Zero to Hero

LEARN

PYTHON

Easy-to-follow

Hands-on learning

HarryCodeCraft

Python For Beginner

- **What is Python?**
  Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used in web development, data analysis, machine learning, automation, and more.
- **Why Learn Python?**
  - Beginner-friendly syntax
  - Extensive libraries and frameworks
  - Strong community support
  - Versatile for various applications

Setting Up Python on Linux and Windows: Step-by-Step Guide

1. Install Python

On Windows

- **Download Python**
  - Visit python.org and download the Windows installer.
  - Choose the appropriate version (32-bit or 64-bit) based on your system.
- **Install Python**
  - Run the installer.
  - Check the box **"Add Python to PATH"** to make Python accessible from the Command Prompt.
  - Choose **Customize Installation** for optional features like pip, IDLE, and development tools.
  - Complete the installation process.
- **Verify Installation**

- Open Command Prompt.

Run:
python --version

pip --version

On Linux

Update System Packages
sudo apt update && sudo apt upgrade -y # For Debian/Ubuntu

- **Install Python**

For Debian/Ubuntu:
sudo apt install python3 python3-pip -y

For Red Hat/CentOS:
sudo yum install python3 python3-pip -y

Verify Installation
python3 --version

pip3 --version

---

2. Choose an Editor

Recommended Editors for Both Linux and Windows

- **VS Code (Visual Studio Code)**
  - Download from code.visualstudio.com.

    - Install the **Python Extension** for debugging, syntax highlighting, and more.

Command to install on Linux (Debian/Ubuntu):
sudo apt install code

- **PyCharm**
    - Download from jetbrains.com/pycharm.
    - Offers a free Community Edition.

- **Jupyter Notebook**

Install via pip:
pip install notebook

Launch:
jupyter notebook

---

3. Verify Python and Pip Installation

On Windows

Open Command Prompt and run:
python --version

pip --version

On Linux

Open a terminal and run:
python3 --version

pip3 --version

---

4. Set Up Virtual Environments (Optional but Recommended)

On Windows

Create a virtual environment:
python -m venv myenv

Activate the environment:
myenv\Scripts\activate

Deactivate with:
deactivate

On Linux

Create a virtual environment:
python3 -m venv myenv

Activate the environment:
source myenv/bin/activate

Deactivate with:
deactivate

---

5. Install Essential Libraries

Use pip to install Python libraries.

Example Commands

Install libraries:
pip install numpy pandas matplotlib

Upgrade pip:
python -m pip install --upgrade pip # Windows

python3 -m pip install --upgrade pip # Linux

---

6. Additional Tips

- **Linux Users**
  - Use a package manager like apt or yum to install Python dependencies.

Install build tools if needed:
sudo apt install build-essential -y

- **Windows Users**
  - Use PowerShell or Command Prompt for Python commands.
  - Use Windows Subsystem for Linux (WSL) for a Linux-like development environment.

Script Mode: Save a file as script.py and run it with:

python script.py

Basic Syntax

Hello World

print("Hello, World!")

Python Comments

Single-line Comments
Comments in Python begin with a # symbol, and Python will ignore everything following the # on that line:

 # This is a comment

print("Hello, World!")

Inline Comments
Comments can also be placed at the end of a line, and Python will ignore the rest of the line:
print("Hello, World!") **# This is a comment**

Multiline Comments

Using Multiple # Symbols
Python does not have a specific syntax for multiline comments. However, you can use multiple # symbols, one per line:

```
# This is a comment

# written in

# more than just one line

print("Hello, World!")
```

Using Triple Quotes for Multiline Comments
Since Python ignores string literals that are not assigned to a variable, you can use triple quotes (""" or ''') to create multiline comments:

```
"""

This is a comment

written in

more than just one line

"""

    print("Hello, World!")
```

1. Operators

Arithmetic Operators

These operators perform mathematical operations like addition, subtraction, etc.

- + : Addition
- - : Subtraction
- * : Multiplication
- / : Division
- % : Modulus (remainder of division)
- // : Floor division (returns the integer part of the division)
- ** : Exponentiation (raising to a power)

Example:

```
a = 10

b = 5

print(a + b)  # Output: 15

print(a - b)  # Output: 5
```

Comparison Operators

These operators compare two values and return True or False.

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

Logical Operators
These operators are used to combine conditional statements.

- and : Returns True if both conditions are true
- or : Returns True if at least one condition is true
- not : Reverses the result (returns True if the condition is false)

---

2. Variables and Data Types:

In Python, variables are used to store data. You can think of a variable as a box where you store something (like a number or a name). Data types tell Python what kind of data you're storing.

- **Integer**: Whole numbers (e.g., 5)
- **Float**: Numbers with decimal points (e.g., 3.14)
- **String**: Text (e.g., "Alice")
- **Boolean**: True or False values (e.g., True)

```
x = 5         # Integer
y = 3.14      # Float
name = "Alice" # String
is_active = True # Boolean
```

---

3. Lists:

A list is like a collection of items. You can store multiple items in a list, and each item can be accessed by its position (index). Lists are very flexible and allow you to change, add, or remove items.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
fruits.append("orange")  # Adding an item
```

Output: ['apple', 'banana', 'cherry', 'orange']

Lists (Advanced Operations):

Lists are one of the most important data structures in Python. You can perform various operations on them.

- **Slicing**: Extract a portion of a list.

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

- **List Comprehension**: A compact way to create

lists. squares = [x**2 for x in range(5)]
print(squares) # Output: [0, 1, 4, 9, 16]

- **Sorting and Reversing**:

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

numbers.reverse()  # Reversing the list
print(numbers)  # Output: [9, 5, 2, 1]
```

---

4. Tuples:

A **tuple** is similar to a list but with one important difference: **it is immutable**. This means that once you create a tuple, you cannot change, add, or remove elements

from it. Tuples are useful when you want to store a collection of values that should not be modified, like the coordinates of a point or days of the week.

Creating a Tuple

Tuples are defined using parentheses (), and they can contain elements of different data types (e.g., integers, strings, booleans).

```
# Creating a tuple
my_tuple = (1, 2, 3, "apple", True)
print(my_tuple)  # Output: (1, 2, 3, 'apple', True)
```

Tuple Immutability

Since tuples are immutable, you cannot change their values after creation. Trying to do so will raise an error.

```
my_tuple[1] = 10  # ✗ This will raise an error
```

Tuple with One Element

To create a tuple with just one element, you need to add a trailing comma.

```
single_tuple = (5,)  # ✓ Correct
not_a_tuple = (5)    # ✗ This is just an integer
print(type(single_tuple)) # Output: <class 'tuple'>
print(type(not_a_tuple))  # Output: <class 'int'>
```

---

5. Dictionaries:

A dictionary is like a collection of key-value pairs. Each value is associated with a unique key. You can use the key to access the value.

```
person = {"name": "Alice", "age": 25}
print(person["name"]) # Output: Alice
```

---

6. Conditional Statements:

Conditional statements let you check if something is true or false and then take different actions based on that. It's like asking a question: "Is this true?" If yes, do one thing; if no, do something else.

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

---

7. Loops:

Loops allow you to repeat a block of code multiple times.

- **For Loop**: You use a for loop when you know how many times you want to repeat something.

```
for i in range(5):
    print(i)  # Output: 0, 1, 2, 3, 4
```

- **While Loop**: You use a while loop when you want to repeat something until a certain condition is met.

```
count = 0
while count < 5:
    print(count)
    count += 1  # Output: 0, 1, 2, 3, 4
```

---

8. Functions:

A function is a block of code that does something. You can define your own functions to organize your code and reuse it. Functions help make your code cleaner and more efficient.

```
def greet(name):
    return "Hello, " + name

print(greet("Alice"))  # Output: Hello, Alice
```

```
def square(x):

        return x * x

print(square(4)) # Output: 1
```

---

9. Classes and Objects:

In Python, you can create your own types using **classes**. A class is like a blueprint for creating objects. An **object** is an instance of a class. Classes allow you to group related data and functions together.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."

person1 = Person("Alice", 25)
print(person1.greet())  # Output: Hello, my name is Alice and I am 25 years old.
```

---

10. Importing Libraries:

Python comes with a lot of built-in libraries (also called **modules**) that help you do common tasks. You can import these libraries into your code to use their functionality.

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

---

11. Fibonacci Series in Python

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()
```

```
fib(1000)
```

Output:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

Fibonacci in Stock Market

```
def fib(n):
    a, b = 0, 1
    sequence = []
    while a < n:
        sequence.append(a)
        a, b = b, a + b
    return sequence

max_price = 1000
retracement_levels = fib(max_price)
print("Fibonacci retracement levels:", retracement_levels)
```

Output:

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]

When to Buy & Sell?

📌 **Buy:** When the price **retraces** to **61.8%, 50%, or 38.2%** and **bounces up**.
📌 **Sell:** When the price **rises** to **38.2%, 50%, or 61.8%** and **struggles to go higher**.
38.2%

- **50%**
- **61.8%**
- **78.6%**

These levels are calculated by dividing each number in the Fibonacci sequence by the number two places ahead of it. For example:

- **55 ÷ 144 ≈ 0.3819**, which is approximately **38.2%**.
- **89 ÷ 233 ≈ 0.3819**, which is also **38.2%**.

Example Trading Strategy

1️⃣ Stock peaks at **₹1000**
2️⃣ Drops to **₹618 (61.8%)** → **Buy**
3️⃣ Rises to **₹786 (78.6%)** → **Sell**
4️⃣ If it **breaks ₹1000**, wait for new **Fibonacci extension levels**.

---

12. String Manipulation:

Strings are used to handle text data. You can perform several operations on strings.

- **Concatenation**: Combine strings.

```
greeting = "Hello"
name = "Alice"
```

```
message = greeting + ", " + name
print(message)  # Output: Hello, Alice
```

- **String Methods**:

```
text = "hello world"
print(text.upper())  # Output: HELLO WORLD
print(text.replace("world", "Python"))  # Output: hello Python
```

- **String Formatting**: Insert variables into strings.

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

---

13. Exception Handling:

Exceptions are errors that occur during program execution. Python provides a way to handle these errors gracefully using try, except.

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

- **Else and Finally**: You can also add else and finally blocks.

```
try:
    x = 10 / 2
```

```
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful!")
finally:
    print("This will always run.")
```

---

14. Lambda Functions:

Lambda functions are small anonymous functions defined using the lambda keyword.

```
# Regular function
def add(x, y):
    return x + y

# Lambda function
add_lambda = lambda x, y: x + y
print(add_lambda(2, 3))  # Output: 5
```

Lambda functions are often used in places where you need a simple function for a short period, like in map(), filter(), or sorted().

---

15. Map, Filter, and Reduce:

- **Map**: Applies a function to all items in an input

```
list. numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

- **Filter**: Filters elements based on a condition.

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

- **Reduce** (from functools): Reduces a list to a single value by applying a function cumulatively.

```
from functools import reduce

numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

---

16. File Handling

Sometimes, you may need to read from or write to files. Python allows you to easily open, read, and write files.

- **Writing to a file**:

```
with open("example.txt", "w") as file:
    file.write("Hello, world!")
```

- **Reading from a file**:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # Output: Hello, world!
```

Working with Files (Advanced):

- **Reading Large Files**: Instead of reading the entire file into memory, you can read it line by line.

```
with open("large_file.txt", "r") as file:
    for line in file:
        print(line.strip())  # strip() removes leading/trailing whitespaces
```

- **Writing to Files (Append**

Mode): with open("log.txt", "a") as file:
    file.write("New log entry\n")

---

17. Regular Expressions (Regex):

What is Regex?

Regular Expressions (Regex) are patterns used to search for specific text in a string. It helps find, validate, and modify text efficiently.

Think of Regex like **"Ctrl + F"** but much more powerful!

🔹 **Example:**

- Searching for a **phone number** in a document
- Checking if an **email address** is valid
- Replacing **spaces with underscores** in a sentence

How to Use Regex in Python?

Python provides a built-in module called re for working with regular expressions.

First, import the module:

```
import re  # Import the regex module
```

Finding a Word in a Sentence

```
import re

text = "I love learning Python!"
match = re.search(r"Python", text)  # Looks for the word "Python"

if match:
    print("Found 'Python' in the text!")
```

✅ **Explanation:**

- re.search(pattern, text) → Searches for "Python" in the text.
- If found, it prints "Found 'Python' in the text!".

Finding a Number in a Sentence

```
import re

text = "My favorite number is 42."
match = re.search(r"\d+", text)  # Looks for one or more digits

if match:
    print("Found number:", match.group())  # Output: 42
```

✅ **Explanation:**

- \d+ → Matches **one or more digits** (e.g., 42).
- match.group() → Returns the found number.

Checking if an Email is Valid

```
import re

email = "user@example.com"
pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"

if re.match(pattern, email):
    print("Valid email!")
else:
    print("Invalid email!")
```

✅ **Explanation:**

- ^ → Start of the string
- [a-zA-Z0-9_.+-]+ → Letters, numbers, and some special characters
- @ → Must contain @
- [a-zA-Z0-9-]+ → Domain name (e.g., example)
- \.[a-zA-Z0-9-.]+$ → Ends with .com, .org, etc.

Extracting a Date from a Sentence

```
import re

text = "The event is on 15/08/2025."
date = re.search(r"\d{2}/\d{2}/\d{4}", text)

if date:
    print("Found date:", date.group())  # Output: 15/08/2025
```

✅ **Explanation:**

- \d{2}/\d{2}/\d{4} → Finds a date in **dd/mm/yyyy** format.

Checking if a Phone Number is Valid

```
import re

phone_number = "(123) 456-7890"
pattern = r"\(\d{3}\) \d{3}-\d{4}"

if re.match(pattern, phone_number):
    print("Valid phone number!")
else:
    print("Invalid phone number!")
```

✅ **Explanation:**

- \(\d{3}\) → Matches (123).
- \d{3}-\d{4} → Matches 456-7890.

Replacing Spaces with Underscores

```
import re

text = "Hello World!"
modified_text = re.sub(r"\s", "_", text)  # Replaces spaces with underscores

print(modified_text)  # Output: Hello_World!
```

✅ **Explanation:**

- \s → Matches spaces.
- re.sub(pattern, replacement, text) → Replaces spaces with underscores.

Extracting a Website URL from Text

```
import re

text = "Visit our website at https://www.example.com for more info."
url = re.search(r"https?://[a-zA-Z0-9./]+", text)

if url:
    print("Found URL:", url.group())  # Output: https://www.example.com
```

✅ **Explanation:**

- https?:// → Matches http:// or https://.
- [a-zA-Z0-9./]+ → Matches the rest of the URL.

Splitting a Sentence into Words

```
import re

text = "apple, banana, cherry"
fruits = re.split(r",\s*", text)  # Splits by commas and spaces

print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

✅ **Explanation:**

- re.split() → Splits text based on commas and spaces.

Finding All Numbers in a Sentence

```
import re

text = "I have 3 apples, 7 bananas, and 12 cherries."
numbers = re.findall(r"\d+", text)

print("Numbers found:", numbers)  # Output: ['3', '7', '12']
```

✅ **Explanation:**

- \d+ → Finds **all numbers** in the text.
- re.findall() → Returns a **list of all matches**.

Finding Words That Start with "A" or "a"

```
import re

text = "Alice and Alex are amazing artists."
words = re.findall(r"\b[Aa]\w+", text)

print("Words found:", words)  # Output: ['Alice', 'Alex', 'amazing', 'artists']
```

✅ **Explanation:**

- \b → Matches **word boundaries** (start of a word).
- [Aa] → Matches **uppercase or lowercase "A"**.
- \w+ → Matches **the rest of the word**.

Checking if a Password is Strong

A strong password should have:

- At least **8 characters**
- At least **one uppercase letter**
- At least **one lowercase letter**
- At least **one number**
- At least **one special character**

```
import re

password = "Secure@123"
pattern =
r"^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$"

if re.match(pattern, password):
    print("Strong password!")
else:
    print("Weak password! Try adding uppercase, lowercase, numbers, and special characters.")
```

✅ **Explanation:**

- (?=.*[A-Z]) → At least **one uppercase letter**
- (?=.*[a-z]) → At least **one lowercase letter**
- (?=.*\d) → At least **one digit**
- (?=.*[@$!%*?&]) → At least **one special character**
- {8,} → At least **8 characters long**

Extracting All Words from a Sentence

```
import re

text = "Python is fun! Let's learn regex together."
words = re.findall(r"\b\w+\b", text)
```

```
print("Words:", words)
# Output: ['Python', 'is', 'fun', 'Let', 's', 'learn', 'regex', 'together']
```

✅ **Explanation:**

- \b → Word boundary (start and end of a word)
- \w+ → One or more word characters (letters, numbers, underscore)

Extracting All Email Addresses from a Text

```
import re

text = "Contact us at support@example.com or sales@company.org."
emails = re.findall(r"[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+", text)

print("Emails found:", emails)
# Output: ['support@example.com', 'sales@company.org']
```

✅ **Explanation:**

- Looks for text with @ and a **valid domain**.

Extracting All Hashtags from a Tweet

```
import re

tweet = "Learning #Python and #Regex is fun! #Coding"
hashtags = re.findall(r"#\w+", tweet)

print("Hashtags found:", hashtags)
```

```
# Output: ['#Python', '#Regex', '#Coding']
```

✅ **Explanation:**

- #\w+ → Finds words starting with #.

Extracting All Capitalized Words (Proper Nouns)

```
import re

text = "Alice and Bob are learning Python in New York."
capitalized_words = re.findall(r"\b[A-Z][a-z]*\b", text)

print("Capitalized words:", capitalized_words)
# Output: ['Alice', 'Bob', 'Python', 'New',
'York']
```

✅ **Explanation:**

- \b[A-Z][a-z]*\b → Finds words that **start with a capital letter**.

Removing Extra Spaces from a Sentence

```
import re

text = "Python     is    awesome!"
cleaned_text = re.sub(r"\s+", " ", text)  # Replace multiple spaces with a single
space

print(cleaned_text)
# Output: "Python is awesome!"
```

✅ **Explanation:**

- \s+ → Matches **one or more spaces**.

Extracting All Numbers from a String

```
import re

text = "I have 3 apples, 10 bananas, and 25 oranges."
numbers = re.findall(r"\d+", text)

print("Numbers found:", numbers)
# Output: ['3', '10', '25']
```

✅ **Explanation:**

- \d+ → Finds **all numbers**.

Extracting All Words That Start with "T" or "t"

```
import re

text = "The tiger and the turtle are in the zoo."
words = re.findall(r"\b[Tt]\w+", text)

print("Words found:", words)
# Output: ['The', 'tiger', 'the', 'turtle', 'the']
```

✅ **Explanation:**

- \b[Tt]\w+ → Finds words that **start with "T" or "t"**.

Checking if a String Contains Only Letters and Numbers

```
import re

text = "Python123"
if re.fullmatch(r"[A-Za-z0-9]+", text):
   print("Valid input (letters and numbers only)")
else:
   print("Invalid input!")
```

✅ **Explanation:**

- re.fullmatch() → Checks if the **entire string** matches the pattern.
- [A-Za-z0-9]+ → Allows only **letters and numbers**.

Extracting Sentences That End with a Question Mark

```
import re

text = "What is your name? Where do you live? I love Python!"
questions = re
```

---

18. Working with Dates and Times:

Python has a built-in module datetime to work with dates and times.

```
from datetime import datetime

# Get current date and time
now = datetime.now()
print(now)  # Output: 2025-01-25 14:30:45.123456
```

 # Format date and time
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print(formatted) # Output: 2025-01-25 14:30:45

---

19. Modules and Packages:

Modules are Python files containing code that you can import and use in your programs. Packages are collections of modules.

- **Importing a Module**:

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

- **Creating Your Own Module**: Create a file

mymodule.py: def greet(name):
    return f"Hello, {name}!"

Then, in another file:

import mymodule
print(mymodule.greet("Alice")) # Output: Hello, Alice!

---

20. Object-Oriented Programming (OOP) Concepts:

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into "objects," which are instances of classes. It is a popular approach in Python, and understanding its core concepts is essential for any new learner. Here’s an introduction to the main OOP concepts in Python:

OOP is a way of writing programs where we create **objects** that represent real-world things.

Think of a **car**:

- It has **features (data)** → Brand, model, color
- It can **perform actions** → Start, stop, honk

In Python, we use **OOP** to organize our code in a similar way.

---

i) Class: A Blueprint for Objects

A **class** is like a blueprint for creating objects.

Example: Car Class (Blueprint)

```
class Car:
    # Class attribute (shared by all cars)
    wheels = 4

    # Constructor (used to create objects)
    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

    # Method (action)
    def display_info(self):
        print(f"{self.brand} {self.model} has {self.wheels} wheels.")
```

✔ Car is the **class** (blueprint).
✔ brand and model are **attributes** (data).
✔ display_info() is a **method** (action).

---

ii) Object: A Real Example of a Class

An **object** is a real item created from a class.

Example: Creating an Object

```
# Creating an object of class Car
my_car = Car("Toyota", "Corolla")

# Calling a method
my_car.display_info()
```

Output:

Toyota Corolla has 4 wheels.

✔ my_car is an **object** of the Car class.
✔ It has **its own data** (Toyota, Corolla).

---

iii) Attributes: Data Inside an Object

Attributes store **information** inside an object.

- **Instance Attributes** → Unique to each object (e.g., brand, model).
- **Class Attributes** → Shared by all objects (e.g., wheels).

Example: Accessing Attributes

```
print(my_car.brand)  # Instance attribute
print(Car.wheels)    # Class attribute
```

---

iv) Methods: Actions an Object Can Perform

Methods are **functions inside a class** that define what an object can do.

Example: A Dog That Can Bark

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} says Woof!")

dog = Dog("Buddy", "Golden Retriever")
dog.bark()  # Calling a method
```

Output:

Buddy says Woof!

✔ The **bark()** method makes the dog "talk."

---

v) Inheritance: Reusing Code from Another Class

Inheritance allows one class (**child class**) to **reuse** the attributes and methods of another class (**parent class**).

Example: A Dog Inheriting from an Animal

python

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Dog inherits from Animal
```

```
    def speak(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Calls Dog's speak method
```

Output:

Dog barks

✔ The **Dog** class gets its behavior from the **Animal** class.

---

vi) Encapsulation: Hiding Data Inside an Object

Encapsulation **hides** internal data and allows access only through specific methods.

Example: A Car with Private Speed

In this example, we'll create a Car class that has a private attribute for speed, and we'll use methods to control the speed of the car.

```
class Car:
    def __init__(self, brand, speed):
        self.__brand = brand  # Private attribute for brand
        self.__speed = speed  # Private attribute for speed

    def accelerate(self, increment):
        if increment > 0:
            self.__speed += increment
```

```
    def brake(self, decrement):
        if decrement > 0 and self.__speed - decrement >= 0:
            self.__speed -= decrement

    def get_speed(self):
        return self.__speed

    def get_brand(self):
        return self.__brand
```

Explanation:

Private Attributes (__brand, __speed):

- self.__brand and self.__speed are private attributes of the Car object. They cannot be accessed directly from outside the class. This helps protect the internal state of the car.

Methods for Controlling Speed:

- accelerate(self, increment): This method increases the car's speed by a given increment (only if the increment is positive).
- brake(self, decrement): This method decreases the car's speed by a given decrement (only if the decrement is positive and the resulting speed does not go below 0).
- get_speed(self): This method allows you to access the current speed of the car.

- get_brand(self): This method allows you to access the brand of the car.

Using the Car Class:

```
my_car = Car("Toyota", 50) # Create a Car object with brand "Toyota" and speed 50

print(my_car.get_speed()) # Output: 50


my_car.accelerate(30) # Accelerate the car by 30

print(my_car.get_speed()) # Output: 80


my_car.brake(20) # Apply the brake to reduce speed by 20

print(my_car.get_speed()) # Output: 60
```

Output:

```
50

80

60
```

Key Points:

- **Private Data (__speed, __brand)**: The speed and brand are private, meaning they can't be changed directly from outside the class. This ensures that the internal state of the car is protected.
- **Methods for Controlled Access**:

- accelerate() and brake() methods control how the speed changes, ensuring that the speed doesn't go below zero and that it only increases when requested.
- get_speed() and get_brand() provide a way to safely access the car's data.

Why This Is Encapsulation:

- The **internal data** (like the car's speed and brand) is **hidden** inside the object and can only be accessed or modified through **methods** like accelerate, brake, and get_speed.
- This ensures that the car's state is always valid (e.g., speed cannot go below zero), and any changes to the car's data are controlled and safe.

---

vii) Polymorphism: One Method, Different Behaviors

Polymorphism allows different classes to have **methods with the same name** but different behaviors.

Example: Different Animals Making Different Sounds

```
class Cat:
    def speak(self):
        print("Meow")

class Dog:
    def speak(self):
        print("Woof")

animals = [Cat(), Dog()]

for animal in animals:
    animal.speak()  # Different behavior based on object type
```

Output:

Meow
Woof

✔ Both Cat and Dog have a speak() method, but they behave differently.

---

viii) Abstraction: Hiding Complex Details

Abstraction hides unnecessary details and shows only what's important.

Example: Using an Abstract Class

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract class
    @abstractmethod
    def speak(self):
        pass  # Must be defined in child classes

class Dog(Animal):
    def speak(self):
        print("Woof")

dog = Dog()
dog.speak()
```

Output:

Woof

✔ The Animal class **only defines** speak(), but doesn't implement it.
✔ The Dog class **must** define speak().

Summary of OOP Concepts in Python

- **Class** → A blueprint for creating objects.
- **Object** → A real instance of a class.
- **Attributes** → Data stored in an object.
- **Methods** → Actions an object can perform.
- **Inheritance** → Allows one class to reuse code from another class.
- **Encapsulation** → Hides data and restricts access to it.
- **Polymorphism** → One method name, different behaviors.
- **Abstraction** → Hides complex details and shows only important parts.

Why Use OOP?

✅ Makes code **organized** and **easier to manage**
✅ Helps **reuse code** (Inheritance)
✅ Improves **security** (Encapsulation)
✅ Allows **flexibility** (Polymorphism)

---

21. Decorators:

Decorators are a way to modify or enhance functions or methods without changing their actual code.

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
```

```
@decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Before function call
Hello!
After function call
```

What Happens Step-by-Step

1. The @decorator syntax applies the decorator function to say_hello. It replaces say_hello with the wrapper function returned by the decorator.
2. When say_hello() is called:
    - The wrapper function runs.
    - Inside the wrapper, the message **"Before function call"** is printed.
    - The original say_hello function is executed, printing **"Hello!"**.
    - After the original function, the message **"After function call"** is printed.

Output

```
Before function call
Hello!
After function call
```

Why Use Decorators?

- **Reusability**: Add the same behavior (e.g., logging, authentication) to multiple functions without repeating code.
- **Clean Code**: Keep the original function's logic separate from additional functionality.
- **Flexibility**: Apply or remove behaviors easily by adding or removing decorators.

---

22. Understanding Iterators and Generators:

Iterator:

- An iterator is like a tool that helps you go through items in a collection (like a list or a tuple) one by one.
- It keeps track of where it is in the collection.
- You can create an iterator from a collection using the iter() function.
- To get the next item, you use the next() function. When there are no more items, next() will stop the iteration and raise a StopIteration error.

Example:

```
numbers = [1, 2, 3]
iterator = iter(numbers)
print(next(iterator))  # Output: 1
```

Generator:

- A generator is a special type of function that returns an iterator. It uses the yield keyword to return values one at a time.
- Instead of running the function completely, a generator "pauses" at the yield statement and gives back a value. The next time it is called, it picks up from where it left off.
- Generators are useful because they don’t generate all the values at once, saving memory.

Example:

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

counter = count_up_to(3)
for num in counter:
    print(num)  # Output: 1, 2, 3
```

Key Differences:

- **Iterator**: It's an object (like a list) that can be iterated over.
- **Generator**: It's a function that generates values one at a time using yield.

Why use Generators?

- **Memory Efficient**: Generators don't store all the values at once. They generate each value only when needed.
- **Great for Large Datasets**: They're useful when working with large amounts of data or infinite sequences because they don't take up a lot of memory.

---

23. List and Dictionary Comprehensions (Advanced):

List and dictionary comprehensions are powerful tools for creating and transforming data structures in a concise way.

- **List Comprehension**: Create a list by applying an expression to each item in an existing iterable.

```
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
```

```
print(squares)  # Output: [1, 4, 9, 16, 25]
```

- **Dictionary Comprehension**: Similar to list comprehension, but it creates a dictionary.

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
age_dict = {name: age for name, age in zip(names, ages)}
print(age_dict)  # Output: {'Alice': 25, 'Bob': 30, 'Charlie': 35}
```

---

24. Enumerate:

enumerate() is a built-in function that allows you to loop over an iterable and get both the index and the value.

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# Output:
# 0: apple
# 1: banana
# 2: cherry
```

---

25. Zip:

zip() is a built-in function that combines multiple iterables (like lists or tuples) into a single iterable of tuples.

```
names = ["Alice", "Bob", "Charlie"]
```

```
scores = [85, 90, 95]
combined = zip(names, scores)
print(list(combined))  # Output: [('Alice', 85), ('Bob', 90), ('Charlie', 95)]
```

---

26. Set Operations:

Sets are unordered collections of unique elements. You can perform mathematical operations like union, intersection, and difference on sets.

- **Union**: Combines two sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

- **Intersection**: Finds common elements between two

sets. intersection_set = set1 & set2
print(intersection_set) # Output: {3}

- **Difference**: Finds elements in the first set that are not in the

second. difference_set = set1 - set2
print(difference_set) # Output: {1, 2}

---

27. Handling Timeouts and Delays (time.sleep):

Sometimes, you need to pause your program for a specific amount of time. You can do this using time.sleep().

```
import time

print("Starting...")
time.sleep(2)  # Waits for 2 seconds
print("2 seconds later...")
```

---

28. Context Managers (with statement):

Context managers are used to set up and tear down resources automatically. The with statement is commonly used for managing files, database connections, and more.

```
with open("example.txt", "w") as file:
    file.write("This is a test file.")
# No need to manually close the file; it will be closed automatically after the block.
```

---

29. Multi-threading:

Python allows you to run multiple threads (tasks) concurrently. This is useful for I/O-bound tasks.

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_numbers)
```

```
# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()
```

Use cases where multi-threading in Python can be highly beneficial:

i. Web Scraping

- **Scenario**: Scraping data from multiple web pages concurrently.
- **Why Multi-threading**: Web scraping often involves waiting for HTTP responses, which is an I/O-bound task. By using threads, you can scrape multiple pages at the same time, improving efficiency.
- **Example**: Using multiple threads to scrape data from different URLs concurrently.

ii. Downloading Files

- **Scenario**: Downloading multiple files from the internet at the same time.
- **Why Multi-threading**: File downloads are typically I/O-bound tasks. Running multiple threads allows you to download files concurrently, reducing the total time required.
- **Example**: Downloading multiple files from different servers or URLs concurrently.

iii. Parallel Data Processing

- **Scenario**: Processing large datasets or performing computationally expensive tasks (e.g., image processing, video encoding).
- **Why Multi-threading**: If the task involves multiple independent operations (e.g., processing different chunks of data), threads can be used to handle these tasks concurrently, speeding up the process.
- **Example**: Processing multiple images or videos at the same time using separate threads.

iv. Server Handling Multiple Requests

- **Scenario**: A server that needs to handle multiple client requests concurrently.
- **Why Multi-threading**: A web server can use threads to handle each incoming request independently. This allows the server to serve multiple clients at once without blocking.
- **Example**: A web server where each incoming HTTP request is handled by a separate thread.

v. Real-time Applications

- **Scenario**: Handling multiple real-time inputs or events (e.g., from sensors, user input, or network data).
- **Why Multi-threading**: In real-time applications, you may need to handle multiple events or inputs concurrently without blocking the main program flow.
- **Example**: A game that handles user input, network data, and game logic in separate threads.

vi. Background Tasks

- **Scenario**: Running background tasks while the main application continues to run.
- **Why Multi-threading**: Some tasks, like periodic data syncing, logging, or monitoring, can be handled in the background while the main application remains responsive.
- **Example**: A chat application that runs background tasks (like checking for new messages) in a separate thread while the user interacts with the UI.

vii. Database Operations

- **Scenario**: Performing multiple database queries concurrently.
- **Why Multi-threading**: Database operations often involve waiting for I/O responses. By using threads, you can perform multiple queries concurrently, reducing the overall time taken to complete the operations.
- **Example**: Running multiple database queries in parallel to fetch data from different tables or databases.

viii. Machine Learning Model Training

- **Scenario**: Training multiple machine learning models or running hyperparameter tuning concurrently.
- **Why Multi-threading**: Training different models or running experiments with different hyperparameters can be done concurrently using threads, speeding up the experimentation process.
- **Example**: Running multiple training jobs for different models or hyperparameter configurations in parallel.

ix. Chatbots or Virtual Assistants

- **Scenario**: Handling multiple user queries simultaneously in a chatbot or virtual assistant.

- **Why Multi-threading**: Each user query can be processed in a separate thread, allowing the assistant to handle multiple conversations at the same time without blocking.
- **Example**: A chatbot that responds to user queries concurrently in a multi-threaded environment.

x. Real-time Data Monitoring

- **Scenario**: Monitoring multiple sources of real-time data (e.g., sensors, logs, system metrics).
- **Why Multi-threading**: Each data source can be monitored in a separate thread, allowing the system to collect and process data concurrently without blocking.
- **Example**: A system that monitors CPU usage, memory usage, and network traffic in separate threads.

---

30. Decorators with Arguments:

What is a Decorator?

A **decorator** is a special function in Python that allows you to **modify** or **enhance** another function without changing its actual code. Think of it like a wrapper that you can put around a function to add extra behavior.

What are Decorators with Arguments?

Decorators can also accept **arguments** (like regular functions do). This makes them even more powerful and flexible because you can customize their behavior based on the values you pass to them.

Simple Example: Repeating a Function

Imagine you have a function that says "Hello", but you want it to say "Hello" multiple times. Instead of writing the same code over and over, you can use a decorator to repeat it for you.

You can create decorators that accept arguments to make them more flexible.

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

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

What Happens Here?

1. **repeat(n)**: This is the main decorator. It takes an argument n (the number of times you want the function to repeat).
2. **decorator(func)**: This function takes the original function greet and wraps it with additional behavior.
3. **wrapper(*args, **kwargs)**: This is the function that actually does the repeating. It runs the original function greet n times.

4. **@repeat(3)**: This is the decorator syntax. It means "decorate the greet function with the repeat decorator, repeating it 3 times."

Output:

Hello, Alice!
Hello, Alice!
Hello, Alice!

Why Use Decorators with Arguments?

Using decorators with arguments gives you **flexibility**. You can customize how the decorator behaves by passing different values. For example, you can make the decorator repeat the function 5 times, 10 times, or just once, depending on what you need.

---

31. Handling JSON Data:

JSON (JavaScript Object Notation) is a common format for exchanging data between systems. Python has a json module to handle JSON data.

Convert Python objects to JSON:

```
import json

person = {"name": "Alice", "age": 25}
json_data = json.dumps(person)
print(json_data)  # Output: {"name": "Alice", "age": 25}
```

Output:

json

{"name": "Alice", "age": 25}

Convert JSON to Python objects:

```
json_string = '{"name": "Alice", "age": 25}'
person = json.loads(json_string)
print(person)  # Output: {'name': 'Alice', 'age': 25}
```

Output:

```
{'name': 'Alice', 'age': 25}
```

---

32. Working with CSV Files:

CSV (Comma-Separated Values) is a simple format for storing tabular data. Python's csv module helps read and write CSV files.

Reading from a CSV file:

```
import csv

with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output (assuming data.csv contains the following data):

```
['name', 'age']
['Alice', '25']
['Bob', '30']
```

Writing to a CSV file:

```
import csv

data = [["name", "age"], ["Alice", 25], ["Bob", 30]]
with open('output.csv', 'w', newline='') as file:
    writer =
    csv.writer(file)
    writer.writerows(data)
```

Output: The output.csv file will contain:

```
name,age
Alice,25
Bob,30
```

---

33. Handling Large Data with Pandas:

Pandas is a powerful library for working with large datasets in a table-like structure called DataFrame.

```
import pandas as pd

# Create a DataFrame
data = {"name": ["Alice", "Bob", "Charlie"], "age": [25, 30, 35]}
df = pd.DataFrame(data)

# Display the DataFrame
print(df)

# Accessing specific columns
print(df["name"])

# Filtering rows based on a condition
print(df[df["age"] > 30])
```

Output:

markdown

```
    name  age
0    Alice   25
1      Bob   30
2  Charlie   35

0     Alice
1       Bob
2    Charlie
Name: name, dtype: object

     name  age
2  Charlie   35
```

---

34. Working with SQLite (Database):

SQLite is a simple, file-based database. Python's sqlite3 module lets you interact with SQLite databases.

```
import sqlite3

# Connect to a database (or create one if it doesn't exist)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGER
PRIMARY KEY, name TEXT, age INTEGER)''')

# Insert data into the table
cursor.execute('''INSERT INTO users (name, age) VALUES ('Alice', 25)''')
conn.commit()
```

```
# Query data from the table
cursor.execute('''SELECT * FROM users''')
print(cursor.fetchall())  # Output: [(1, 'Alice', 25)]

# Close the connection
conn.close()
```

Output:

```
[(1, 'Alice', 25)]
```

---

35. Working with APIs (Requests Library):

The requests library makes it easy to send HTTP requests (like GET, POST) and work with the responses.

API stands for **Application Programming Interface**. It allows one application to communicate with another. For example, when you use an app like Facebook or Twitter, it communicates with their servers through APIs to fetch data (like posts, user information, etc.).

When you want to interact with an API, you send an **HTTP request** (like GET, POST) to a server, and it responds with data.

GET Request:

A **GET request** is used to fetch data from a server. It's like asking a server, "Please send me this information."

Code Explanation:

```
import requests
```

This line imports the requests library, which makes it easy to send HTTP requests.

response = requests.get("https://api.github.com")

Here, we're sending a **GET request** to the GitHub API. The URL "https://api.github.com" is the address of the GitHub API that provides information about the current status of GitHub.

print(response.status_code)

After sending the GET request, the server responds. The status_code tells us whether the request was successful. A 200 status code means "OK" (the request was successful).

Output:

200

print(response.json())

The .json() method converts the response from the server into a Python dictionary. The data returned by the server is usually in **JSON format**, which is a way of structuring data that is easy to read and write for both humans and machines.

Output:

json
{
"current_user_url": "https://api.github.com/user",
"current_user_authorizations_html_url":
"https://github.com/settings/connections/applications{/client_id}",
...
}

This is the data returned by the GitHub API. It contains links and information about the current user, such as the URL to access the current user's details.

POST Request:

A **POST request** is used to send data to a server. It's like saying, "Here's some data, please save it or process it."

Code Explanation:

data = {"name": "Alice", "age": 25}

Here, we create a dictionary data that contains the information we want to send to the server. In this case, it's a simple dictionary with a person's name and age.

response = requests.post("https://api.example.com", json=data)

This sends a **POST request** to "https://api.example.com", along with the data dictionary in JSON format. The server will process this data (maybe save it or perform some action).

print(response.status_code)

After the POST request is made, the server responds again. The status_code tells us if the request was successful. A 201 status code means "Created," which typically indicates that the data was successfully created or saved.

Output:

201

Summary:

- **GET request**: Used to fetch data from a server. The server sends back information, and you can read it.
- **POST request**: Used to send data to a server. The server processes the data and responds.

The requests library simplifies the process of interacting with APIs in Python. You can use it to both fetch and send data to web services, which is an essential part of working with modern web applications.

---

36. Working with HTML and Web Scraping (BeautifulSoup):

i) What is BeautifulSoup?

BeautifulSoup is a Python library that makes it easy to extract data from HTML and XML documents. It's especially useful for web scraping, where you want to retrieve data from websites.

ii) How does it work?

Web scraping involves fetching the HTML content of a webpage and then parsing it to extract useful information. BeautifulSoup helps you navigate and search through the HTML structure in a readable way.

iii) Example Breakdown:

```
from bs4 import BeautifulSoup
import requests
```

- **from bs4 import BeautifulSoup**: This imports the BeautifulSoup class from the bs4 module, which is the library used for parsing HTML.
- **import requests**: This imports the requests module, which is used to send HTTP requests to fetch the content of a webpage.

```
response = requests.get("https://example.com")
```

```
soup = BeautifulSoup(response.text, 'html.parser')
```

- **requests.get("https://example.com")**: This sends a request to the URL "https://example.com". The requests.get() function retrieves the HTML content of that page.
- **response.text**: This is the HTML content of the page returned by the requests.get() function.
- **BeautifulSoup(response.text, 'html.parser')**: This creates a BeautifulSoup object (soup) that parses the HTML content using the 'html.parser' method. Now, soup holds a structured representation of the HTML page.

```
for link in soup.find_all('a'):
    print(link.get('href'))
```

- **soup.find_all('a')**: This searches the HTML content for all <a> tags, which are used to define hyperlinks.
- **link.get('href')**: For each <a> tag found, this retrieves the value of the href attribute, which contains the URL of the link.
- **print(link.get('href'))**: This prints each URL (or link) found on the page.

Output:

If the webpage contains links like:

```
<a href="https://www.example.com/page1">Page 1</a>
<a href="https://www.example.com/page2">Page 2</a>
```

The output of the script will be:

```
https://www.example.com/page1
https://www.example.com/page2
```

5. Key Concepts for a New Learner:

- **HTML Tags**: HTML pages are structured with tags like <a>, , <p>, etc. These tags define the content and structure of the page.
- **requests Module**: This is used to fetch content from the web.
- **BeautifulSoup**: This is used to parse the fetched HTML content and extract useful information from it.
- **Navigating HTML**: Using methods like find_all() allows you to search for specific tags and attributes in the HTML.

Summary:

This example demonstrates how to use BeautifulSoup and requests to scrape a webpage, extract all the links (<a> tags), and print their URLs. It's a simple introduction to web scraping, where you retrieve and process web data using Python.

---

37. Multiprocessing

The multiprocessing module helps you run multiple processes simultaneously, which is useful for tasks that need a lot of CPU power. This can make your program run faster by doing multiple things at once.

Example:

```
from multiprocessing import Process

# Function to print numbers from 0 to 4
def print_numbers():
    for i in range(5):
        print(i)

# Create a process to run the print_numbers function
process = Process(target=print_numbers)

# Start the process
```

```
process.start()

# Wait for the process to finish
process.join()
```

- Process(target=print_numbers) creates a new process to run the print_numbers function.
- process.start() begins running the function in a separate process.
- process.join() makes the program wait until the process finishes before moving on.

---

38. Asyncio (Asynchronous Programming)

The asyncio module allows you to write code that can perform tasks while waiting for something else to finish (like reading a file or making a network request). This makes your program more efficient because it can do other work instead of waiting.

Example:

```
import asyncio

# Asynchronous function that sleeps for 1 second and then prints a message
async def greet():
    await asyncio.sleep(1)
    print("Hello, Async!")

# Run the asynchronous function
asyncio.run(greet())
```

- async def defines an asynchronous function.
- await asyncio.sleep(1) pauses the function for 1 second, but allows other tasks to run during this time.

- asyncio.run(greet()) runs the asynchronous function.

---

39. Custom Exceptions

Custom exceptions allow you to create your own error types that are specific to your program. This helps make error handling more meaningful and easier to manage.

Example:

```
# Define a custom exception class
class CustomError(Exception):
    pass

# Raise the custom exception
try:
    raise CustomError("An error occurred!")
except CustomError as e:
    print(e)
```

- class CustomError(Exception) creates a new exception class.
- raise CustomError("An error occurred!") raises the custom error.
- except CustomError as e catches the error and prints the message.

---

40. YAML Files

YAML is a simple format for storing data, often used for configuration files. You can use the PyYAML library to read and write YAML files in Python.

Example:

```
import yaml

# Data to be written to a YAML file
data = {"name": "Alice", "age": 25}

# Convert Python dictionary to YAML string
yaml_string = yaml.dump(data)

# Print the YAML string
print(yaml_string)
```

- yaml.dump(data) converts the Python dictionary to a YAML-formatted string.
- print(yaml_string) displays the YAML content.

---

42. Type Hinting

Type hinting in Python helps make your code easier to understand by showing

what types of values variables and function arguments should have. It can also help catch errors when using tools like mypy.

Example:

```
# Function that adds two integers and returns an integer
def add(a: int, b: int) -> int:
    return a + b
```

- a: int and b: int specify that both a and b are integers.
- -> int indicates that the function will return an integer.

---

43. ConfigParser

The configparser module is used to work with .ini configuration files, which store settings for your program. You can read, write, and modify these files using this module.

Example:

```
import configparser

# Create a ConfigParser object
config = configparser.ConfigParser()

# Read the configuration file
config.read("config.ini")

# Access a setting from the DEFAULT section
print(config["DEFAULT"]["Setting"])
```

- config.read("config.ini") reads the configuration file.
- config["DEFAULT"]["Setting"] accesses a setting from the DEFAULT section.

---

44. Command-Line Interfaces (argparse)

The argparse module helps you create command-line tools that accept arguments from the user. This is useful for building programs that can be run from the terminal with different options.

The argparse module helps you create programs that can accept input directly from the terminal (or command line). This is useful when you want your program to be flexible and run with different options.

Example:

```
import argparse

# Step 1: Create an ArgumentParser object
parser = argparse.ArgumentParser(description="A simple CLI tool.")

# Step 2: Add an argument for the user's name
parser.add_argument("--name", type=str, help="Enter your name.")

# Step 3: Parse the arguments entered by the user
args = parser.parse_args()

# Step 4: Print a greeting message with the user's name
print(f"Hello, {args.name}!")
```

Explanation:

- **Step 1**: We create an ArgumentParser object that will handle the command-line arguments.
- **Step 2**: We add an argument called --name, which allows the user to provide their name when running the program. The help part gives a description of what this argument is for.
- **Step 3**: We use parse_args() to read the arguments the user provides when running the program.
- **Step 4**: We print a message that greets the user by the name they provided.

How to Use:

When you run the program, you can provide your name like this:

python your_program.py --name John

This will output:

Hello, John!

In this example, --name is the argument, and John is the value that the program uses to print the greeting.

---

45. Advanced String Formatting (f-strings)

F-strings, introduced in Python 3.6, are a way to embed expressions inside string literals, making string formatting simpler and more readable. They allow you to directly insert variables or expressions inside a string without needing to concatenate or use formatting methods like .format() or %.

Key Features of F-strings:

- **Cleaner and More Readable**: F-strings make it easier to include variables and expressions inside strings.
- **Performance**: F-strings are faster than older string formatting methods like % formatting or .format().
- **Inline Expressions**: You can even include expressions (like calculations) inside the curly braces {}.

Basic Syntax:

The syntax for an f-string is to prefix the string with the letter f or F and use curly braces {} to insert variables or expressions directly inside the string.

Example 1: Basic Variable Insertion

```
name = "Alice"
```

```
age = 25

# Use an f-string to format the message

print(f"{name} will be {age + 5} years old in 5 years.")
```

Output:

- Alice will be 30 years old in 5 years.

Here, name and age are variables, and inside the f-string, we can directly reference them. We can also perform calculations inside the curly braces, such as age + 5.

Example 2: Inserting Expressions

You can include any valid Python expression inside the curly braces.

```
width = 5

height = 10


# Calculate the area directly inside the f-string

print(f"The area of the rectangle is {width * height}.")
```

Output:

- The area of the rectangle is 50.

Example 3: Formatting Numbers

F-strings also allow you to format numbers, dates, or other data types directly inside the string.

```
pi = 3.14159
```

```
# Format pi to 2 decimal places
print(f"Pi rounded to two decimal places is {pi:.2f}.")
```

Output:

```
Pi rounded to two decimal places is 3.14.
```

In this case, :.2f is a formatting specifier that rounds the number to two decimal places.

Advantages of F-strings:

- **More Readable**: It's easy to understand because the variables and expressions are directly in the string.
- **Fewer Errors**: There's no need to worry about mismatched parentheses or formatting placeholders.
- **Performance**: F-strings are faster than using the .format() method or % formatting.

F-strings are an essential feature for anyone learning Python, as they make string manipulation more intuitive and efficient.