is used in function definitions to accept a **variable number of positional arguments**. These arguments are collected into a tuple.

```
def sum_numbers(*args):
    return sum(args)
```

```
print(sum_numbers(1, 2, 3))  # Output: 6
print(sum_numbers(4, 5, 6, 7))  # Output: 22
```

```
6
22
```

- `*args` allows the function to accept any number of positional arguments.

---

5. Unpacking in Function Calls

The `*` operator can be used to **unpack an iterable into individual arguments** when calling a function.

```
def greet(name, message):
    print(f"{message}, {name}!")

data = ["Alice", "Hello"]
greet(*data)  # Output: Hello, Alice!
```

```
Hello, Alice!
```

- `*data` unpacks the list into two arguments: `name="Alice"` and `message="Hello"`.

---

6. Keyword Argument Unpacking (``)**

The `**` operator is used to **unpack a dictionary into keyword arguments**.

```
def greet(name, message):
    print(f"{message}, {name}!")

data = {"name": "Bob", "message": "Hi"}
greet(**data)  # Output: Hi, Bob!
```

```
Hi, Bob!
```

- `**data` unpacks the dictionary into keyword arguments: `name="Bob"` and `message="Hi"`.

---

7. Keyword-Only Arguments in Functions

The `*` operator can be used in function definitions to enforce **keyword-only arguments**. Arguments after `*` must be passed as keyword arguments.

```
def greet(*, name, message):
    print(f"{message}, {name}!")

greet(name="Alice", message="Hi")  # Output: Hi, Alice!
# greet("Alice", "Hi")  # This would raise a TypeError
```

```
Hi, Alice!
```

- The `*` ensures that `name` and `message` must be passed as keyword arguments.

Combining `*` and `` in Function Calls**

You can combine `*` and `**` to unpack both positional and keyword arguments.

```
def func(a, b, c):
    print(f"a: {a}, b: {b}, c: {c}")

args = [1, 2]
kwargs = {"c": 3}
func(*args, **kwargs)  # Output: a: 1, b: 2, c: 3
```

```
a: 1, b: 2, c: 3
```

Summary of Uses

Use Case	Example
Multiplication	`5 * 3` → `15`
Exponentiation	`2 ** 3` → `8`
Unpacking iterables	`print(*[1, 2, 3])` → `1 2 3`
Extended unpacking	`first, *middle, last = [1, 2, 3, 4, 5]`
Variable-length arguments (`*args`)	`def func(*args): ...`
Unpacking in function calls	`func(*[1, 2, 3])`

Use Case	Example
Keyword argument unpacking (`**`)	`func(**{"a": 1, "b": 2})`
Keyword-only arguments	`def func(*, a, b): ...`

---

21. Shallow vs Deep Copying in Python

In Python, copying objects is a common operation, but it’s important to understand the difference between **shallow copying** and **deep copying**. The behavior of these operations depends on whether the object contains mutable or immutable elements.

Key Concepts

1. **Mutable vs Immutable Objects**:
   - **Mutable objects**: Objects whose state can be changed after creation (e.g., lists, dictionaries, sets).
   - **Immutable objects**: Objects whose state cannot be changed after creation (e.g., integers, strings, tuples).
2. **Assignment**:
   - When you assign an object to a new variable, both variables reference the **same object** in memory.
   - Changes to the object through one variable will affect the other.
3. **Shallow Copy**:
   - Creates a new object but inserts **references** to the original nested objects.
   - Changes to mutable nested objects will affect both the original and the copy.
4. **Deep Copy**:
   - Creates a new object and recursively copies all nested objects.
   - Changes to mutable nested objects will **not** affect the original or the copy.

Shallow Copy

A shallow copy creates a new object but does not recursively copy nested objects. Instead, it inserts references to the original nested objects.

How to Create a Shallow Copy:

- Use the `copy()` method (for lists, dictionaries, etc.).
- Use the `copy.copy()` function from the `copy` module.

Example:

```
import copy

original = [[1, 2, 3], [4, 5, 6]]
shallow_copy = copy.copy(original)

# Modify the nested list in the shallow copy
shallow_copy[0][0] = 99

print(original)     # Output: [[99, 2, 3], [4, 5, 6]]
print(shallow_copy) # Output: [[99, 2, 3], [4, 5, 6]]
```

```
[[99, 2, 3], [4, 5, 6]]
[[99, 2, 3], [4, 5, 6]]
```

- Notice that modifying the nested list in the shallow copy also affects the original.

Deep Copy

A deep copy creates a new object and recursively copies all nested objects, ensuring that no references to the original nested objects are retained.

How to Create a Deep Copy:

- Use the `copy.deepcopy()` function from the `copy` module.

Example:

```
import copy

original = [[1, 2, 3], [4, 5, 6]]
deep_copy = copy.deepcopy(original)

# Modify the nested list in the deep copy
deep_copy[0][0] = 99

print(original)  # Output: [[1, 2, 3], [4, 5, 6]]
print(deep_copy) # Output: [[99, 2, 3], [4, 5, 6]]
```

```
[[1, 2, 3], [4, 5, 6]]
[[99, 2, 3], [4, 5, 6]]
```

- Notice that modifying the nested list in the deep copy does **not** affect the original.

When to Use Shallow Copy vs Deep Copy

Use Case	Shallow Copy	Deep Copy
Object contains only immutable elements	Use shallow copy (no difference in behavior).	Use deep copy (no difference in behavior).
Object contains mutable nested objects	Use shallow copy if you want changes to nested objects to affect the original.	Use deep copy if you want changes to nested objects to **not** affect the original.
Performance	Faster (less memory and computation).	Slower (more memory and computation due to recursive copying).

Practical Examples

Example 1: Shallow Copy with a List of Lists

```
import copy

original = [[1, 2], [3, 4]]
shallow_copy = copy.copy(original)

shallow_copy[0][0] = 99
print(original)     # Output: [[99, 2], [3, 4]]
print(shallow_copy) # Output: [[99, 2], [3, 4]]
```

```
[[99, 2], [3, 4]]
[[99, 2], [3, 4]]
```

Example 2: Deep Copy with a List of Lists

```
import copy

original = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(original)

deep_copy[0][0] = 99
print(original)  # Output: [[1, 2], [3, 4]]
print(deep_copy) # Output: [[99, 2], [3, 4]]
```

```
[[1, 2], [3, 4]]
[[99, 2], [3, 4]]
```

Example 3: Shallow Copy with a Dictionary

```python
import copy

original = {"a": [1, 2], "b": [3, 4]}
shallow_copy = copy.copy(original)

shallow_copy["a"][0] = 99
print(original)     # Output: {'a': [99, 2], 'b': [3, 4]}
print(shallow_copy) # Output: {'a': [99, 2], 'b': [3, 4]}
```

```
{'a': [99, 2], 'b': [3, 4]}
{'a': [99, 2], 'b': [3, 4]}
```

Example 4: Deep Copy with a Dictionary

```python
import copy

original = {"a": [1, 2], "b": [3, 4]}
deep_copy = copy.deepcopy(original)

deep_copy["a"][0] = 99
print(original)  # Output: {'a': [1, 2], 'b': [3, 4]}
print(deep_copy) # Output: {'a': [99, 2], 'b': [3, 4]}
```

```
{'a': [1, 2], 'b': [3, 4]}
{'a': [99, 2], 'b': [3, 4]}
```

Summary

Aspect	Shallow Copy	Deep Copy
Copies nested objects?	No (references are shared).	Yes (recursively copies nested objects).
Performance	Faster.	Slower.
Use Case	When nested objects are immutable or shared references are acceptable.	When nested objects are mutable and independent copies are needed.

22. Context Managers in Python

Context managers are a way to manage resources (e.g., files, database connections, locks) in Python. They ensure that resources are properly acquired and released, even if an exception occurs. The most common use of context managers is with the `with` statement.

Key Concepts

1. **Resource Management**:
    - Resources like files, network connections, or locks need to be properly opened/acquired and closed/released.
    - Failing to release resources can lead to leaks, which can cause performance issues or crashes.
2. **Context Manager Protocol**:
    - A context manager is an object that implements the `__enter__` and `__exit__` methods.
    - The `__enter__` method is called when entering the `with` block.
    - The `__exit__` method is called when exiting the `with` block, even if an exception occurs.
3. **The `with` Statement**:
    - The `with` statement simplifies resource management by automatically calling the `__enter__` and `__exit__` methods.

Using Context Managers

The most common example of a context manager is working with files. Instead of manually opening and closing a file, you can use the `with` statement to ensure the file is properly closed.

Example: File Handling with `with`

```python
# Without context manager
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()  # Must remember to close the file

# With context manager
with open("example.txt", "w") as file:
    file.write("Hello, World!")
# File is automatically closed when the block is exited
```

- The `with` statement ensures that the file is closed, even if an exception occurs within the block.

Creating Custom Context Managers

You can create your own context managers by defining a class with `__enter__` and `__exit__` methods.

Example: Custom Context Manager

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self  # Optional: Return an object to use in the `with` block

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
        if exc_type is not None:
            print(f"An exception occurred: {exc_value}")
        # Return True to suppress the exception, False to propagate it
        return False

# Using the custom context manager
with MyContextManager() as cm:
    print("Inside the context")
    # raise ValueError("Something went wrong")  # Uncomment to test
exception handling
```

```
Entering the context
Inside the context
Exiting the context
```

- If an exception occurs, the `__exit__` method is still called, and you can handle the exception within it.

Using `contextlib` for Simpler Context Managers

The `contextlib` module provides utilities for creating context managers without defining a class. The most common utility is `contextlib.contextmanager`, which allows you to create a context manager using a generator function.

Example: Context Manager with `contextlib`

```
from contextlib import contextmanager

@contextmanager
def my_context_manager():
    print("Entering the context")
    try:
        yield  # The block inside the `with` statement runs here
    except Exception as e:
        print(f"An exception occurred: {e}")
    finally:
        print("Exiting the context")

# Using the context manager
with my_context_manager():
    print("Inside the context")
    # raise ValueError("Something went wrong")  # Uncomment to test
exception handling
```

```
Entering the context
Inside the context
Exiting the context
```

Common Use Cases for Context Managers

4. **File Handling**:
    - Automatically close files after reading or writing.

```
with open("example.txt", "r") as file:
    content = file.read()
```

5. **Database Connections**:
    - Automatically close database connections.

```
with db_connection() as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM table")
```

6. **Locks in Multithreading**:
    - Automatically release locks.

```
with threading.Lock():
    # Critical section
    pass
```

7. **Temporary Changes**:
    - Temporarily change the state (e.g., redirecting `stdout`).

```
from contextlib import redirect_stdout
import io

f = io.StringIO()
with redirect_stdout(f):
    print("This goes to the buffer")
print(f.getvalue())  # Output: This goes to the buffer
```

```
This goes to the buffer
```

Advantages of Context Managers

- **Resource Safety**: Ensures resources are properly released, even if an exception occurs.
- **Readability**: Makes code cleaner and easier to understand.
- **Reusability**: Context managers can be reused across different parts of the code.

Summary

Aspect	Details
Purpose	Manage resources (e.g., files, locks) safely and efficiently.
Syntax	`with context_manager as variable: ...`
Built-in Context Managers	`open()`, `threading.Lock()`, `contextlib.redirect_stdout()`, etc.
Custom Context Managers	Implement `__enter__` and `__exit__` methods or use `contextlib.contextmanager`.
Exception Handling	The `__exit__` method can handle exceptions raised in the `with` block.