PYTHON
Programming
An beginner Guide

MACHHA ISHANK REDDY

About the Author

Hi, I'm Ishank,

I specialize in **HTML**, **JavaScript**, and back-end technologies like **Node.js**, **Python**, and databases. My expertise lies in creating **dynamic, user-friendly AI web applications** by seamlessly integrating front-end and back-end technologies.

My work primarily focuses on:

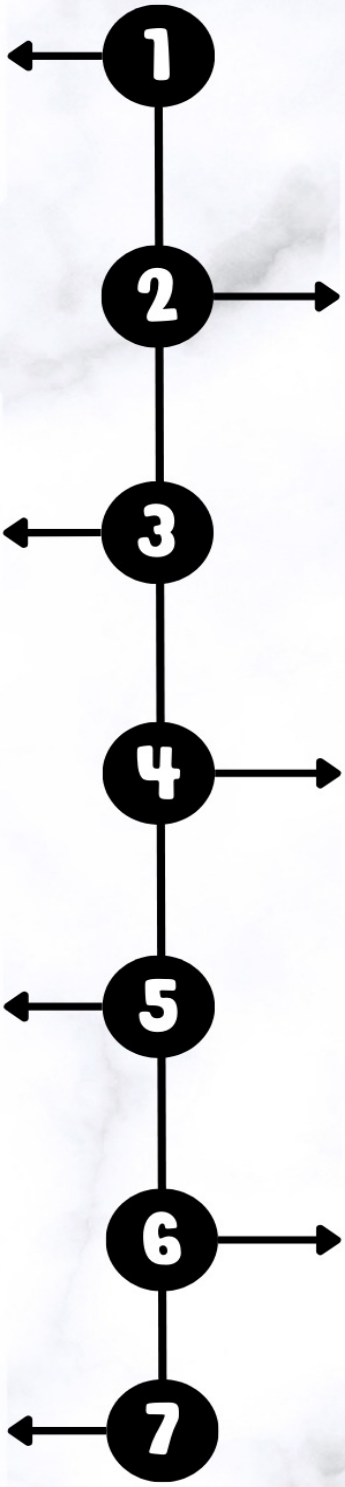
- **Generative AI**
- **Chatbot Design**
- **Machine Learning**

With a passion for innovation, I strive to build solutions that enhance user experiences and push the boundaries of AI technology.

TABLE OF CONTENT

INTRODUCTION

1. What is Python
2. What can Python do
3. Python Installation
   VS Code Installation
   Python Installtion
   Run Hello world Program



PYTHON BASICS

1. Variables
2. Data Types
3. Conditions and If statements
4. Loops
5. Functions

OBJECT ORIENTED PROGRAMMING

1. Classes/Objects
2. Inheritance
3. Function Polymorphism
4. Scope
5. Modules
6. Dates & Math
7. JSON
8. RegEx
9. PIP
10. User Input
11. String Fomatting

FILE HANDLING

1. File Open
2. Read an Existing File
3. Write to an Existing File
4. Delete a File

EXCEPTION HANDLING

1. The try block
2. The except block
3. The else block
4. The finally block

PYTHON MODULES

1. NumPy
2. Pandas
3. SciPy
4. FAST API

FULL STACK PROJECT

Html
Java script
Python FAST API

Python Introduction

What is 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 emphasizes code readability, using significant indentation to define blocks of code rather than curly brackets or keywords.

Key Features of Python

1. **Easy to Learn and Use**: Python's simple syntax makes it an excellent language for beginners.
2. **Interpreted Language**: Python executes code line-by-line, making debugging easier.
3. **Cross-platform**: Python runs on various platforms such as Windows, Mac, Linux, etc.
4. **Dynamic Typing**: You don't need to declare variable types. Python determines them at runtime.
5. **Extensive Libraries**: Python has a wide range of libraries and frameworks, such as NumPy, Pandas, Django, and Flask, which make development faster and more efficient.
6. **Open Source**: Python is free to use and distribute, even for commercial purposes.
7. **Community Support**: Python has a vast and active community that contributes to its growing ecosystem.

Applications

- **Web Development**: Using frameworks like Django or Flask.
- **Data Science & Machine Learning**: With libraries like NumPy, Pandas, Scikit-learn, and TensorFlow.
- **Automation**: Automating repetitive tasks with Python scripts.
- **Scripting**: Python is frequently used for writing scripts to automate tasks like file handling, network connections, etc.
- **Game Development**: Using libraries like Pygame.

Python Installation

1. Download Python

• Visit the official Python website: https://www.python.org/downloads/
• Download the latest version for your operating system (Windows, macOS, or Linux).

2. Install Python

• **Windows**: Run the installer and check the box that says “Add Python to PATH.” Then click “Install Now.”
• **macOS**: Open the .pkg file and follow the instructions.
• **Linux**: You can use your package manager. For example, on Ubuntu, run:

```bash
sudo apt update
sudo apt install python3
```

3. Verify the Installation

• Open a terminal (Command Prompt on Windows) and type:

```bash
python --version
```

Python Installation for VS Code

To install Python and set up Visual Studio Code (VS Code) for Python development, follow these steps:

1. Install Python

- Visit the official Python website: https://www.python.org/downloads/.
- Download and install the latest version of Python.
- Make sure to check the box **"Add Python to PATH"** during installation.

2. Install Visual Studio Code (VS Code)

- Download and install VS Code from https://code.visualstudio.com/.
- Follow the installation instructions for your operating system.

3. Install Python Extension in VS Code

- Open VS Code.
- Go to the **Extensions** view by clicking on the square icon in the sidebar or press Ctrl + Shift + X.
- Search for **Python** and install the Python extension provided by Microsoft.

4. Install Code Runner Extension in VS Code

- Open VS Code
- Go to the **Extensions** view by clicking on the square icon in the sidebar or press Ctrl + Shift + X.
- Search for **Code Runner** and install the extension.
- Run any Python file you want

Python Getting Started

Python QuickStart:

Python is an interpreted programming language. this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

python helloworld.py

Example:

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

Output of above Program:

```
Hello, World!
```

Python Syntax

1. Execute Python Code

Python code can be executed by running a .py file in a terminal or directly in an IDE.

Example Code (`hello.py`)

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

To execute:

1. Save the code in a file named `hello.py`.
2. Open a terminal or command prompt.
3. Navigate to the folder where you saved the file.
4. Run this command:

```
python hello.py
```

Output:

```
Hello, World!
```

Python Syntax

2. Variables

In Python, variables don't need to be explicitly declared. You simply assign a value to a variable, and Python infers the type.

Here's an example:

```python
# Variable assignments
name = "John"        # String
age = 25             # Integer
height = 5.9         # Float
is_student = True    # Boolean

# Output the variables
print(name)
print(age)
print(height)
print(is_student)
```

Output:

```graphql
John
25
5.9
True
```

Python Syntax

3. Comments

Comments in Python begin with a # symbol and are ignored by the interpreter. They're used to explain code.

Here's a simple example:

```python
# This is a comment
x = 5  # This is another comment

# Print the value of x
print(x)
```

Explanation:

- `# This is a comment` is ignored by Python.
- `x = 5` assigns the value `5` to the variable `x`.
- `print(x)` prints the value of `x`, which is `5`.

Output:

```
5
```

Python Syntax

4. Indentation

Python uses indentation (spaces or tabs) to define code blocks. Unlike other languages, it doesn’t use curly braces.

Here's the same code without comments:

```python
def say_hello():
    print("Hello!")

say_hello()
```

Explanation:

- The line `print("Hello!")` is indented (by 4 spaces or a tab), so it is part of the `say_hello` function.
- Python will raise an error if the indentation is incorrect.

Output:

```
Hello!
```

Python Variables and Data Types

1. Numbers

- **Integers** are whole numbers, and they don't have decimal points. In Python, we can just assign a number to a variable, and Python will automatically treat it as an integer.

Example:

```python
age = 25  # Integer
print(age)
print(type(age))
```

Explanation:

- `age = 25`: Here, we are assigning the number `25` to the variable `age`.
- `print(age)`: This will output the value of `age`, which is `25`.
- `print(type(age))`: This will show the data type of `age`, which is `<class 'int'>`, meaning it's an integer.

Output:

```arduino
25
<class 'int'>
```

Python Variables and Data Types

- **Floats** are numbers that have decimals. In Python, any number with a decimal point is considered a float.

  Example:

```python
price = 19.99  # Float
print(price)
print(type(price))
```

Explanation:

- `price = 19.99`: We are assigning the number `19.99` to the variable `price`. Since it has a decimal, it's treated as a float.
- `print(price)`: This will output the value `19.99`.
- `print(type(price))`: This will display the data type of `price`, which is `<class 'float'>`.

Output:

```arduino
19.99
<class 'float'>
```

Python Variables and Data Types

2. **Strings**

- **Strings** are sequences of characters (like words or sentences). In Python, we define them using single or double quotes.

  Example:

```python
name = "Alice"  # String
print(name)
print(type(name))
```

Explanation:

- `name = "Alice"`: We are assigning the string `"Alice"` to the variable `name`.
- `print(name)`: This will output the text stored in `name`, which is `"Alice"`.
- `print(type(name))`: This shows that the data type of `name` is `<class 'str'>`, meaning it is a string.

Output:

```arduino
Alice
<class 'str'>
```

Python Variables and Data Types

Python allows both single (`'`) and double (`"`) quotes for strings. Both work the same way.

Example:

```python
greeting = 'Hello, world!'
print(greeting)
```

Explanation:

- `greeting = 'Hello, world!'` : Here we use single quotes, but the result is the same. We are storing the string `"Hello, world!"` in the variable `greeting` .
- `print(greeting)` : This will print the text `"Hello, world!"` .

Output:

```
Hello, world!
```

Python Variables and Data Types

3. **Booleans**

- **Booleans** are used to represent two values: `True` or `False`. They are helpful in situations where we need to check conditions or set flags (e.g., is the user logged in or not).

  Example:

```python
is_sunny = True  # Boolean
print(is_sunny)
print(type(is_sunny))
```

Explanation:

- `is_sunny = True`: We are assigning the boolean value `True` to the variable `is_sunny`.
- `print(is_sunny)`: This will output the value `True`.
- `print(type(is_sunny))`: This shows that the data type of `is_sunny` is `<class 'bool'>`, meaning it is a boolean.

Output:

```python
True
<class 'bool'>
```

Python Type Conversion

1. Implicit Type Conversion

Implicit type conversion happens automatically in Python when you perform operations between different types, and Python converts the smaller data type into a larger one to prevent data loss. For example, if you add an `int` to a `float`, Python will convert the integer into a float automatically.

Example:

python

```python
# Implicit Type Conversion Example
num_int = 5   # Integer
num_float = 2.5  # Float

# Adding an int and a float
result = num_int + num_float

print(result)  # The output will be a float
print(type(result))  # Check the type of the result
```

Explanation:

- `num_int` is an integer, and `num_float` is a float.
- When you add them together (`5 + 2.5`), Python automatically converts the integer `5` into a float so that both numbers are of the same type, avoiding any loss of information.
- The result is `7.5`, which is a float because Python preserves the more precise data type.

Python Type Conversion

Output:

```arduino
7.5
<class 'float'>
```

---

2. Explicit Type Conversion

Explicit type conversion (also called typecasting) is when you manually convert a data type to another. You use specific functions like `int()`, `float()`, `str()`, and `bool()` to force Python to change the type.

Examples of Explicit Type Conversion

Example 1: Convert Float to Integer

```python
# Explicit conversion from float to int
num_float = 7.8
num_int = int(num_float)  # Convert float to int

print(num_int)
```

Python Type Conversion

Explanation:

- The `float` value `7.8` is converted to an `int` using the `int()` function.
- **Important**: When converting a float to an integer, the decimal part is truncated (not rounded), meaning `7.8` becomes `7`.

Output:

```
7
```

Example 2: **Convert to Boolean**

Booleans (`True` or `False`) represent binary values. You can convert various data types into booleans using the `bool()` function.

python

```python
# Convert different types to boolean
print(bool(1))    # True
print(bool(0))    # False
print(bool("Hello"))  # True
print(bool(""))   # False
```

Python Type Conversion

Explanation:

- `bool(1)` : In Python, any non-zero number is considered `True` , so `1` becomes `True` .
- `bool(0)` : The value `0` is considered `False` .
- `bool("Hello")` : Any non-empty string is `True` . So, the string `"Hello"` evaluates to `True` .
- `bool("")` : An empty string evaluates to `False` .

Output:

```graphql
True
False
True
False
```

Key Points:

- For numbers, any non-zero value is `True` , while `0` is `False` .
- For strings, any non-empty string is `True` , while an empty string (`""`) is `False` .

Python Input

In Python, **input** is a function that allows the user to provide some data during the program's execution. The function collects the user's input as a string, and you can assign this input to a variable for further processing.

Here's a basic example:

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

Explanation:

1. `input("Enter your name: ")`:
   - This line prompts the user to type their name. The text inside the parentheses ("Enter your name: ") is shown to the user as a prompt.
   - Whatever the user types is captured as a string.
2. `name = input(...)`:
   - The entered string is stored in the variable `name`.
3. `print("Hello, " + name + "!")`:
   - This prints a greeting message, using the string stored in `name`. The `+` is used to concatenate the strings.

Python Input

Example Execution:

```mathematica
Enter your name: Alice
Hello, Alice!
```

Output Explanation:

- If the user types "Alice" when prompted, the program stores that in the variable `name`, and then it prints out `"Hello, Alice!"`.

Python Operators

3.1 Arithmetic Operators

These operators are used to perform basic arithmetic operations.

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (remainder of division)	`a % b`
`**`	Exponentiation (power)	`a ** b`
`//`	Floor division (quotient without remainder)	`a // b`

Example:

python

```python
a = 10
b = 3

print("Addition:", a + b)       # 10 + 3 = 13
print("Subtraction:", a - b)    # 10 - 3 = 7
print("Multiplication:", a * b) # 10 * 3 = 30
print("Division:", a / b)       # 10 / 3 = 3.33
print("Modulus:", a % b)        # 10 % 3 = 1
print("Exponentiation:", a ** b) # 10 ** 3 = 1000
print("Floor Division:", a // b) # 10 // 3 = 3
```

Python Operators

Explanation:

- `print("Addition:", a + b)`:
  - This adds `a` and `b` (i.e., `10 + 3 = 13`), then prints the result.
- `print("Subtraction:", a - b)`:
  - This subtracts `b` from `a` (i.e., `10 - 3 = 7`) and prints the result.
- `print("Multiplication:", a * b)`:
  - Multiplies `a` and `b` (i.e., `10 * 3 = 30`) and prints the result.
- `print("Division:", a / b)`:
  - Divides `a` by `b` (i.e., `10 / 3 = 3.33`). Python performs true division, meaning it returns a floating-point number.
- `print("Modulus:", a % b)`:
  - Computes the remainder when `a` is divided by `b` (i.e., `10 % 3 = 1`). This is called the modulus operation.
- `print("Exponentiation:", a ** b)`:
  - Raises `a` to the power of `b` (i.e., `10 ** 3 = 1000`), which means `10` raised to the power of `3`.
- `print("Floor Division:", a // b)`:
  - Performs floor division, which divides `a` by `b` and returns the integer part of the quotient (i.e., `10 // 3 = 3`).

Python Operators

Output:

```yaml
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Modulus: 1
Exponentiation: 1000
Floor Division: 3
```

3.2 Comparison Operators

These operators compare two values and return a boolean result (`True` or `False`).

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

Python Operators

Example:

```python
# Initialize variables
a = 10
b = 20
c = 10

# Comparison operations
print(a == b)  # Equal to
print(a != b)  # Not equal to
print(a > b)    # Greater than
print(a < b)    # Less than
print(a >= c)  # Greater than or equal to
print(b <= c)  # Less than or equal to
```

Explanation:

- `a == b`: checks if `a` is equal to `b`.
- `a != b`: checks if `a` is not equal to `b`.
- `a > b`: checks if `a` is greater than `b`.
- `a < b`: checks if `a` is less than `b`.
- `a >= c`: checks if `a` is greater than or equal to `c`.
- `b <= c`: checks if `b` is less than or equal to `c`.

Python Operators

Output:

```graphql
False
True
False
True
True
False
```

3.3 Logical Operators

Logical operators combine boolean expressions and return `True` or `False`.

Operator	Description	Example
`and`	Logical AND (both true)	`(a > b) and (b < c)`
`or`	Logical OR (either true)	`(a > b) or (b < c)`
`not`	Logical NOT (negates value)	`not (a > b)`

Python Operators

Example:

```python
# Initialize variables
a = 10
b = 20
c = 15

# Logical operations
print(a > b and b < c)  # Logical AND
print(a > b or b < c)   # Logical OR
print(not (a > b))      # Logical NOT
```

Explanation:

- `a > b and b < c`: `False and True` → `False`
- `a > b or b < c`: `False or True` → `True`
- `not (a > b)`: `not False` → `True`

Output:

```python
False
True
True
```

Python Operators

3.4 Assignment Operators

These operators are used to assign values to variables, often combining an arithmetic operation with the assignment.

Operator	Description	Example
`=`	Assigns value to a variable	`a = 5`
`+=`	Add and assign	`a += 3`
`-=`	Subtract and assign	`a -= 3`
`*=`	Multiply and assign	`a *= 3`
`/=`	Divide and assign	`a /= 3`
`%=`	Modulus and assign	`a %= 3`
`**=`	Exponent and assign	`a **= 2`
`//=`	Floor divide and assign	`a //= 2`

Example:

python

```python
a = 5; print(a)
a += 3; print(a)
a -= 3; print(a)
a *= 3; print(a)
a /= 3; print(a)
a %= 3; print(a)
a **= 2; print(a)
a //= 2; print(a)
```

Python Operators

1. `a = 5; print(a)`
   - The variable `a` is assigned the value 5. The `print(a)` statement outputs the value of `a`, which is `5`.
2. `a += 3; print(a)`
   - The value `3` is added to `a`, so `a` becomes `5 + 3 = 8`. The `print(a)` outputs `8`.
3. `a -= 3; print(a)`
   - The value `3` is subtracted from `a`, so `a` becomes `8 - 3 = 5`. The `print(a)` outputs `5`.
4. `a *= 3; print(a)`
   - The value `a` is multiplied by `3`, so `a` becomes `5 * 3 = 15`. The `print(a)` outputs `15`.
5. `a /= 3; print(a)`
   - The value `a` is divided by `3`, so `a` becomes `15 / 3 = 5.0`. The `print(a)` outputs `5.0` (as division in Python returns a float).
6. `a %= 3; print(a)`
   - `a` is replaced by the remainder when `a` is divided by `3`, so `a` becomes `5.0 % 3 = 2.0`. The `print(a)` outputs `2.0`.
7. `a **= 2; print(a)`
   - `a` is raised to the power of `2`, so `a` becomes `2.0 ** 2 = 4.0`. The `print(a)` outputs `4.0`.

8. `a //= 2; print(a)`

    - Floor division is performed, which divides `a` by `2` and rounds down to the nearest integer. So `a` becomes `4.0 // 2 = 2.0`. The `print(a)` outputs `2.0`.

Output:

```
5
8
5
15
5.0
2
4
2
```

Python Lists

In Python, **lists** are one of the most versatile and widely-used data structures. A list is an ordered collection of items (elements) that can hold a variety of data types, including integers, strings, floats, or even other lists.

Lists in Python:

- **Ordered Collection**: A list is an ordered sequence of elements, meaning that the order in which elements are stored is maintained. When you retrieve an element, it will always come from the same position unless the list is altered.
- **Mutable**: Lists are mutable, meaning their contents can be changed after creation. This allows you to modify elements, add new elements, or remove elements from a list.
- **Heterogeneous**: A list can store elements of different data types. For example, a list may contain integers, strings, floats, or even other lists (nested lists).
- **Indexing**: Lists use a zero-based indexing system. This allows you to access individual elements by referring to their position in the list, starting from `0` for the first element.
- **Dynamic**: The size of a list can change during the program's execution. You can add or remove items without needing to declare a fixed size for the list beforehand.

Python Lists

Here's an example that demonstrates these properties:

```python
# List can contain different data types
my_list = [10, "Python", 3.14, True]

# Lists are mutable, so elements can be changed
my_list[1] = "Coding"

# Lists maintain order
print(my_list[0])  # Output: 10
print(my_list[3])  # Output: True
```

Explanation:

1. **List Creation:**
   - `my_list = [10, "Python", 3.14, True]`: This creates a list called `my_list` that contains four elements: an integer (`10`), a string (`"Python"`), a floating-point number (`3.14`), and a Boolean value (`True`). Lists in Python can hold different data types.
2. **List Mutation:**
   - `my_list[1] = "Coding"`: This line changes the second element in the list (`"Python"`) to the string `"Coding"`. Lists in Python are **mutable**, meaning you can modify their elements after creation.
3. **Accessing Elements:**
   - `print(my_list[0])`: This prints the first element of the list, which is `10`.
   - `print(my_list[3])`: This prints the fourth element of the list, which is `True`.

Python Lists Operations and Methods

extend()

```python
my_list = [1, 2, 3]
my_list.extend([4, 5])
print(my_list)
```

Explanation:

The code initializes a list with the elements 1, 2, and 3. It then extends this list by adding the elements 4 and 5, resulting in the list containing 1, 2, 3, 4, and 5. Finally, it prints a message showing the updated list.

The output of the code is:

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

Python Lists Operations and Methods

remove()

```python
my_list = [1, 2, 3, 4, 5]
my_list.remove(3)
print(my_list)
```

Explanation:

The code initializes a list with the elements 1, 2, 3, 4, and 5. It then removes the element 3 from the list, resulting in the list containing 1, 2, 4, and 5. Finally, it prints a message showing the updated list.

Output:

```python
[1, 2, 4, 5]
```

Python Lists Operations and Methods

sort()

```python
my_list = [5, 3, 1, 4, 2]
my_list.sort()
print(my_list)
```

Explanation:

1. **Sorting**:
   - The `sort()` method is called on `my_list`. This method rearranges the elements in ascending order, modifying the original list in place. After this operation, the list will be organized from the smallest to the largest value.
2. **Printing the Result**:
   - The `print(my_list)` statement outputs the sorted list.

Output:

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

Python Tuples and Methods

A **tuple** in Python is an ordered, immutable collection of items. Like lists, tuples can store a sequence of elements, but once a tuple is created, its contents cannot be changed (i.e., it's immutable). Tuples are often used when you want to ensure that data remains constant throughout the execution of a program.

Key Characteristics:

1. **Immutable**: You cannot modify, add, or remove items after a tuple is created.
2. **Ordered**: Tuples maintain the order of items, so you can access elements by index.
3. **Allows mixed data types**: A tuple can contain items of different data types (e.g., integers, strings, floats, etc.).
4. **Can contain duplicates**: Tuples can contain repeated items, just like lists.

Tuple Syntax:

A tuple is created by placing the elements inside parentheses `()` and separating them with commas.

python

```python
# Creating a tuple
my_tuple = (1, "hello", 3.5)
print("Tuple:", my_tuple)  # Output: Tuple: (1, 'hello', 3.5)
```

Python Tuples and Methods

Explanation:

The code starts by defining a tuple named `my_tuple` which contains three different types of elements: an integer (`1`), a string (`"hello"`), and a floating-point number (`3.5`). Tuples in Python are a type of data structure used to store multiple items in a single variable. Unlike lists, tuples are immutable, meaning their elements cannot be changed after creation.

After defining the tuple, the `print()` function is used to display the content of the tuple along with a label `"Tuple:"`.

The output of the code is:

```python
Tuple: (1, 'hello', 3.5)
```

2. Accessing Elements:

```python
# Accessing elements by index
my_tuple = (10, "Python", 5.5)
print(my_tuple[0])
print(my_tuple[1])
print(my_tuple[2])
```

Python Tuples and Methods

Explanation:

This code accesses and prints elements from the tuple `my_tuple` by their index positions.

- `my_tuple[0]` accesses the first element, which is `10`.
- `my_tuple[1]` accesses the second element, which is `"Python"`.
- `my_tuple[2]` accesses the third element, which is `5.5`.

The output will be:

```
10
Python
5.5
```

3. Tuple Immutability:

python

```python
# Trying to modify a tuple (this will raise an error)
my_tuple = (1, 2, 3)
my_tuple[0] = 10
print(my_tuple)
```

Explanation:

The code attempts to modify a tuple, but since tuples are immutable, it raises a `TypeError` with the message: `'tuple' object does not support item assignment`.

The error message will be something like:

```php
TypeError: 'tuple' object does not support item assignment
```

4. Length of Tuple:

```python
# Finding the length of a tuple
my_tuple = ("apple", "banana", "cherry")
print(len(my_tuple))
```

Explanation:

This code calculates the length of the tuple `my_tuple`, which contains three elements: `"apple"`, `"banana"`, and `"cherry"`.

The `len()` function is used to determine how many items are in the tuple.

Python Sets and Methods

In Python, a **set** is a built-in data type that represents an unordered collection of unique elements. Sets are useful for storing items when you want to ensure that there are no duplicates and for performing mathematical set operations like union, intersection, and difference.

Key Features of Sets:

1. **Unordered**: The items in a set do not have a specific order, and you cannot access items by index.
2. **Unique Elements**: Sets automatically remove duplicate entries.
3. **Mutable**: You can add or remove elements from a set after its creation.
4. **Dynamic Size**: Sets can grow and shrink as you add or remove elements.

Creating Sets:

You can create sets using the `set()` function or by placing items in curly braces `{}`.

python

```python
# Creating a set using curly braces
my_set = {1, 2, 3, 4}
print(my_set)
```

Python Sets and Methods

Explanation:

The example demonstrates how to create a set in Python. A set is an unordered collection of unique elements, defined using curly braces. In this case, the set contains the numbers 1, 2, 3, and 4. When printed, the set displays its contents, though the order of elements may not necessarily be the same as when defined, because sets are unordered. Duplicates are not allowed, so if any repeated values were included, they would be automatically removed. The function then outputs the set of unique elements.

The output of the code:

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

Since the set contains unique numbers, and there are no duplicates, it will print the elements as shown. The order of elements may vary in some cases, but for small sets like this one, it often appears as entered.

Python Sets and Methods

1. **Adding Elements:**
   - You can add a single element using `add()` or multiple elements using `update()`.

```python
my_set = {1, 2, 3}
my_set.add(4)          # Adding one element
my_set.update([5, 6])  # Adding multiple elements
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}
```

In this code:

1. A set `my_set` is created with the values `{1, 2, 3}`.
2. The `.add(4)` method adds a single element, `4`, to the set.
3. The `.update([5, 6])` method adds multiple elements (in this case, the list `[5, 6]`) to the set. This method can take any iterable (like a list, tuple, etc.), and it will add each element individually to the set.
4. When `print(my_set)` is executed, the set now contains the values `{1, 2, 3, 4, 5, 6}`. Since sets are unordered, the elements may not appear in the exact same order when printed, but all unique values will be present.

The expected output is:

```
{1, 2, 3, 4, 5, 6}
```

2. **Removing Elements**:

   - Use `remove()` or `discard()` to remove an element. The key difference is that `remove()` raises an error if the element doesn't exist, while `discard()` does not.

```python
my_set = {1, 2, 3}
my_set.remove(2)    # Removes 2, raises KeyError if not present
my_set.discard(3)   # Removes 3, no error if not present
print(my_set)  # Output: {1}
```

Python Sets and Methods

In this code:

1. A set `my_set` is initialized with the values `{1, 2, 3}`.
2. The `.remove(2)` method removes the element `2` from the set. If the element is not present, this method raises a `KeyError`. However, since `2` is in the set, it will be removed without any error.
3. The `.discard(3)` method removes the element `3` from the set. Unlike `remove()`, it does not raise an error if the element is not present. Since `3` is in the set, it will be removed successfully.
4. When `print(my_set)` is executed, the remaining element in the set is `{1}`.

The expected output is:

```
{1}
```

Python Sets and Methods

Other Useful Methods:

- **clear()**: Removes all elements from the set.
- **copy()**: Returns a shallow copy of the set.
- **len()**: Returns the number of elements in the set.
- **isdisjoint()**: Returns `True` if two sets have no elements in common.
- **issubset()**: Checks if one set is a subset of another.
- **issuperset()**: Checks if one set is a superset of another.

```python
my_set = {1, 2, 3}
my_set.clear()  # Now my_set is empty
print(my_set)   # Output: set()
```

Python Dictionaries and Methods

In Python, a **dictionary** is a collection of key-value pairs where each key is unique. Dictionaries are mutable, meaning you can change their contents after creation, and they are unordered (prior to Python 3.7, dictionaries did not preserve order, but now they maintain insertion order).

Key Features of Dictionaries:

- **Key-Value Pairs**: Each item in a dictionary is a pair consisting of a key and a value.
- **Mutable**: You can add, remove, or modify key-value pairs.
- **Unique Keys**: Dictionary keys must be unique, though values can be duplicated.
- **Unordered**: Keys are not stored in a specific order (but insertion order is preserved from Python 3.7+).

Python Dictionaries and Methods

Creating a Dictionary

python

```python
# Creating a dictionary with key-value pairs
student = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
print("Student Dictionary:", student)
```

In this code:

1. A dictionary named `student` is created using curly braces, containing key-value pairs. The keys are `"name"`, `"age"`, and `"city"`, and their corresponding values are `"Alice"`, `25`, and `"New York"`.
2. Each key in a dictionary must be unique and is used to access its associated value.
3. The `print` statement outputs the string `"Student Dictionary:"` followed by the contents of the `student` dictionary.

Python Dictionaries and Methods

The expected output will look like this:

```css
Student Dictionary: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

Adding or Updating Values

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

# Adding a new key-value pair
student["major"] = "Computer Science"

print(student)
```

Python Dictionaries and Methods

Removing Key-Value Pairs

python

```python
student = {
    "name": "Alice",
    "age": 25,
    "city": "New York",
    "major": "Computer Science"
}

del student["age"]

print(student)
```

In this code:

1. A dictionary named `student` is created with four key-value pairs: `"name"` (with the value `"Alice"`), `"age"` (with the value `25`), `"city"` (with the value `"New York"`), and `"major"` (with the value `"Computer Science"`).
2. The `del` statement is used to remove the key `"age"` from the dictionary. After this operation, the key-value pair associated with `"age"` is deleted.
3. When `print(student)` is executed, the remaining dictionary will contain the keys `"name"`, `"city"`, and `"major"`.

The expected output will be:

arduino

```
{'name': 'Alice', 'city': 'New York', 'major': 'Computer Science'}
```

Python Strings

In Python, strings are sequences of characters enclosed within single quotes (`' '`) or double quotes (`" "`). They are immutable, meaning once a string is created, it cannot be changed.

1. String Operations

- **Concatenation**: Combining two or more strings using the `+` operator.

```python
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: "Hello World"
```

This code concatenates (joins) two strings, `str1` and `str2`, with a space in between. Here's how it works:

1. `str1 = "Hello"` assigns the string `"Hello"` to the variable `str1`.
2. `str2 = "World"` assigns the string `"World"` to the variable `str2`.
3. `result = str1 + " " + str2` concatenates the value of `str1`, a space `" "`, and the value of `str2`, then stores the result in the variable `result`.
4. `print(result)` prints the concatenated string, which is `"Hello World"`.

Python Strings

The output of the code is:

```
Hello World
```

- **Repetition**: Repeating a string multiple times using the `*` operator.

```python
result = "Hello " * 3
print(result)  # Output: "Hello Hello Hello "
```

This code repeats the string `"Hello "` three times and stores the result in the variable `result`. Here's how it works:

1. `result = "Hello " * 3`: The string `"Hello "` is multiplied by `3`, which means it is repeated three times. This creates the string `"Hello Hello Hello "`.
2. `print(result)`: This statement prints the value of `result`, which is `"Hello Hello Hello "`.

So, the output of the code is:

```
Hello Hello Hello
```

- **Length**: Finding the length of a string using the `len()` function.

```python
print(len("Hello"))  # Output: 5
```

This code calculates and prints the length of the string `"Hello"`. Here's how it works:

1. `len("Hello")`: The `len()` function is called with the argument `"Hello"`, which returns the number of characters in the string. In this case, `"Hello"` has 5 characters.
2. `print(len("Hello"))`: This statement prints the length of the string, which is `5`.

So, the output of the code is:

```
5
```

2. String Methods

- `.upper()`: Converts all characters in the string to uppercase.

```python
print("hello".upper())  # Output: "HELLO"
```

This code converts the string `"hello"` to uppercase. Here's how it works:

1. `"hello".upper()`: The `upper()` method is called on the string `"hello"`, which transforms all lowercase letters in the string to their uppercase equivalents. The result is `"HELLO"`.
2. `print("hello".upper())`: This statement prints the uppercase version of the string.

So, the output of the code is:

```
HELLO
```

Python Strings

`.lower()` : Converts all characters to lowercase.

```python
print("HELLO".lower())  # Output: "hello"
```

This code converts the string `"HELLO"` to lowercase. Here's how it works:

1. `"HELLO".lower()` : The `lower()` method is called on the string `"HELLO"`, which changes all uppercase letters in the string to their lowercase equivalents. The result is `"hello"`.
2. `print("HELLO".lower())` : This statement prints the lowercase version of the string.

So, the output of the code is:

```
hello
```

Python Strings

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

```python
print("  hello  ".strip())  # Output: "hello"
```

This code removes any leading and trailing whitespace from the string `" hello "`. Here's how it works:

1. `" hello ".strip()` : The `strip()` method is called on the string, which removes any spaces (or other whitespace characters) at the beginning and end of the string. The result is `"hello"`.
2. `print(" hello ".strip())` : This statement prints the modified string without the extra spaces.

So, the output of the code is:

```
hello
```

Python Strings

3. String Formatting

3.1 f-Strings (formatted string literals)

```python
name = "John"
age = 30
message = f"Hello, my name is {name} and I am {age} years old."
print(message)
```

This code creates a formatted string using f-strings (formatted string literals) and prints it. Here's how it works:

1. `name = "John"`: This line assigns the string `"John"` to the variable `name`.
2. `age = 30`: This line assigns the integer `30` to the variable `age`.
3. `message = f"Hello, my name is {name} and I am {age} years old."`: The variable `message` is assigned a formatted string. The f-string allows you to embed the values of variables directly into the string using curly braces `{}`. Here, `{name}` is replaced by `"John"` and `{age}` is replaced by `30`, resulting in the string: `"Hello, my name is John and I am 30 years old."`
4. `print(message)`: This statement prints the value of `message`.

So, the output of the code is:

```csharp
Hello, my name is John and I am 30 years old.
```

Python Conditional Statements

In Python, conditional statements allow you to execute different blocks of code depending on whether a condition is true or false. The most common conditional statements are `if`, `else`, and `elif` (short for "else if"). Here's how they work:

1. `if` Statement

The `if` statement checks a condition. If the condition evaluates to `True`, the block of code under the `if` statement is executed.

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

Example:

```python
x = 5
if x > 3:
    print("x is greater than 3")
```

Python Conditional Statements

This Python code checks if the variable `x` is greater than 3, and if that condition is true, it prints the message `"x is greater than 3"`. Here's a breakdown:

1. **x = 5**: This assigns the value 5 to the variable `x`.
2. **if x > 3**: This is a conditional statement that checks if the value of `x` is greater than 3.
3. **print("x is greater than 3")**: If the condition `x > 3` is true (which it is, because 5 is greater than 3), the program will execute this line and print the message `"x is greater than 3"`.

The output of the code:

```csharp
x is greater than 3
```

2. `else` Statement

The `else` statement follows an `if` statement and runs if the condition in the `if` statement is `False`.

```python
if condition:
    # Code to execute if the condition is True
else:
    # Code to execute if the condition is False
```

Python Conditional Statements

Example:

```python
x = 2
if x > 3:
    print("x is greater than 3")
else:
    print("x is not greater than 3")
```

In this code, the `else` block is added to handle the case when the condition `x > 3` is false. Here's a breakdown of the execution:

1. **x = 2**: The variable `x` is assigned the value 2.
2. **if x > 3**: This condition checks if `x` is greater than 3. Since `x` is 2, the condition is **false**.
3. **else**: Because the condition is false, the `else` block will execute.
4. **print("x is not greater than 3")**: Since the `else` block is executed, this line prints the message `"x is not greater than 3"`.

Output:

```csharp
x is not greater than 3
```

Python Conditional Statements

3. `elif` Statement

The `elif` (else if) statement is used to check multiple conditions. If the `if` statement's condition is `False`, the `elif` condition is checked. If the `elif` condition is `True`, the code block under it is executed. You can have multiple `elif` statements.

```python
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition1 is False and condition2 is True
else:
    # Code to execute if both condition1 and condition2 are False
```

Example:

```python
x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")
else:
    print("x is less than or equal to 5")
```

Python Conditional Statements

In this code, the `elif` (else if) block is used to add another condition between the `if` and `else`. Here's a breakdown:

1. **x = 7**: The variable `x` is assigned the value 7.
2. **if x > 10**: This condition checks if `x` is greater than 10. Since `x` is 7, the condition is **false**, so this block is skipped.
3. **elif x > 5**: This checks if `x` is greater than 5. Since `x` is 7 (which is greater than 5), this condition is **true**.
4. **print("x is greater than 5 but less than or equal to 10")**: Since the `elif` condition is true, this line is executed, printing the message `"x is greater than 5 but less than or equal to 10"`.
5. **else**: This block is skipped since one of the previous conditions is true.

Output:

```vbnet
x is greater than 5 but less than or equal to 10
```

Python Conditional Statements

Summary:

Statement	Description	Syntax	Example
`if`	Executes if the condition is `True`.	`if condition:`	`python if x > 3: print("x > 3")`
`else`	Executes if the `if` condition is `False`.	`else:`	`python else: print("x ≤ 3")`
`elif`	Checks another condition if `if` is `False`.	`elif condition:`	`python elif x > 5: print("x > 5")`

Python For Loops

In Python, a `for` loop is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) or any other iterable object. It allows you to execute a block of code multiple times, once for each item in the sequence.

Basic Syntax

python

```python
for item in iterable:
    # Code block to execute
```

- `item`: Represents the current element from the iterable during each iteration.
- `iterable`: The sequence or collection you're iterating over.

Example: Looping through a list

python

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Python For Loops

This code defines a list of fruits and then iterates over each item in the list, printing the name of each fruit.

- `fruits = ["apple", "banana", "cherry"]`: This creates a list called `fruits` containing three elements: "apple", "banana", and "cherry".
- `for fruit in fruits:`: This line starts a `for` loop, which will go through each item in the `fruits` list, assigning each item to the variable `fruit` one by one.
- `print(fruit)`: This prints the current value of `fruit` to the console. The loop will execute this line three times, printing "apple", "banana", and "cherry" in order.

Output:

```
apple
banana
cherry
```

Python For Loops

Looping Through a String

You can also use a `for` loop to iterate over characters in a string:

```python
word = "Python"
for letter in word:
    print(letter)
```

This code iterates through each letter in the word "Python" and prints them one by one.

- `word = "Python"`: A string variable `word` is defined with the value "Python".
- `for letter in word:`: This `for` loop goes through each character in the string `word`, assigning each character to the variable `letter` one at a time.
- `print(letter)`: This prints the current `letter` to the console. The loop will execute six times, printing each letter of "Python" on a new line.

Output:

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

Python For Loops

Using `range()` in For Loops

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

Example:

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

This code uses a `for` loop to iterate over a range of numbers from 0 to 4, and prints each number.

- `for i in range(5):` : The `range(5)` generates a sequence of numbers starting from 0 up to, but not including, 5. So, it generates the numbers 0, 1, 2, 3, and 4. The variable `i` will take on each value in this sequence.
- `print(i)` : This prints the current value of `i` to the console on each iteration of the loop.

Output:

```
0
1
2
3
4
```

Python For Loops

Looping with `else`

A `for` loop can have an optional `else` clause. The `else` block is executed after the loop finishes, unless the loop is terminated by a `break` statement.

Example:

```python
for i in range(3):
    print(i)
else:
    print("Loop is done")
```

This code loops through numbers 0 to 2 and prints each. After the loop finishes, the `else` block is executed, printing "Loop is done."

- `range(3)` generates numbers 0, 1, and 2.
- `print(i)` prints each number.
- The `else` block runs when the loop finishes, printing "Loop is done."

Output:

```vbnet
0
1
2
Loop is done
```

Python For Loops

Breaking Out of a Loop

The `break` statement can be used to exit a loop early:

```python
for i in range(5):
    if i == 3:
        break
    print(i)
```

This code uses a `for` loop to iterate over a range of numbers from 0 to 4, but it stops the loop when the value of `i` reaches 3 due to the `break` statement.

- `for i in range(5):` : This starts a loop that will iterate through the numbers 0, 1, 2, 3, and 4 (from `range(5)`).
- `if i == 3:` : This checks if the value of `i` is equal to 3.
- `break` : If `i` is 3, the loop is terminated immediately, and no further iterations are made.
- `print(i)` : This prints the current value of `i` unless the loop is stopped by the `break` statement.

Output:

```
0
1
2
```

Python For Loops

Skipping Iteration with `continue`

The `continue` statement skips the current iteration and moves to the next one:

```python
for i in range(5):
    if i == 3:
        continue
    print(i)
```

This code uses a `for` loop to iterate over a range of numbers from 0 to 4, but it skips the value 3 using the `continue` statement.

- `for i in range(5):` : This starts a loop that iterates through the numbers 0, 1, 2, 3, and 4 (from `range(5)`).
- `if i == 3:` : This checks if the value of `i` is 3.
- `continue` : If `i` is 3, the `continue` statement is executed, which skips the rest of the loop's code for that iteration. The loop then moves to the next iteration, so 3 is not printed.
- `print(i)` : This prints the current value of `i` , unless `i` is 3, in which case that iteration is skipped.

Output:

```
0
1
2
4
```

Python For Loops

Looping through a Dictionary

When looping through dictionaries, you can iterate over the keys, values, or both:

```python
person = {"name": "Kiran", "age": 25, "city": "Mumbai"}
for key, value in person.items():
    print(f"{key}: {value}")
```

This code defines a dictionary called `person` and then iterates over its key-value pairs, printing each one.

- `person = {"name": "Kiran", "age": 25, "city": "Mumbai"}`: This creates a dictionary named `person` with three key-value pairs: `"name"` mapped to `"Kiran"`, `"age"` mapped to `25`, and `"city"` mapped to `"Mumbai"`.
- `for key, value in person.items():`: This `for` loop iterates through each key-value pair in the dictionary using the `items()` method. In each iteration, the variable `key` holds the current key, and `value` holds the corresponding value.
- `print(f"{key}: {value}")`: This prints each key and its associated value in the format `key: value`.

Python For Loops

Output:

```makefile
name: Kiran
age: 25
city: Mumbai
```

Summary of for loops in Python:

Feature	Description	Example
Basic Syntax	Iterates over items in an iterable.	`for item in iterable:`
Iterating through a List	Loops through each element in a list.	`for fruit in fruits:`
Iterating through a String	Loops through each character in a string.	`for letter in word:`
Using `range()`	Generates a sequence of numbers.	`for i in range(5):`
Using `else`	Executes after the loop unless interrupted by `break`.	`for i in range(3): ... else:`
Breaking out of a Loop	Exits the loop using `break`.	`if condition: break`
Skipping Iteration	Skips the current iteration with `continue`.	`if condition: continue`
Iterating through a Dictionary	Loops through keys and values in a dictionary.	`for key, value in dict.items():`

Python While Loop

In Python, a `while` loop is used to repeatedly execute a block of code as long as a given condition is `True`. The syntax is:

```python
while condition:
    # code to execute
```

Key Points:

- The `condition` is evaluated before each iteration. If it is `True`, the loop executes the code inside the block.
- Once the `condition` becomes `False`, the loop stops.
- If the condition never becomes `False`, the loop runs indefinitely, which can cause an infinite loop unless interrupted (e.g., using a `break` statement).

Example:

```python
count = 0

while count < 5:
    print(f"Count is: {count}")
    count += 1
```

Python While Loop

Explanation:

- The variable `count` starts at 0.
- The `while` loop checks if `count` is less than 5. If true, it prints the current value and increments `count` by 1.
- When `count` reaches 5, the condition becomes `False`, and the loop ends.

The given code will produce the following output:

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

`break` and `continue` Statements:

- `break`: Exits the loop entirely.
- `continue`: Skips the current iteration and moves to the next one.

Python While Loop

Example with `break`:

```python
count = 0

while True:
    print(f"Count is: {count}")
    count += 1
    if count == 3:
        break  # Stops the loop when count reaches 3
```

Here's how the code works:

- The `while True` loop `True`.
- Inside the loop, the current value of `count` is printed, and `count` is incremented by 1 after each print.
- When `count` becomes 3, the `if` condition `count == 3` is satisfied, and the `break` statement is executed, which terminates the loop.

This will print:

```csharp
Count is: 0
Count is: 1
Count is: 2
```

Python While Loop

Example with `continue`:

```python
count = 0

while count < 5:
    count += 1
    if count == 3:
        continue  # Skip printing when count is 3
    print(f"Count is: {count}")
```

Here's how it works:

- The loop runs while `count` is less than 5.
- On each iteration, `count` is
- When `count` reaches 3, the `continue` statement is triggered, which skips the current iteration, so it doesn't print when `count` is
- For other values of `count`, it prints the current value.

This will print:

```csharp
Count is: 1
Count is: 2
Count is: 4
Count is: 5
```

Python While Loop

Summary of python while loops:

Concept	Explanation
Basic Syntax	Repeats while a condition is `True`.
Condition	Loop continues as long as the condition is `True`.
Infinite Loop	Occurs if the condition never becomes `False`.
`break` **Statement**	Exits the loop entirely.
`continue` **Statement**	Skips to the next iteration of the loop.
Variable Update	Condition variable should be updated inside the loop.

Python Functions

Python Functions

In Python, a function is a block of organized and reusable code that performs a specific task. Functions allow for better modularity and code reusability. The basic syntax for defining a function in Python is:

Syntax:

```python
def function_name(parameters):
    # Function body
    # Optional return statement
```

- **def**: This keyword is used to define a function.
- **function_name**: This is the name of the function. It should be a valid Python identifier.
- **parameters**: Optional. These are inputs to the function, which can be used inside the function.
- **Function body**: The block of code that the function will execute.
- **return**: Optional. The return statement is used to send a value back to the caller.

Python Functions

Example 1: Basic Function Example

```python
def greet():
    print("Hello, welcome to Python functions!")
```

Explanation:

- `def greet()`: This defines a function named `greet` with no parameters.
- Inside the function, the `print()` function is called to display a greeting message.
- Once defined, the function does nothing until it is called.

Calling the Function:

```python
greet()
```

Output:

```bash
Hello, welcome to Python functions!
```

Python Functions

Passing Arguments to a Function

Arguments are values passed to a function when it is called. These arguments can be used by the function to perform operations. Functions can accept one or more arguments, which are specified in the parentheses when defining the function.

Example 2: Function with Arguments

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

Explanation:

- `def greet_user(name)`: This defines a function `greet_user` that takes one argument, `name`.
- Inside the function, the `print()` statement uses the `name` argument to personalize the greeting message.

Calling the Function:

```python
greet_user("Kiran")
```

Output:

```
Hello, Kiran!
```

Python Functions

The `return` Statement

The `return` statement is used to exit a function and return a value back to the caller. A function can return a value that can be used elsewhere in the code.

Example 3: Function with a Return Statement

```python
def add_numbers(a, b):
    return a + b
```

Explanation:

- `def add_numbers(a, b)`: This defines a function `add_numbers` that takes two arguments, `a` and `b`.
- The function adds the two arguments together and returns the result using the `return` statement.

Calling the Function and Storing the Result:

```python
result = add_numbers(3, 5)
print("The sum is:", result)
```

Output:

```python
The sum is: 8
```

Python Lambda and Built-in Functions

Lambda Functions in Python

Lambda functions are small, anonymous functions used for short, quick tasks. They are defined without a name and are often used when a function is needed only once.

Syntax:

```python
lambda arguments: expression
```

- **Lambda**: The keyword used to define the function.
- **Arguments**: Input values the lambda function will take (can be multiple).
- **Expression**: The operation or logic performed on the arguments, and this expression must return a value.

Example:

```python
# Lambda function to add two numbers
add = lambda x, y: x + y
print(add(3, 4))
```

Python Lambda and Built-in Functions

This code defines a **lambda function** called `add` to add two numbers. The `lambda x, y: x + y` creates an anonymous function that takes two arguments, `x` and `y`, and returns their sum. The result is stored in the variable `add`.

Next, `print(add(3, 4))` calls the `add` function with 3 and 4 as inputs, computes their sum (7), and prints it.

The output of the code is:

```
7
```

Built-in Functions in Python

Built-in functions are functions that come with Python and are always available. Some common examples:

1. `print()`: Outputs text to the console.

```python
print("Hello!")  # Output: Hello!
```

The code `print("Hello!")` displays the text "**Hello!**" on the screen. The output is:

```
Hello!
```

Python Lambda and Built-in Functions

2. `len()` : Returns the length of a list, string, etc.

```python
len([1, 2, 3])  # Output: 3
```

The code `len([1, 2, 3])` returns the number of elements in the list `[1, 2, 3]`. The output is:

```
3
```

This means there are three items in the list. The `len()` function is used to determine the length of various data types, including lists, strings, and tuples.

3. `sum()` : Adds all elements in a list.

```python
sum([1, 2, 3])  # Output: 6
```

The code `sum([1, 2, 3])` calculates the total of all the elements in the list `[1, 2, 3]`. The output is:

```
6
```

This means that adding 1, 2, and 3 together results in 6. The `sum()` function is used to find the sum of items in an iterable, such as a list or tuple.

4. `max()` / `min()`: Finds the largest or smallest element.

```python
max([1, 2, 3])  # Output: 3
```

The code `max([1, 2, 3])` finds the maximum value in the list `[1, 2, 3]`. The output is:

```
3
```

This means that the largest number in the list is 3. The `max()` function returns the highest value from a given iterable, such as a list or tuple.

Python Classes and Objects

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of **objects**. Objects represent real-world entities and are instances of **classes**, which define their characteristics (attributes) and behavior (methods). OOP is widely used because it helps in organizing complex code and makes it easier to reuse, extend, and maintain.

Key OOP concepts:

1. **Class**: A blueprint or template for creating objects (instances).
2. **Object**: An instance of a class that holds data (attributes) and methods (behavior).
3. **Encapsulation**: Bundling of data and methods into a single unit (class), restricting access to some components.
4. **Inheritance**: A mechanism where one class can inherit properties and behavior from another class.
5. **Polymorphism**: The ability to define methods in different ways (method overriding or overloading).
6. **Abstraction**: Hiding the implementation details and exposing only the necessary parts to the user.

Python Classes and Objects

Classes in Python

A **class** is a code structure that defines attributes (data) and methods (functions) for objects. It's a blueprint used to create objects with similar properties and behaviors.

Syntax of a Class

```python
class ClassName:
    # Constructor method (called when an object is created)
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1  # Instance attribute
        self.attribute2 = attribute2  # Instance attribute

    # Method of the class
    def method_name(self):
        # Code to perform actions
        pass
```

Components of a Class:

1. **Class keyword**: The class definition starts with the keyword `class`.
2. **Attributes**: These are variables that store data related to the object. Defined inside the constructor (`__init__` method).
3. **Methods**: Functions that define the behavior of the objects.
4. `self`: Refers to the instance of the class and is used to access attributes and methods within the class.

Python Classes and Objects

Example of a Class:

python

```python
class Dog:
    # Constructor method to initialize the object
    def __init__(self, name, breed, age):
        self.name = name       # Instance attribute
        self.breed = breed     # Instance attribute
        self.age = age         # Instance attribute

    # Method to describe the dog
    def bark(self):
        return f"{self.name} says: Woof!"

    # Method to get the dog's details
    def get_details(self):
        return f"{self.name} is a {self.age}-year-old {self.breed}."
```

Explanation of the Example:

- **Class**: `Dog` is the class. It acts as a blueprint for creating dog objects.
- **Constructor** (`__init__`): This method is called automatically when a new object is created. It initializes the object's attributes (`name`, `breed`, `age`).
- **Methods**: The `bark()` and `get_details()` methods define behaviors that dog objects can perform.

Python Classes and Objects

Creating Objects from a Class:

Once a class is defined, you can create objects (instances) from it.

```python
# Creating instances (objects) of the class Dog
dog1 = Dog("Buddy", "Golden Retriever", 3)
dog2 = Dog("Max", "Bulldog", 5)

# Calling methods on the objects
print(dog1.bark())  # Output: Buddy says: Woof!
print(dog2.get_details())  # Output: Max is a 5-year-old Bulldog.
```

How It Works:

1. **dog1** and **dog2** are two objects created from the `Dog` class. Each object has its own unique set of attributes (name, breed, and age).
2. The method `bark()` is called on `dog1` and it prints `"Buddy says: Woof!"`.
3. The method `get_details()` is called on `dog2` and it returns details about the dog, `"Max is a 5-year-old Bulldog."`.

Python Classes and Objects

Here is the output when the code is executed:

1. `dog1.bark()` will return:

```yaml
Buddy says: Woof!
```

2. `dog2.get_details()` will return:

```sql
Max is a 5-year-old Bulldog.
```

Benefits of OOP:

1. **Modularity**: Code can be organized into small, manageable parts (classes).
2. **Reusability**: You can reuse classes by creating multiple objects or by inheritance.
3. **Maintainability**: Encapsulation ensures that modifications to code in one part don't affect other parts.
4. **Abstraction**: Hides unnecessary implementation details, simplifying the interface for users.

Python Classes and Objects

Summary

Subtopic	Description	Example
Class	A blueprint for creating objects. It defines attributes (data) and methods (functions) that the objects will have.	`class Car:`
Object	An instance of a class. An object holds specific data defined by the class and can interact with methods of that class.	`car1 = Car()`
Attributes	Variables inside a class that store object data. Each object can have different values for these attributes.	`self.brand`
Methods	Functions inside a class that define behaviors the objects can perform. Methods operate on the attributes of the object.	`def describe(self):`
Constructor (`__init__`)	Special method used to initialize the attributes when an object is created.	`def __init__(self, brand):`
Self	A reference to the current object. It allows access to the object's attributes and methods within the class.	`self.brand = brand`
Creating Objects	Instantiating a class to create an object. Each object has its own set of attribute values.	`car1 = Car("Tesla", "Model S")`
Accessing Attributes	Accessing the data stored in an object's attributes using the dot notation.	`print(car1.brand)`
Calling Methods	Invoking an object's methods using the dot notation.	`car1.describe()`

Python Constructors and Destructors

Object-Oriented Programming (OOP) in Python involves the use of classes and objects. Constructors and destructors are special methods used for initializing and cleaning up objects, respectively. Let's break them down:

Constructors in Python

A **constructor** is a special method called when an object of a class is instantiated. It initializes the object's attributes and prepares the object for use. In Python, the constructor is defined using the `__init__` method.

Syntax

```python
class ClassName:
    def __init__(self, parameters):
        # Initialize object attributes
        self.attribute1 = value1
        self.attribute2 = value2
        # Other initialization code
```

Explanation

- `__init__`: This method is automatically invoked when an object of the class is created.
- `self`: A reference to the current instance of the class. It is used to access variables that belong to the class.
- `parameters`: Arguments that can be passed to the constructor to initialize object attributes.

Python Constructors and Destructors

Example

python

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

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

# Creating an object of Dog class
my_dog = Dog("Buddy", 3)

# Accessing attributes
print(f"My dog name is{my_dog.name}and he is{my_dog.age}years.")
print(my_dog.bark())
```

The code defines a `Dog` class with two attributes: `name` and `age`, set through the constructor (`__init__`). It also has a method `bark()` that returns a string with the dog's name and "Woof!"

An instance of the class `my_dog` is created with the name "Buddy" and age 3. The dog's name and age are printed, along with the result of calling `bark()`:

- `my_dog.name` gives `"Buddy"`
- `my_dog.age` gives `3`
- `my_dog.bark()` returns `"Buddy says Woof!"`

Python Constructors and Destructors

The output of the code will be:

```
My dog's name is Buddy and he is 3 years old.
Buddy says Woof!
```

Destructors in Python

A **destructor** is a special method called when an object is about to be destroyed. It allows you to perform cleanup actions, such as closing files or releasing resources. In Python, the destructor is defined using the `__del__` method.

Syntax

```python
class ClassName:
    def __del__(self):
        # Cleanup code
```

Explanation

- `__del__`: This method is automatically invoked when an object is about to be destroyed. It is not guaranteed to be called immediately when an object goes out of scope, due to Python's garbage collection.
- You typically use destructors to release resources, but it's generally better to use context managers for this purpose in Python.

Python Constructors and Destructors

Example

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

    def __del__(self):
        print(f"{self.name} is being destroyed.")

# Creating an object of Dog class
my_dog = Dog("Buddy")

# Deleting the object
del my_dog
```

Here's what happens:

- When the `del my_dog` statement is executed, it calls the `__del__` method (the destructor) of the `Dog` class.
- This method prints the message `"{self.name} is being destroyed."` just before the object `my_dog` is deleted from memory.

The output of this code will be:

```
Buddy is being destroyed.
```

Python Constructors and Destructors

Here's a summary of constructors and destructors in Python:

Feature	Constructor (`__init__`)	Destructor (`__del__`)
Purpose	Initialize object attributes when an object is created.	Clean up resources or perform final actions when an object is destroyed.
Method Name	`__init__(self, ...)`	`__del__(self)`
Called When	Automatically called when an object is instantiated.	Automatically called when an object is about to be destroyed (usually when no references are left to the object).
Arguments	Can take parameters to initialize the object's attributes.	No parameters (other than `self`), usually no direct user input.
Usage	Used to set up the initial state of an object (like initializing variables).	Used to clean up resources such as closing files or network connections.
Example	`python \n def __init__(self, name): \n self.name = name`	`python \n def __del__(self): \n print("Object destroyed")`
Common Use	Initializing instance variables and preparing the object for use.	Resource management, though in Python, context managers are more commonly used for resource handling.
Output Example	`python \n Dog("Buddy") -> Object created and initialized`	`python \n del Dog -> Object destroyed message printed`

Python Inheritance

In Python, **inheritance** is a feature of object-oriented programming (OOP) that allows one class (called a *child class* or *subclass*) to inherit attributes and methods from another class (called a *parent class* or *superclass*). This helps in reusing code and organizing it in a hierarchical structure.

Key Concepts of Inheritance

1. **Parent Class (Super Class)**: The class whose properties and methods are inherited by another class.
2. **Child Class (Sub Class)**: The class that inherits the properties and methods of the parent class.

Syntax of Inheritance

To create a subclass, you specify the parent class in parentheses when defining the child class.

Python Inheritance

python

```python
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

# Child class (inherits from Animal)
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call the parent class's constructor
        self.breed = breed

    def speak(self):
        return f"{self.name} says Woof!"

# Child class (inherits from Animal)
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"
```

Python Inheritance

Explanation:

- **Parent class**: `Animal` has an `__init__()` method that initializes the `name` attribute and a `speak()` method.
- **Child class**: `Dog` inherits from `Animal` and extends it by adding a `breed` attribute. The `speak()` method is overridden to provide a specific implementation for dogs.
- The `super()` function is used to call methods from the parent class.

Example usage:

python

```python
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers")

print(dog.speak())  # Output: Buddy says Woof!
print(cat.speak())  # Output: Whiskers says Meow!
```

Types of Inheritance

1. **Single Inheritance**: A child class inherits from one parent class.
2. **Multiple Inheritance**: A child class inherits from multiple parent classes.
3. **Multilevel Inheritance**: A class can inherit from another child class.
4. **Hierarchical Inheritance**: Multiple child classes inherit from one parent class.
5. **Hybrid Inheritance**: A combination of more than one type of inheritance.

Python Polymorphism

Polymorphism in Python's Object-Oriented Programming (OOP) allows objects of different classes to be treated as objects of a common superclass. It enables methods to behave differently based on the object or class that calls them, even if they share the same method name. There are two main types of polymorphism in Python:

1. Method Overriding (Runtime Polymorphism)

This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The subclass method overrides the method in the parent class, and Python determines at runtime which method to invoke based on the object calling the method.

python

```python
class Animal:
    def speak(self):
        return "Some generic sound"

class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

# Polymorphism in action
animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())  # Output: Bark, Meow
```

Python Polymorphism

This code demonstrates **polymorphism**, where a method can behave differently depending on the object calling it.

- The base class, `Animal`, has a method `speak()` that returns `"Some generic sound"`.
- Two subclasses, `Dog` and `Cat`, inherit from `Animal` and each override the `speak()` method.
  - `Dog`'s `speak()` returns `"Bark"`.
  - `Cat`'s `speak()` returns `"Meow"`.

A list of `animals` is created, containing a `Dog` and a `Cat`. When looping through the list, each object's specific version of `speak()` is called. This is polymorphism in action: despite both being treated as `Animal` objects, they execute their own version of the `speak()` method.

The output is:

```
Bark
Meow
```

Python Polymorphism

2. Method Overloading (Compile-time Polymorphism)

In some languages, method overloading allows multiple methods with the same name but different parameters. However, Python does not support method overloading directly. Instead, you can achieve similar behavior by using default arguments or variable-length arguments (`*args` , `**kwargs`).

Example:

python

```python
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c

calc = Calculator()
print(calc.add(2, 3))       # Output: 5
print(calc.add(2, 3, 4))    # Output: 9
```

This code shows how **method overloading** can be simulated in Python using default parameters.

- **Class** `Calculator` : It has a method `add` that takes three arguments: `a` , `b` , and `c` . The parameter `c` is optional and has a default value of `0` .
- In the first call, `calc.add(2, 3)` , only two arguments are provided. The method uses `a=2` , `b=3` , and since `c` is not provided, it defaults to `0` . The result is `2 + 3 + 0 = 5` .
- In the second call, `calc.add(2, 3, 4)` , all three arguments are provided, so `a=2` , `b=3` , and `c=4` . The result is `2 + 3 + 4 = 9` .

Python Polymorphism

Output:

```
5
9
```

3. Duck Typing (Informal Polymorphism)

Python is dynamically typed, so it does not require strict type checking. If an object implements a method that is called, Python does not check the object's type, as long as the method exists. This is often referred to as "duck typing" ("If it walks like a duck and quacks like a duck, it's a duck").

Example:

python

```python
class Bird:
    def fly(self):
        return "Flies in the sky"

class Airplane:
    def fly(self):
        return "Flies in the air"

def let_fly(thing):
    print(thing.fly())

let_fly(Bird())       # Output: Flies in the sky
let_fly(Airplane())   # Output: Flies in the air
```

Python Polymorphism

This code illustrates **polymorphism** by using two unrelated classes, `Bird` and `Airplane`, that both have a method named `fly()` but return different results.

- **Class** `Bird`: The `fly()` method returns `"Flies in the sky"`.
- **Class** `Airplane`: The `fly()` method returns `"Flies in the air"`.

The function `let_fly(thing)` takes any object as an argument and calls its `fly()` method, regardless of the object's class.

- In the first call, `let_fly(Bird())`, the `fly()` method of the `Bird` class is executed, returning `"Flies in the sky"`.
- In the second call, `let_fly(Airplane())`, the `fly()` method of the `Airplane` class is executed, returning `"Flies in the air"`.

This demonstrates **polymorphism** because both objects (`Bird` and `Airplane`) respond to the same method `fly()` but with different behaviors.

Output:

```
Flies in the sky
Flies in the air
```

Python Polymorphism

Here's a summary of Python OOP Polymorphism:

Type of Polymorphism	Description	Example	Key Features
Method Overriding (Runtime Polymorphism)	A subclass provides a specific implementation of a method already defined in its superclass.	`class Dog(Animal):` `def speak(self):` `return "Bark"` `class Cat(Animal):` `def speak(self):` `return "Meow"` Both override the superclass method `speak()`.	Method behavior changes based on the object type at runtime.
Method Overloading (Compile-time Polymorphism)	Python doesn't support true overloading, but similar behavior can be achieved with default or variable-length arguments.	`def add(self, a, b, c=0):` Handles 2 or 3 arguments using a default value for `c`.	Same method name, different parameter counts (through default or variable-length arguments).
Duck Typing (Informal Polymorphism)	Python does not require strict type checking, and polymorphism is based on method existence.	`def let_fly(thing):` `thing.fly()` Can accept objects of unrelated classes like `Bird` and `Airplane`, both having a `fly()` method.	Focuses on method presence, not the object's class, allowing unrelated objects to share method names.

Python Encapsulation

Encapsulation in Python (and object-oriented programming in general) is the practice of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, called a class, and restricting access to some of the object's components. This helps to hide the internal state of the object and only expose the necessary parts, promoting modularity and reducing complexity.

Key Concepts of Encapsulation:

1. **Private Attributes/Methods**:
    - In Python, encapsulation can be enforced by prefixing the attribute or method with an underscore (`_`) or double underscore (`__`).
    - A single underscore (`_`) indicates that the attribute or method is intended for internal use, but it can still be accessed from outside (a soft convention).
    - A double underscore (`__`) enforces name mangling, which makes it harder to access the attribute or method from outside the class directly.

Python Encapsulation

Example:

```python
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self._mileage = 0
        self.__engine_status = "Off"

    def start_engine(self):
        self.__engine_status = "On"
        print(f"The engine is now {self.__engine_status}")

    def drive(self, miles):
        self._mileage += miles
        print(f"The car has now driven {self._mileage} miles")

my_car = Car("Toyota", "Camry")
my_car.start_engine()  # Works fine
my_car.drive(10)       # Works fine

print(my_car._mileage)
```

Python Encapsulation

This Python code defines a `Car` class that simulates a car's behavior with attributes and methods. It includes public attributes `make` and `model`, a protected attribute `_mileage` for tracking mileage (accessible but not recommended for outside use), and a private attribute `__engine_status` to indicate the engine's state (not accessible from outside the class). The methods `start_engine()` and `drive(miles)` allow starting the engine and increasing the mileage, respectively. An instance `my_car` is created with the make "Toyota" and model "Camry," and the engine is started before driving 10 miles. While `_mileage` can be accessed directly, attempting to access `__engine_status` will raise an `AttributeError`, illustrating encapsulation and the handling of attribute visibility in Python.

plaintext

```
The engine is now On
The car has now driven 10 miles
0
```

Python Encapsulation

2. **Getter and Setter Methods**:
    - To control how attributes are accessed or modified, we often use getter and setter methods. This allows us to add validation logic when getting or setting an attribute.

Example:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def get_balance(self):
        return self.__balance  # Getter method

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Deposit amount must be positive")

account = BankAccount(1000)
print(account.get_balance())  # Access balance via getter
account.deposit(500)          # Modify balance via setter
print(account.get_balance())
```

Python Encapsulation

The provided Python code defines a `BankAccount` class that simulates a bank account with encapsulated methods for managing the account balance. The class has a private attribute `__balance` initialized in the `__init__` method. It includes a getter method, `get_balance()`, which returns the current balance, and a `deposit(amount)` method that allows deposits while validating that the amount is positive. An instance of `BankAccount` named `account` is created with an initial balance of `1000`. The code prints the initial balance using `get_balance()`, deposits `500` into the account, and prints the updated balance again. The output of the code is `1000` for the initial balance and `1500` after the deposit, demonstrating encapsulation by restricting direct access to the private balance attribute.

When the code is executed, the output will be:

```plaintext
1000
1500
```

3. **Why Encapsulation?**:

   - **Data Protection**: By making some attributes private, we ensure that the internal state of the object cannot be changed arbitrarily, protecting it from unwanted or incorrect modification.
   - **Controlled Access**: Getters and setters allow precise control over how data is accessed and modified.
   - **Cleaner Code**: It helps keep code modular and manageable, hiding unnecessary complexity from the user of the class.

Python Encapsulation

Here's a summary of the concept of encapsulation in Python OOP:

Concept	Description
Definition	Bundling data (attributes) and methods (functions) into a single class while restricting access to some components.
Private Attributes/Methods	Prefixing with `_` (protected) or `__` (private) to control access. `_` is a convention; `__` enforces name mangling.
Getter and Setter Methods	Methods to control access and modification of private attributes, allowing validation and logic during these operations.
Data Protection	Prevents arbitrary changes to internal state, ensuring data integrity.
Controlled Access	Allows precise control over how data is accessed or modified through getter and setter methods.
Cleaner Code	Promotes modularity, making code easier to maintain and reducing complexity by hiding unnecessary details.

This table provides a concise overview of encapsulation in Python OOP,

Python Reading a File

In Python, handling and reading files is pretty straightforward. Here's a beginner-friendly guide to get you started.

1. Opening a File

You can open a file in Python using the `open()` function. This function needs at least one argument: the file name (or file path).

```python
file = open("example.txt")  # Opens a file called example.txt
```

2. Reading the Contents

Once you open the file, you can read its contents in different ways:

- **Reading the whole file at once** with `read()`:

```python
file = open("example.txt", "r")  # "r" means read mode
content = file.read()
print(content)
file.close()  # Always close the file after you're done
```

Python Reading a File

The Python code performs the following steps:

1. **Open a File**: The `open()` function is called with two arguments: the filename `"example.txt"` and the mode `"r"`, which stands for read mode. This means the file will be opened for reading its contents.
2. **Read the File**: The `read()` method is called on the file object (stored in the variable `file`). This reads the entire content of the file and stores it in the variable `content`.
3. **Print the Content**: The `print()` function is used to display the contents of the variable `content` on the screen.
4. **Close the File**: Finally, the `close()` method is called on the file object. This is important as it releases any system resources associated with the file and ensures that all changes (if any were made) are saved.

- **Reading line by line** with `readline()`:

```python
file = open("example.txt", "r")
first_line = file.readline()  # Reads the first line
second_line = file.readline()  # Reads the second line
print(first_line)
print(second_line)
file.close()
```

Python Reading a File

The Python code performs the following steps:

1. **Open a File**: The `open()` function is called with the filename `"example.txt"` and the mode `"r"` (read mode), which opens the file for reading.
2. **Read the First Line**: The `readline()` method is called on the file object (stored in the variable `file`). This reads the first line of the file and stores it in the variable `first_line`.
3. **Read the Second Line**: The `readline()` method is called again on the same file object. This reads the second line of the file and stores it in the variable `second_line`.
4. **Print the Lines**: The `print()` function is used twice: first to display the contents of `first_line`, and second to display the contents of `second_line`.
5. **Close the File**: Finally, the `close()` method is called on the file object to release any resources associated with the file and ensure that any changes (if applicable) are saved.

- **Reading all lines as a list** with `readlines()`:

```python
file = open("example.txt", "r")
lines = file.readlines()  # Returns a list of all lines
for line in lines:
    print(line.strip())  # Strip removes extra newlines
file.close()
```

Python Reading a File

The provided Python code performs the following steps:

1. **Open a File**: The `open()` function is called with the filename `"example.txt"` and the mode `"r"` (read mode), which opens the file for reading.
2. **Read All Lines**: The `readlines()` method is called on the file object (stored in the variable `file`). This reads all the lines of the file and returns them as a list, which is stored in the variable `lines`.
3. **Iterate Over the Lines**: A `for` loop is used to iterate through each line in the `lines` list. The loop processes each line one at a time.
4. **Print Each Line**: Inside the loop, the `print()` function is called for each `line`. The `strip()` method is used to remove any leading or trailing whitespace, including extra newline characters, before printing. This ensures that the output is clean and does not include unnecessary blank lines.
5. **Close the File**: Finally, the `close()` method is called on the file object to release any resources associated with the file and to ensure that any changes (if applicable) are saved.

Python Writing a File

Introduction to Writing Files in Python:

File handling in Python is a way to work with files (like text files) for storing, reading, or manipulating data. When writing files, we can create or modify a file by adding content to it. This is useful for saving data, such as the results of a program or user input. Key features of Python's file handling include:

- **Creating files**: Automatically generates a new file if it doesn't exist.
- **Writing data**: Allows you to add content to the file.
- **Modes**: You can choose between different modes (like overwriting, appending) when writing to a file.
- **Closing files**: Ensures proper management of system resources.

2) Syntax and Explanation:

The basic syntax for writing to a file in Python involves opening the file, writing to it, and then closing it.

```python
file = open("filename.txt", "w")
file.write("Hello, World!")
file.close()
```

Python Writing a File

Explanation:

- `open("filename.txt", "w")` : Opens a file called `filename.txt` in write mode. If the file doesn't exist, it creates one. The `"w"` mode means that if the file exists, it will be overwritten.
- `file.write()` : Writes the string data ("Hello, World!") to the file.
- `file.close()` : Closes the file to ensure that all data is saved and resources are freed.

3) **Various Simple Examples with Explanation:**

Example 1: Writing a single line to a file

```python
file = open("example.txt", "w")
file.write("This is a simple example.")
file.close()
```

- **Explanation**: This code creates (or overwrites) a file called `example.txt` and writes "This is a simple example." to it.

Python Writing a File

Example 2: Writing multiple lines to a file

```python
file = open("example.txt", "w")
file.write("Line 1\n")
file.write("Line 2\n")
file.write("Line 3\n")
file.close()
```

- **Explanation**: This example writes multiple lines into the file. The `\n` at the end of each string ensures each line appears on a new line in the file.

4) Use Cases of Writing Files in Python:

- **Storing user input**: You can save user-entered data into a file for later use (e.g., saving form data).
- **Logging**: Writing logs of an application's behavior into a file, useful for tracking events or debugging.
- **Saving program results**: Storing the output of a program, such as statistical results or processed data.
- **Configuration files**: Writing and storing settings or preferences that the program can load later.
- **Reports and Summaries**: Automatically generating and saving reports, summaries, or analyses in text format.

Python Appending a File

Appending a File in Python

In Python, appending to a file means adding new content at the end of an existing file without overwriting the previous content. This can be done by opening the file in *append mode* (`"a"`), which places the file pointer at the end. When the file is in append mode, any data written will go directly to the end of the file. If the file doesn't exist, Python will create it automatically, making this method especially useful for tasks like logging or maintaining entries over time.

Syntax and Explanation

```python
file = open("filename.txt", "a")  # Open in append mode
file.write("New content to append\n")  # Write content to end of file
file.close()  # Close the file after writing
```

- `open("filename.txt", "a")` : Opens `filename.txt` in append mode. If the file doesn't exist, it will be created.
- `file.write("New content to append\n")` : Appends the specified text to the end of the file. Using `\n` ensures that new content starts on a new line.
- `file.close()` : Closes the file, freeing up resources.

Python Appending a File

Examples and Explanation

1. **Appending a Single Line to a File**

```python
file = open("log.txt", "a")
file.write("User logged in at 12:00 PM\n")
file.close()
```

- **Explanation**: This code appends `"User logged in at 12:00 PM"` to `log.txt` and closes the file after writing. The newline (`\n`) ensures that the next line starts on a new line.

2. **Appending Multiple Lines**

```python
file = open("data.txt", "a")
file.write("Item 1: Apples\n")
file.write("Item 2: Bananas\n")
file.close()
```

- **Explanation**: This code appends two lines to `data.txt`. Each call to `write()` adds a new line to the end of the file.

Python Appending a File

3. **Checking File Content after Appending**

```python
file = open("notes.txt", "a")
file.write("This is a new note.\n")
file.close()

# Reopen the file in read mode to verify the appended content
file = open("notes.txt", "r")
content = file.read()
file.close()
print(content)
```

- **Explanation**: Here, `notes.txt` is first opened in append mode to add a new line. Then, it's reopened in read mode to verify the updated content. The `print(content)` statement displays the entire content of the file, including the new line.

Python Appending a File

Summary in Tabular Form

Concept	Code Example	Explanation
Basic Append	`file = open("file.txt", "a")`	Opens the file in append mode; creates it if it doesn't exist.
Append a Single Line	`file.write("New line\n")`	Adds a new line at the end of the file.
Close File Manually	`file.close()`	Manually closes the file after writing, freeing resources.
Append Multiple Lines	`file.write("Line 1\n"); file.write("Line 2\n")`	Appends multiple lines by calling `write()` for each line, adding `\n` at the end.
Verify File Content	`file = open("file.txt", "r")`	Reopens the file in read mode after appending, allowing you to check the updated content.

Python Exception Handling

Python Exception Handling Overview

In Python, exception handling allows you to manage errors that may occur during the execution of a program. Rather than abruptly stopping the program when an error is encountered, exception handling lets you catch and respond to these errors gracefully, allowing the program to continue running or take corrective action. It uses `try`, `except`, `else`, and `finally` blocks to define how your program should behave when an exception occurs. This ensures that essential steps (like resource cleanup) can be executed regardless of whether an error happens.

Syntax and Explanation

The basic structure of exception handling in Python is as follows:

```python
try:
    # Code that may cause an exception
except ExceptionType:
    # Code to handle the exception
else:
    # Code that runs if no exception occurs
finally:
    # Code that always runs, regardless of an exception
```

Python Exception Handling

- `try` **block**: This block contains the code that might throw an exception. If an error occurs in this block, the exception is immediately raised, and the program moves to the appropriate `except` block.
- `except` **block**: This block catches and handles the exception raised in the `try` block. You can specify multiple `except` blocks to handle different types of exceptions.
- `else` **block**: This optional block runs if no exception occurs in the `try` block. It is useful for code that should only execute if the `try` block is successful.
- `finally` **block**: This block always executes, regardless of whether an exception was raised or not. It's often used for cleanup actions, such as closing a file or releasing resources.

Simple Examples

Example 1: Catching a Specific Exception

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

Explanation: This code attempts to divide by zero, which raises a `ZeroDivisionError`. The `except` block catches the exception and prints an error message instead of crashing.

Python Exception Handling

Example 2: Handling Multiple Exceptions

```python
try:
    result = int("hello")
except ValueError:
    print("Error: Cannot convert to integer.")
except TypeError:
    print("Error: Wrong type provided.")
```

Explanation: This code tries to convert a string to an integer, which raises a `ValueError`. The `except ValueError` block catches it and handles it with an error message.

Example 3: Using `else` Block

```python
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print("Division successful, result:", result)
```

Explanation: The division here is successful, so the `else` block executes, displaying the result.

Python Exception Handling

Example 4: Using `finally` for Cleanup

```python
try:
    file = open("example.txt", "r")
except FileNotFoundError:
    print("File not found!")
finally:
    print("Cleaning up...")
```

Explanation: The `finally` block executes whether or not the `FileNotFoundError` occurs, making it ideal for resource cleanup, like closing files.

Example 5: Raising an Exception

```python
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    return age

try:
    check_age(-5)
except ValueError as e:
    print(e)
```

Explanation: Here, `check_age` raises a `ValueError` if an invalid age is provided. The `except` block then catches and prints the exception message.

Python Exception Handling

Summary Table

Block	Purpose	Example
`try`	Contains code that might raise an exception	`try: result = 10 / 0`
`except`	Catches and handles a specific exception	`except ZeroDivisionError: print("Error")`
`else`	Executes if no exception occurs in the `try` block	`else: print("Success")`
`finally`	Executes whether or not an exception occurs	`finally: print("Cleanup")`
`raise`	Manually raises an exception	`raise ValueError("Invalid input!")`

This format provides a concise overview and demonstrates how Python's exception handling can make your code more resilient and easier to troubleshoot.

Python Imports

Importing a Python File

In Python, importing a file (or "module") allows you to reuse code across different files, making your project more organized and maintainable. By importing, you can access functions, variables, or classes from one file within another without rewriting the same code. The imported file must be in the same directory or in a location accessible by Python's path settings.

Syntax

The syntax for importing a Python file is:

```python
import filename
```

Or to import specific functions or variables:

```python
from filename import function_name
```

Or, you can rename the imported file or function for easier reference:

```python
import filename as alias
from filename import function_name as alias
```

Python Imports

- `import filename`: Imports all functions, variables, and classes from `filename.py`.
- `from filename import function_name`: Imports only the specified `function_name` from `filename.py`.
- `import filename as alias`: Allows you to refer to `filename` using a shorter name (`alias`).
- `from filename import function_name as alias`: Allows you to refer to `function_name` using a shorter name.

Examples

1. **Basic Import**
   - **Files**:
     - `helper.py`:

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

     - `main.py`:

```python
import helper
print(helper.greet("Kiran"))  # Output: Hello, Kiran!
```

   - **Explanation**: The `helper` module is imported into `main.py`. The `greet` function is accessed using `helper.greet()`.

Python Imports

2. **Importing Specific Functions**

   - **Files**:

     - `helper.py`:

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

       def farewell(name):
           return f"Goodbye, {name}!"
       ```

     - `main.py`:

       ```python
       from helper import greet
       print(greet("Kiran"))  # Output: Hello, Kiran!
       ```

- **Explanation**: Only the `greet` function is imported, so `farewell` is not accessible in `main.py`. You call `greet()` directly.

Python Imports

3. **Using Aliases**

   - **Files**:

     - `helper.py`:

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

     - `main.py`:

       ```python
       import helper as h
       print(h.greet("Kiran"))  # Output: Hello, Kiran!
       ```

   - **Explanation**: The `helper` module is imported as `h`. This shortens the module name when accessing `greet`, using `h.greet()` instead.

Python Imports

Summary Table

Import Method	Syntax	Example	Description
Import entire module	`import filename`	`import helper`	Imports all functions and variables from `helper.py`.
Import specific function/variable	`from filename import function_name`	`from helper import greet`	Imports only `greet` function, not others.
Alias entire module	`import filename as alias`	`import helper as h`	Imports `helper.py` as `h`, allowing `h.greet()`.
Alias specific function/variable	`from filename import function_name as alias`	`from helper import greet as g`	Imports `greet` as `g`, allowing `g()` directly.

This modular approach makes your code cleaner, more organized, and easier to maintain.

Importing Python's Built-in Libraries

Importing Python's Built-in Libraries

In Python, importing libraries enables you to use pre-written code that simplifies complex tasks. The Python Standard Library comes with built-in modules for a range of functionalities—from math operations and file handling to date manipulation. These modules help avoid rewriting common functions and let you implement advanced functionality with minimal code. Importing a library is like bringing in a toolkit; it makes essential functions readily available to use directly in your program.

Syntax and Explanation

1. **Basic Import**

```python
import library_name
```

This imports the entire library, making all its functions accessible. To use a function from the library, prefix it with the library name.

Importing Python's Built-in Libraries

Examples and Explanations

1. **Using the `math` Library**

   - The `math` library provides mathematical functions and constants.

```python
import math
print(math.sqrt(16))
print(math.pi)
```

2. **Using the `random` Library**

   - `random` is helpful for generating random numbers, which is useful in games, simulations, or any random selection task.

```python
import random
print(random.randint(1, 10))
```

Importing Python's Built-in Libraries

3. **Using the `datetime` Library**

   - `datetime` helps manage date and time. It can provide the current date and time, format dates, and perform time-based calculations.

```python
import datetime
current_time = datetime.datetime.now()
print(current_time)  # Output: Current date and time
```

4. **Using the `os` Library**

   - The `os` library allows interaction with the operating system, like accessing files and directories.

```python
import os
print(os.getcwd())  # Output: Current working directory
```

Importing Python's Built-in Libraries

Summary Table

Library	Purpose	Example Syntax	Example Usage
`math`	Mathematical functions and constants	`import math`	`math.sqrt(25)`, `math.pi`
`random`	Generate random numbers	`import random`	`random.randint(1, 10)`
`datetime`	Date and time operations	`import datetime`	`datetime.datetime.now()`
`os`	Interact with the operating system	`import os`	`os.getcwd()`, `os.listdir()`

Importing External Libraries Into Python

About External Libraries

In Python, external libraries (also known as third-party libraries or packages) are collections of code that extend the capabilities of the language. These libraries are developed and maintained by Python's vast community and serve specialized purposes, like data analysis, machine learning, or game development, making it easier to add complex functionalities to projects. They save developers time by providing tested and optimized code, allowing a focus on high-level design and problem-solving rather than reinventing the wheel.

Syntax to Install an External Library

To install an external library in Python, the `pip` package manager is typically used. `pip` is Python's built-in tool for managing packages, making it easy to install, upgrade, and remove libraries from the Python Package Index (PyPI).

Basic Syntax:

```bash
pip install library_name
```

Explanation:

- `pip`: The command-line tool for Python package management.
- `install`: The command to install a new package.
- `library_name`: The name of the library you wish to install. For example, `pygame` for game development.

Importing External Libraries Into Python

Step-by-Step Guide to a Library (Pygame)

1. **Open a Command Prompt or Terminal**:
   - On Windows, you can open Command Prompt.
   - On macOS and Linux, open a terminal window.
2. **Run the `pip install` Command**:
   - Enter the following command to install Pygame:

     ```bash
     pip install pygame
     ```

   - This command will download and install the latest version of Pygame available on PyPI.
3. **Verify the Installation**:
   - Open a Python interactive shell by typing `python` or `python3` in your command prompt or terminal.
   - Enter the following command to check if Pygame was installed correctly:

     ```python
     import pygame
     print("Pygame version:", pygame.ver)
     ```

   - If Pygame was installed correctly, this will display the version of Pygame you just installed.

Importing External Libraries Into Python

Simple Example Using Pygame

Here's a basic example to create a simple window using Pygame. This script initializes Pygame, sets up a display, and keeps the window open until you close it.

python

```python
import pygame  # Import the Pygame library

# Initialize Pygame
pygame.init()

# Set up display
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption("My First Pygame Window")

# Main game loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    screen.fill((255, 255, 255))
    pygame.display.flip()

# Quit Pygame
pygame.quit()
```

Explanation:

- `pygame.init()`: Initializes all Pygame modules.
- `pygame.display.set_mode()`: Creates a window with the specified width and height.
- `pygame.display.set_caption()`: Sets the title of the window.
- **Event loop**: Checks for events like closing the window. If the quit event is detected, the loop ends.
- `screen.fill((255, 255, 255))`: Fills the screen with white.
- `pygame.display.flip()`: Updates the window to reflect any drawing or changes made.

Python NumPy

Introduction to NumPy

NumPy, short for "Numerical Python," is an essential Python library for numerical and scientific computing. It is designed to work with large, multi-dimensional arrays and matrices, as well as a vast collection of mathematical functions to process these arrays efficiently. Known for its performance and versatility, NumPy is widely used in data science, machine learning, and other fields that involve heavy mathematical computations. NumPy's capabilities include array operations, linear algebra, Fourier analysis, and random number generation, making it foundational in Python's scientific ecosystem.

Step-by-Step Guide: Installing NumPy

1. **Open your terminal (or command prompt)**.
2. **Install NumPy using pip**: Run the following command to install NumPy:

   ```bash
   pip install numpy
   ```

   This will download and install the latest version of NumPy.
3. **Verify Installation**: After installation, you can verify it by opening a Python interpreter and running:

   ```python
   import numpy as np
   print(np.__version__)
   ```

   If you see the version number printed, the installation was successful!

Python NumPy

Simple Examples with Explanations

1. Creating Arrays

- **1D Array**: A single-dimensional array is like a list of numbers.

```python
import numpy as np

arr1 = np.array([1, 2, 3, 4])
print(arr1)
```

Explanation: `arr1` is a 1D array containing four elements. NumPy arrays enable fast arithmetic and complex operations.

- **2D Array**: A two-dimensional array (like a matrix) has rows and columns.

```python
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
```

Explanation: `arr2` is a 2D array or matrix with two rows and three columns.

Python NumPy

2. Array Operations

- **Arithmetic Operation**: Adding a scalar to an array adds the scalar to each element.

```python
arr3 = arr1 + 10
print(arr3)
```

Explanation: Here, we add 10 to each element in `arr1`. This is called "broadcasting," where NumPy applies the operation across all elements.

- **Element-wise Multiplication**:

```python
arr4 = arr1 * arr2[0]
print(arr4)
```

Explanation: This multiplies each element in `arr1` with each element in the first row of `arr2` (element-wise multiplication).

3. Statistical Operations

- **Mean**:

```python
mean_val = np.mean(arr1)
print(mean_val)
```

Explanation: `np.mean()` calculates the average of the elements in `arr1`.

- **Sum of All Elements in a 2D Array**:

```python
sum_val = np.sum(arr2)
print(sum_val)
```

Explanation: `np.sum()` computes the total of all elements in `arr2`.

4. Random Sampling

- **Generating Random Numbers**:

```python
random_arr = np.random.rand(5)
print(random_arr)
```

Explanation: `np.random.rand(5)` generates a 1D array with five random numbers between 0 and 1.

Python NumPy

Summary Table

Feature	Description	Example Code
1D Array Creation	Creates a single-dimensional array	`arr1 = np.array([1, 2, 3, 4])`
2D Array Creation	Creates a two-dimensional matrix	`arr2 = np.array([[1, 2, 3], [4, 5, 6]])`
Arithmetic Operation	Adds scalar to each element	`arr3 = arr1 + 10`
Element-wise Multiplication	Multiplies elements by corresponding ones	`arr4 = arr1 * arr2[0]`
Mean Calculation	Calculates the mean of elements	`mean_val = np.mean(arr1)`
Sum Calculation	Adds all elements in a 2D array	`sum_val = np.sum(arr2)`
Random Sampling	Generates random numbers between 0 and 1	`random_arr = np.random.rand(5)`

NumPy is a versatile library that streamlines complex mathematical operations, making it invaluable in various scientific and data-driven fields.

Python Pandas

Pandas Overview

Pandas is an open-source Python library that simplifies data manipulation and analysis, making it essential for data science, finance, and analytics. With structures like DataFrames and Series, Pandas provides efficient handling of structured data, supporting tasks like data cleaning, aggregation, and complex transformations. Built on top of NumPy, Pandas helps you load, process, and analyze data with ease, and it integrates well with other data-focused libraries.

The main purpose of Pandas is to provide a powerful, flexible, and easy-to-use data manipulation and analysis library for Python. Here are some key functions and features that highlight its purpose:

1. Data Structures

- **DataFrame**: A two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).
- **Series**: A one-dimensional labeled array capable of holding any data type.

2. Data Manipulation

- **Indexing and Selection**: Allows for easy access to subsets of data using labels and conditions.
- **Data Alignment**: Automatically aligns data from different sources based on index labels.

3. Data Cleaning and Preparation

- **Handling Missing Data**: Functions to identify, fill, or drop missing values in datasets.
- **Data Transformation**: Ability to reshape, pivot, and merge data from various sources.

Python Pandas

4. Data Analysis

- **Aggregation and Grouping**: Tools to group data and compute statistics like mean, sum, or count on grouped data.
- **Time Series Analysis**: Supports date and time functionality, making it easier to handle time series data.

5. Input and Output

- Supports reading from and writing to a variety of file formats (CSV, Excel, JSON, SQL databases, etc.), facilitating data import and export.

6. Performance

- Designed for efficiency in handling large datasets and performing complex operations, utilizing vectorized operations to speed up computations.

7. Integration with Other Libraries

- Works seamlessly with other data analysis and visualization libraries such as NumPy, Matplotlib, and Seaborn, making it a cornerstone of the data science ecosystem in Python.

Python Pandas

Step-by-Step Guide to Installing Pandas

1. **Check Python Version**: Ensure Python is installed on your system. Run:

```bash
python --version
```

2. **Install Pandas with pip**: Run the following command in your terminal or command prompt:

```bash
pip install pandas
```

3. **Verify Installation**: Open Python and check that Pandas is installed:

```python
import pandas as pd
print(pd.__version__)
```

Python Pandas

Pandas Examples with a Single CSV File

Assume we have a CSV file, **example.csv**, with the following data:

Name	Age	City
Alice	25	New York
Bob	30	San Francisco
Charlie	35	Chicago

1. Reading the CSV File

```python
import pandas as pd
data = pd.read_csv("example.csv")
print(data)
```

2. Selecting Specific Columns

```python
selected_columns = data[['Name', 'City']]
print(selected_columns)
```

3. Filtering Rows Based on a Condition

```python
filtered_data = data[data['Age'] > 28]
print(filtered_data)
```

Python Pandas

4. Adding a New Column

```python
data['Country'] = 'USA'
print(data)
```

5. Calculating Summary Statistics for Age

```python
stats = data['Age'].describe()
print(stats)
```

Summary Table

Example	Description	Code Summary
Reading CSV File	Load data from a CSV file	`pd.read_csv("example.csv")`
Selecting Specific Columns	Extract specific columns	`data[['Name', 'City']]`
Filtering Rows by Condition	Filter rows where Age > 28	`data[data['Age'] > 28]`
Adding a New Column	Add a new column with fixed value 'USA'	`data['Country'] = 'USA'`
Summary Statistics for Age	Generate summary statistics for Age	`data['Age'].describe()`

Python Matplotlib

Matplotlib Overview

Matplotlib is a powerful and widely-used plotting library for Python that provides an object-oriented API for embedding plots into applications. It is ideal for creating static, animated, and interactive visualizations in a variety of formats. Matplotlib's flexibility and extensive functionality make it a go-to tool for data analysis, scientific research, and machine learning, enabling users to produce high-quality graphics with ease.

Functionality

Matplotlib offers a broad range of plotting capabilities, including:

- **Variety of Plots**: Users can create line plots, scatter plots, bar charts, histograms, pie charts, and more.
- **Customization**: The library allows extensive customization of plots, including colors, styles, and annotations, enabling tailored visualizations.
- **Interactivity**: Plots can be made interactive, especially in Jupyter Notebooks, allowing for dynamic exploration of data.
- **Integration**: Matplotlib works seamlessly with Pandas, making it easy to visualize data directly from data frames.
- **Export Options**: Users can save plots in various formats, including PNG, PDF, and SVG, facilitating sharing and presentation.

Python Matplotlib

Step-by-Step Guide to Installing Matplotlib and Pandas

1. **Open Command Line Interface**: This could be Command Prompt, Terminal, or an Anaconda Prompt.
2. **Install Matplotlib and Pandas**:
   - If you are using pip, run the following command:

     ```bash
     pip install matplotlib pandas
     ```

   - If you are using Anaconda, you can install them via:

     ```bash
     conda install matplotlib pandas
     ```

3. **Verify Installation**: Open a Python interpreter or a Jupyter Notebook and type:

   ```python
   import matplotlib
   import pandas as pd
   print(matplotlib.__version__)
   print(pd.__version__)
   ```

Python Matplotlib

Examples with Data Files

1. **Example 1: Line Plot**

   - **Data File**: `line_plot_data.csv`

     ```csv
     x,y
     0,0
     1,0.84
     2,0.91
     3,0.14
     4,-0.76
     5,-0.99
     ```

   - **Code**:

     ```python
     import pandas as pd
     import matplotlib.pyplot as plt

     # Load data
     data = pd.read_csv('line_plot_data.csv')

     # Create line plot
     plt.plot(data['x'], data['y'])
     plt.title('Line Plot Example')
     plt.xlabel('X-axis')
     plt.ylabel('Y-axis')
     plt.show()
     ```

Python Matplotlib

2. **Example 2: Bar Chart**

   - **Data File**: `bar_chart_data.csv`

     csv

     ```
     category,value
     A,10
     B,15
     C,7
     D,12
     ```

   - **Code**:

     python

     ```python
     import pandas as pd
     import matplotlib.pyplot as plt

     # Load data
     data = pd.read_csv('bar_chart_data.csv')

     # Create bar chart
     plt.bar(data['category'], data['value'])
     plt.title('Bar Chart Example')
     plt.xlabel('Category')
     plt.ylabel('Value')
     plt.show()
     ```

3. **Example 3: Histogram**

   - **Data File**: `histogram_data.csv`

     ```csv
     values
     1
     2
     2
     3
     4
     4
     5
     5
     5
     6
     ```

   - **Code**:

     ```python
     import pandas as pd
     import matplotlib.pyplot as plt

     # Load data
     data = pd.read_csv('histogram_data.csv')

     # Create histogram
     plt.hist(data['values'], bins=5, alpha=0.7, color='blue')
     plt.title('Histogram Example')
     plt.xlabel('Value')
     plt.ylabel('Frequency')
     plt.show()
     ```

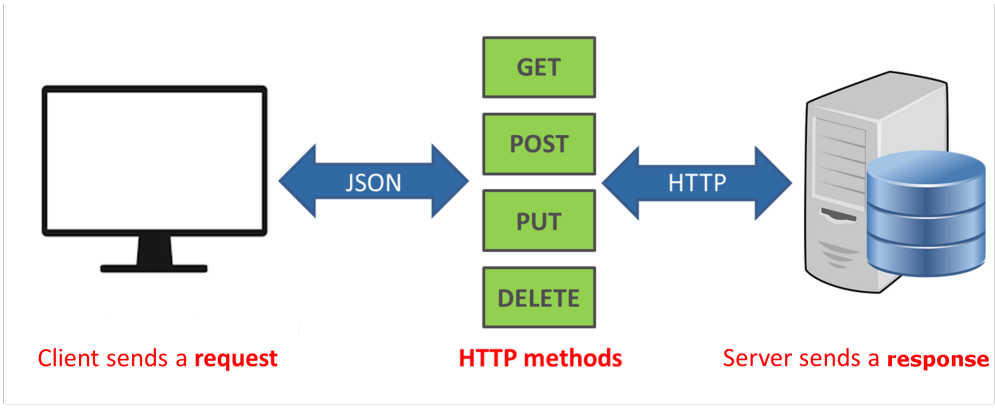
Python Matplotlib

Summary Table

Feature	Description
Variety of Plots	Create line plots, bar charts, histograms, scatter plots, etc.
Customization	Customize colors, markers, lines, and labels.
Interactivity	Interactive features for exploring plots in Jupyter Notebooks.
Integration	Works well with Pandas for seamless data handling.
Export Options	Save plots in formats like PNG, PDF, and SVG.

Python Fast API

REST API (Representational State Transfer Application Programming Interface) is a way for different software applications to communicate over the internet. Imagine you have a mobile app that needs data from a server, like a list of products. A REST API allows your app to "ask" the server for that data and "receive" it in a way that both the app and server understand.



The "methods" in a REST API are like actions that you can perform. These actions are usually called **HTTP methods**, and each one has a specific job. Here are the four main methods:

1. **GET** – *Retrieve Data*: This method is like reading information. If you want to see a list of products or details about a specific product, your app would send a GET request to the server, and the server would send back the data you need.
2. **POST** – *Create Data*: This is used when you want to add something new. For example, if you're signing up for a new account, your app sends a POST request with your info to the server, and the server creates a new account for you.
3. **PUT** – *Update Data*: When you need to change or update existing data, like editing your profile info, you use the PUT method. The app sends a PUT request with the updated info, and the server updates it for you.
4. **DELETE** – *Remove Data*: As the name suggests, DELETE is used to remove data. For example, if you want to delete a post, your app sends a DELETE request, and the server removes the post.

Python Fast API

Here's all of that simplified into table for REST API methods:

HTTP Method	Action	Purpose	Example
GET	Retrieve Data	Fetches information	Get a list of products
POST	Create Data	Adds new information	Sign up for a new account
PUT	Update Data	Changes existing information	Update your profile info
DELETE	Remove Data	Deletes information	Delete a post

This table gives a quick overview of the REST API methods and their uses!

Here’s a comparison between **Standalone Python** and a **REST API**:

Standalone Python

- **Use**: Runs local scripts and automates tasks.
- **Execution**: Runs directly on your computer.
- **Users**: Typically for one user; not designed for multiple users.
- **Internet Access**: Can access the internet, but not optimized for web requests.
- **Example**: Good for data analysis or automating simple tasks.

Python Fast API

REST API

- **Use**: Allows different applications to communicate over the web.
- **Execution**: Runs on a server and listens for requests.
- **Users**: Supports multiple users and simultaneous requests.
- **Internet Access**: Built for internet interactions using HTTP methods (GET, POST, etc.).
- **Example**: Great for web services and mobile apps.

Summary

Standalone Python is for local tasks; **REST APIs** are for web communication between applications.

What is FAST API ?

FastAPI is a high-performance Python framework for building APIs, known for its simplicity, automatic data validation, and interactive documentation. It supports asynchronous programming, making it efficient for high-concurrency tasks, suitable for scalable and maintainable applications.

Python Fast API

Functionality of FastAPI

FastAPI offers several powerful features:

- **High Performance**: It is one of the fastest frameworks in Python, capable of handling many concurrent requests with low latency, thanks to ASGI support.
- **Automatic Data Validation**: FastAPI uses Python type hints to validate data inputs and outputs automatically using Pydantic.
- **Interactive Documentation**: FastAPI automatically generates interactive documentation (Swagger UI and ReDoc) for your API, making it easy to test endpoints.
- **Asynchronous Support**: It allows asynchronous handling of requests using `async` and `await`, optimizing for tasks that require non-blocking I/O operations.

Installation Guide

1. Install FastAPI and Uvicorn (ASGI server):

   ```bash
   pip install fastapi uvicorn
   ```

2. Create a new Python file (e.g., `main.py`) to define your FastAPI application.

Python Fast API

Example 1: Basic "Hello World" Example

This example demonstrates a basic GET route that returns a simple JSON response.

python

```python
from fastapi import FastAPI

# Create FastAPI app instance
app = FastAPI()

# Define a basic route
@app.get("/")
def read_root():
    return {"message": "Hello World"}
```

To run this, save it in a file (e.g., `main.py`) and use Uvicorn to run the server:

bash

```bash
uvicorn main:app --reload
```

This creates a route at `http://127.0.0.1:8000/` that returns a JSON response: `{"message": "Hello World"}`.

Python Fast API

Example 2: Dynamic Route with Path and Query Parameters

This example shows how to create a route that captures path parameters and query parameters.

python

```python
from fastapi import FastAPI

app = FastAPI()

# Define a route with path and query parameters
@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "query": q}
```

In this example, the route `/items/{item_id}` accepts an `item_id` as a path parameter and an optional query parameter `q`. For instance:

- `GET /items/5?q=test` will return: `{"item_id": 5, "query": "test"}`

To run this, save it in a file (e.g., `main.py`) and use Uvicorn:

bash

```bash
uvicorn main:app --reload
```

Python Fast API

Example 3: Data Validation Example Using Pydantic

python

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

# In-memory storage for items
items = []

# Pydantic model for an item
class Item(BaseModel):
    name: str
    price: float
    in_stock: bool

# POST route to add an item
@app.post("/items/")
def create_item(item: Item):
    items.append(item.dict())  # Add item to the list
    return {"message": "Item successfully added"}

# GET route to retrieve all items
@app.get("/items/")
def get_items():
    return {"items": items}
```

Python Fast API

Example 4: Asynchronous Route Example

This example demonstrates an asynchronous route using `async` and `await` for non-blocking I/O operations.

python

```python
from fastapi import FastAPI
import asyncio

app = FastAPI()

# Asynchronous route that simulates a long-running task
@app.get("/delay/")
async def delay_response():
    await asyncio.sleep(3)
    return {"message": "This took 3 seconds!"}
```

This route simulates a task that takes 3 seconds to complete. The `await asyncio.sleep(3)` ensures the server doesn't block while waiting. To run this example:

- Make a GET request to `/delay/`, and it will respond after 3 seconds with:

json

```json
{
    "message": "This took 3 seconds!"
}
```

Example 5: Automatic Documentation

FastAPI automatically generates API documentation using **Swagger UI** and **ReDoc**. To demonstrate this, simply create a basic route.

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello World"}
```

Once the server is running, go to the following URLs in your browser:

- Swagger UI: `http://127.0.0.1:8000/docs`
- ReDoc: `http://127.0.0.1:8000/redoc`

These pages will show an interactive API interface with all routes and allow you to test the endpoints directly from the documentation.

Full Stack Project using AI

AI is transforming coding into a more creative, collaborative process. With tools like ChatGPT, developers can now generate, debug, and optimize code with just a prompt. This shift lets us focus on innovation and problem-solving, enabling faster, more efficient development. It's not about AI replacing coders but enhancing their capabilities, making coding more dynamic and interactive.

Now we are going to create a **Full stack project using only AI**, But first there some requirements you need to know before doing this project:

- **You need to know what is Postgres SQL and some knowledge on it's SQL queries**
- **You must have some knowledge about FastAPI and its HTTP Methods**
- **You need to know about frontend languages such as HTML, CSS, JavaScript**

Full Stack Project using AI

Now we are ready to crate our full stack project using ChatGPT. We are
Login to https://chatgpt.com

1) The first prompt is to create a virtual environment using Anaconda
So we going load up ChatGPT to kick start this project:

Prompt: Create a virtual environment using Anaconda

2) Now we going to use docker to pull Postgres SQL and initialize the database using commands:

Prompt : write a steps to spin up Postegress SQL using docker

1) docker pull postgres

2) docker run --name my_postgres –e POSTGRES_PASSWORD=ishank123 -d -p 5432:5432 postgres

3) open up pgAdmin 4 and write the table schema:

Prompt : Create a student table in PostgreSQL with columns for roll_number as the primary key, name, age, grade, and section. (Use Only Str and Int)

Full Stack Project using AI

4) Let's create the python functionality using FastAPI and test using postman:

Prompt : Create a single Python file that performs the CRUD operation son the database based on the above table and the imports:

```
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel
import psycopg2
from psycopg2 import OperationalError
from psycopg2.extras import RealDictCursor
from typing import List
from fastapi.middleware.cors import CORSMiddleware
```

5) Let’s test this python file by using postman:

Prompt : Create Postman test API for above python API

6) Now we are going to create the Frontend using HTML, CSS, JS:

Prompt : Generate UI with HTML ,CSS ,Java Script in separate files based on below API GET ,POST ,PUT DELETE and refresh list after every operation like update ,delete or create , I will give you $20

Thank You for Choosing This Python Book!

Dear Reader,

Thank you for embarking on this journey to learn Python. Your dedication and curiosity are what drive innovation and progress. I hope this book has not only met your expectations but also ignited your passion for programming.

This book was designed to equip you with the foundational and practical skills needed to excel in Python, from basic syntax to advanced concepts. As you move forward, remember: coding is not just a skill—it's a way of thinking. Keep experimenting, stay curious, and never stop learning.

If you have any questions, feedback, or suggestions, I'd love to hear from you. Your input is invaluable and helps shape better resources for learners like you. Feel free to reach out to me via:

- 📧 Email: ishankreddy2608@gmail.com
- 📽️ YouTube: Kavi Web Designs
- 💻 GitHub: IshankReddy
- 🌐 Website: kavidigitalmarketing.com

Once again, thank you for choosing this book. I wish you all the best on your programming journey. Happy coding!

Warm regards,
Ishank Reddy